GNU Emacs Lisp Reference Manual

For Emacs Version 20.3
Revision 2.5, May 1998

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

Edition 2.5
Revised for Emacs Version 20.3,
May 1998.

ISBN 1-882114-72-8

Published by the Free Software Foundation
59 Temple Place, Suite 330
Boston, MA 02111-1307 USA

Cover art by Etienne Suvasa.

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary.

To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an an-

nouncement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source

code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.
5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.
6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.
7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.
10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN

OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

> *one line to give the program's name and an idea of what it does.*
> Copyright (C) 19*yy* *name of author*
>
> This program is free software; you can redistribute it and/or
> modify it under the terms of the GNU General Public License
> as published by the Free Software Foundation; either version 2
> of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful,
> but WITHOUT ANY WARRANTY; without even the implied warranty of
> MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
> GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License
> along with this program; if not, write to the Free Software
> Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

> Gnomovision version 69, Copyright (C) 19*yy* *name of author*
> Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
> type 'show w'. This is free software, and you are welcome
> to redistribute it under certain conditions; type 'show c'
> for details.

The hypothetical commands '`show w`' and '`show c`' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than '`show w`' and '`show c`'; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright
interest in the program 'Gnomovision'
(which makes passes at compilers) written
by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

1 Introduction

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

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

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

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

This is edition 2.5.

1.1 Caveats

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

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

As you use the manual, we ask that you mark pages with corrections so you can later look them up and send them in. 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 find-

ing the text you are talking about. Also state the number of the edition you are criticizing.

Please mail comments and corrections to

`bug-lisp-manual@gnu.org`

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

1.2 Lisp History

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

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

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

A certain amount of Common Lisp emulation is available via the '`cl`' library See section "Common Lisp Extension" 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 Guile in all new GNU software that calls for extensibility.

1.3 Conventions

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

1.3.1 Some Terms

Throughout this manual, the phrases "the Lisp reader" and "the Lisp printer" refer to those routines in Lisp that convert textual representations of Lisp objects into actual Lisp objects, and vice versa. See Section 2.1 [Printed Representation], page 17, 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 appear in this font or form: `(list 1 2 3)`. Names that represent metasyntactic variables, or arguments to a function being described, appear in this font or form: *first-number*.

1.3.2 `nil` and `t`

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

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

In this manual, we use `()` when we wish to emphasize that it means the empty list, and we use `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
(not 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. The same is true of any symbol whose name starts with a colon (':'). See Section 10.2 [Constant Variables], page 148.

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 '$\mapsto$'. 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 '$\equiv$'.

```
(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. You should be aware that text displayed in the echo area is truncated to a single line.

Examples in this manual indicate printed text with '$\dashv$', irrespective of where that text goes. The value returned by evaluating the form (here `bar`) follows on a separate line.

```
(progn (print 'foo) (print 'bar))
     ⊣ foo
     ⊣ bar
     ⇒ bar
```

1.3.5 Error Messages

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

```
(+ 23 'x)
error  Wrong type argument: number-or-marker-p, x
```

1.3.6 Buffer Text Notation

Some examples show modifications to text in a buffer, with "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, '$\star$' 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 following argument name will have a value, as a variable, which is a list of all these remaining arguments. 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 17, 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 11.2 [Lambda Expressions], page 172, for a more complete description of optional and rest arguments.

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

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

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

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

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

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

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

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

1.3.7.2 A Sample Variable Description

A *variable* is a name that can hold a value. Although any variable can be set by the user, certain variables that exist specifically so that users can change them 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 Command

This function returns a string describing the version of Emacs that is running. It is useful to include this string in bug reports.

```
(emacs-version)
     "GNU Emacs 20.3.5 (i486-pc-linux-gnulibc1, X toolkit)
 of Sat Feb 14 1998 on psilocin.gnu.org"
```

Called interactively, the function prints the same information in the echo area.

emacs-build-time Variable

The value of this variable indicates the time at which Emacs was built at the local site. It is a list of three integers, like the value of `current-time` (see Section 37.5 [Time of Day], page 723).

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

emacs-version Variable

The value of this variable is the version of Emacs being run. It is a string such as `"20.3.1"`. The last number in this string is not really part of the Emacs release version number; it is incremented each time you build Emacs in any given directory.

The following two variables have existed since Emacs version 19.23:

emacs-major-version Variable

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

emacs-minor-version Variable

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

1.5 Acknowledgements

This manual was written by Robert Krawitz, Bil Lewis, Dan LaLiberte, Richard M. Stallman and Chris Welty, the volunteers of the GNU manual group, in an effort extending over several years. Robert J. Chassell helped to review and edit the manual, with the support of the Defense Advanced Research Projects Agency, ARPA Order 6082, arranged by Warren A. Hunt, Jr. of Computational Logic, Inc.

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

2 Lisp Data Types

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

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

A few fundamental object types are built into Emacs. These, from which all other types are constructed, are called *primitive types*. Each object belongs to one and only one primitive type. These types include *integer*, *float*, *cons*, *symbol*, *string*, *vector*, *subr*, *byte-code function*, plus several special types, such as *buffer*, that are related to editing. (See Section 2.4 [Editing Types], page 32.)

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

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

In most languages, the programmer must declare the data type of each variable, and the type is known by the compiler but not represented in the data. Such type declarations do not exist in Emacs Lisp. A Lisp variable can have any type of value, and it remembers whatever value you store in it, type and all.

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. The *read syntax* of an object is the format of the input accepted by the Lisp reader (the function `read`) for that object. See Chapter 18 [Read and Print], page 283.

Most objects have more than one possible read syntax. Some types of object have no read syntax, since it may not make sense to enter objects of these types directly in a Lisp program. Except for these cases, the printed representation of an object is also a read syntax for it.

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.

Every type has a printed representation. Some types have no read syntax—for example, the buffer type has none. Objects of these types are printed in *hash notation*: the characters '`#<`' followed by a descriptive string (typically the type name followed by the name of the object), and closed with a matching '`>`'. Hash notation cannot be read at all, so the Lisp reader signals the error `invalid-read-syntax` whenever it encounters '`#<`'.

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

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 8 [Evaluation], page 119). 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 18.3 [Input Functions], page 286, 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 15 [Byte Compilation], page 223). It isn't meant for source files, however.

See Section A.4 [Comment Tips], page 788, 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 −134217728 to 134217727 (28 bits; i.e., -2^{27} to $2^{28} - 1$) on most machines. (Some machines may provide a wider range.) It is important to note that the Emacs

Lisp arithmetic functions do not check for overflow. Thus `(1+ 134217727)` is −134217728 on most machines.

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

```
-1              ; The integer -1.
1               ; The integer 1.
1.              ; Also The integer 1.
+1              ; Also the integer 1.
268435457       ; Also the integer 1 on a 28-bit implementation.
```

See Chapter 3 [Numbers], page 43, for more information.

2.3.2 Floating Point Type

Emacs supports floating point numbers (though there is a compilation option to disable them). The precise range of floating point numbers is machine-specific.

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 43, 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 not often used in programs. It is far more common to work with *strings*, which are sequences composed of characters. See Section 2.3.8 [String Type], page 27.

Characters in strings, buffers, and files are currently limited to the range of 0 to 524287—nineteen bits. But not all values in that range are valid character codes. Codes 0 through 127 are ASCII codes; the rest are non-ASCII (see Chapter 32 [Non-ASCII Characters], page 629). Characters that represent keyboard input have a much wider range, to encode modifier keys such as Control, Meta and Shift.

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 a very bad idea. 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 space 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, return, and escape as '`?\a`', '`?\b`', '`?\t`', '`?\n`', '`?\v`', '`?\f`', '`?\r`', '`?\e`', respectively. Thus,

```
?\a ⇒ 7                 ; 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-[
?\\ ⇒ 92                ; backslash character, \
```

These sequences which start with backslash are also known as *escape sequences*, because backslash plays the role of an escape character; this usage has nothing to do with the character ESC.

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

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

```
?\^I ⇒ 9     ?\C-I ⇒ 9
```

In strings and buffers, the only control characters allowed are those that exist in ASCII; but for keyboard input purposes, you can turn any character into a control character with '`C-`'. The character codes for these non-ASCII control characters include the 2^{26} bit as well as the code for the corresponding non-control character. Ordinary terminals have no way of generating non-ASCII control characters, but you can generate them straightforwardly using X and other window systems.

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

```
?\^? ⇒ 127     ?\C-? ⇒ 127
```

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

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

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 (which on most machines makes it a negative number). We use high bits for this and other modifiers to make possible a wide range of basic character codes.

In a string, the 2^7 bit attached to an ASCII character indicates a meta character; thus, the meta characters that can fit in a string have codes in the range from 128 to 255, and are the meta versions of the ordinary ASCII characters. (In Emacs versions 18 and older, this convention was used for characters outside of strings as well.)

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

The case of a graphic character is indicated by its character code; for example, ASCII distinguishes between the characters '`a`' and '`A`'. But ASCII has no way to represent whether a control character is upper case or lower case. Emacs uses the 2^{25} bit to indicate that the shift key was used in typing a control character. This distinction is possible only when you use X terminals or other special terminals; ordinary terminals do not report the distinction to the computer in any way.

The 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*. Numerically, the bit values are 2^{22} for alt, 2^{23} for super and 2^{24} for hyper.

Finally, 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 `?\x8e0` for the character '`à`'.

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. 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`', instead of an actual whitespace character such as a tab.

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 use, the 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 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 sixth example because the rest of the name makes it invalid as a number.

```
foo                 ; A symbol named 'foo'.
FOO                 ; A symbol named 'FOO', different from 'foo'.
char-to-string      ; A symbol named 'char-to-string'.
```

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

2.3.5 Sequence Types

A *sequence* is a Lisp object that represents an ordered set of elements. There are two kinds of sequence in Emacs Lisp, lists and arrays. Thus, an object of type list or of type array is also considered a sequence.

Arrays are further subdivided into strings, vectors, char-tables and bool-vectors. Vectors can hold elements of any type, but string elements must be characters, and bool-vector elements must be `t` or `nil`. The characters in a string can have text properties like characters in a buffer (see Section 31.19 [Text Properties], page 610); vectors and bool-vectors do not support text properties even when their elements happen to be characters. Char-tables are like vectors except that they are indexed by any valid character code.

Lists, strings and the other array types are different, but they have important similarities. For example, all have a length *l*, and all have elements which can be indexed from zero to *l* minus one. Several functions, called sequence functions, accept any kind of sequence. For example, the function `elt` can be used to extract an element of a sequence, given its index. See Chapter 6 [Sequences Arrays Vectors], page 97.

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 pointers or slots, called the CAR slot and the CDR slot. Each slot can *point to* or hold to any Lisp object. We also say that the "the CAR of this cons cell is" whatever object its CAR slot currently points to, 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. See Chapter 5 [Lists], page 75, for functions that work on lists. Because most cons cells are used as part of lists, the phrase *list structure* has come to refer to any structure made out of cons cells.

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

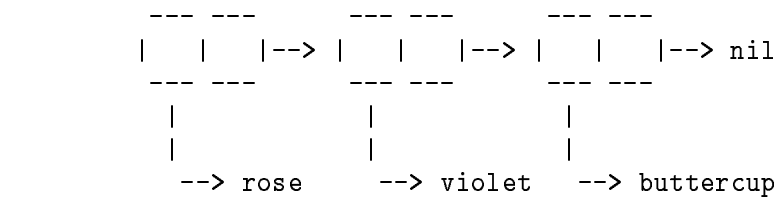
two parts, called the "address" part and the "decrement"; CAR was an instruction to extract the contents of the address part of a register, and CDR an instruction to extract the contents of the decrement. By contrast, "cons cells" are named for the function `cons` that creates them, which in turn is named for its purpose, the construction of cells.

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

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

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 points to the element, and its CDR slot points 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 point to `nil`.

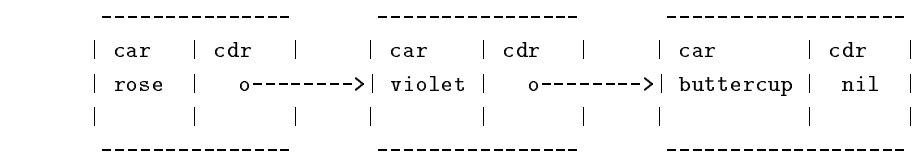
A list can be illustrated by a diagram in which the cons cells are shown as pairs of boxes, like dominoes. (The Lisp reader cannot read such an illustration; unlike the textual notation, which can be understood by both humans and computers, the box illustrations can be understood only by humans.) This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



In this diagram, each box represents a slot that can point to any Lisp object. Each pair of boxes represents a cons cell. Each arrow is a pointer 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, points to or "contains" `rose` (a symbol). The second box, holding the CDR of the first cons cell, points to the next pair of boxes, the second cons cell. The CAR of the second cons cell is `violet`, and its CDR is the third cons cell. The CDR of the third (and last) cons cell is `nil`.

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

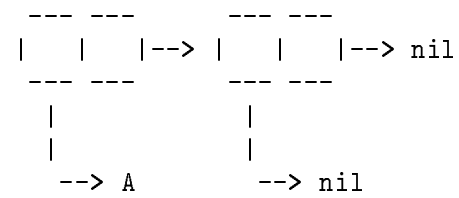


A list with no elements in it is the *empty list*; it is identical to the symbol `nil`. In other words, `nil` is both a symbol and a list.

Here are examples of lists written in Lisp syntax:

```
(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).
```

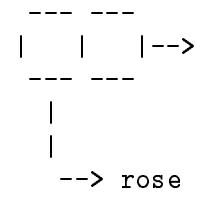
Here is the list `(A ())`, or equivalently `(A nil)`, depicted with boxes and arrows:



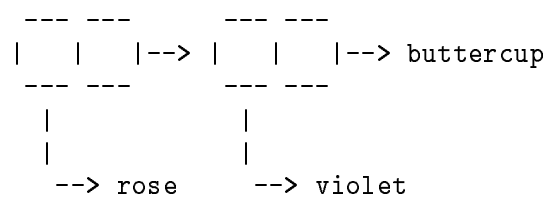
2.3.6.1 Dotted Pair Notation

Dotted pair notation is an alternative 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 therefore more general than list syntax. In the dotted pair notation, the list '`(1 2 3)`' is written as '`(1 . (2 . (3 . nil)))`'. For `nil`-terminated lists, you can use either notation, but list notation is usually clearer and more convenient. When printing a list, the dotted pair notation is only used if the CDR of a cons cell is not a list.

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

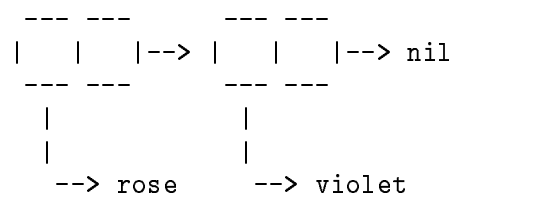


You can combine dotted pair notation with list notation to represent conveniently a chain of cons cells with a non-`nil` final CDR. You write a dot after the last element of the list, followed by the CDR of the final cons cell. For example, `(rose violet . buttercup)` is equivalent to `(rose . (violet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



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

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



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

2.3.6.2 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 91, for a further explanation of alists and for functions that work on alists.

2.3.7 Array Type

An *array* is composed of an arbitrary number of slots for pointing 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 an array of arrays.) Each type of array has its own read syntax; see the following sections for details.

The array type is contained in 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 59, for functions that operate on strings.

2.3.8.1 Syntax for Strings

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

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

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

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

2.3.8.2 Non-ASCII Characters in Strings

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

You can also represent a multibyte non-ASCII character with its character code, using a hex escape, '`\x`*nnnnnnn*', with as many digits as necessary. (Multibyte non-ASCII character codes are all greater than 256.) Any character which is not a valid hex digit terminates this construct. If the character that would follow is a hex digit, write '`\ `' (backslash and space) to terminate the hex escape—for example, '`\x8e0\ `' represents one character, '`a`' with grave accent. '`\ `' in a string constant is just like backslash-newline; it does not contribute any character to the string, but it does terminate the preceding hex escape.

Using a multibyte hex escape forces the string to 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). This forces a unibyte string.

See Section 32.1 [Text Representations], page 629, for more information about the two text representations.

2.3.8.3 Nonprinting Characters in Strings

You can use the same backslash escape-sequences in a string constant as in character literals (but do not use the question mark that begins a character constant). For example, you can write a string containing the nonprinting characters tab and `C-a`, with commas and spaces between them, like this: `"\t, \C-a"`. See Section 2.3.3 [Character Type], page 19, 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 19.

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 31.19 [Text Properties], page 610, 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 102, 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 104, for special functions to operate on char-tables. Uses of char-tables include:

- Case tables (see Section 4.9 [Case Tables], page 72).
- Character category tables (see Section 34.9 [Categories], page 680).
- Display Tables (see Section 38.14 [Display Tables], page 762).
- Syntax tables (see Chapter 34 [Syntax Tables], page 669).

2.3.11 Bool-Vector Type

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

The printed representation of a Bool-vector is like a string, except that it begins with '`#&`' followed by the length. The string constant that follows actually specifies the contents of the bool-vector as a bitmap—each "character" in the string contains 8 bits, which specify the next 8 elements of the bool-vector (1 stands for `t`, and 0 for `nil`). The least significant bits of the character correspond to the lowest indices in the bool-vector. If the length is not a multiple of 8, the printed representation shows extra elements, but these extras really make no difference.

```
(make-bool-vector 3 t)
     ⇒ #&3"\007"
(make-bool-vector 3 nil)
     ⇒ #&3"\0"
;; These are equal since only the first 3 bits are used.
(equal #&3"\377" #&3"\007")
     ⇒ t
```

2.3.12 Function Type

Just as functions in other programming languages are executable, *Lisp function* objects are pieces of executable code. However, functions in Lisp are primarily Lisp objects, and only secondarily the text which represents them. These Lisp objects are lambda expressions: lists whose first element is the symbol `lambda` (see Section 11.2 [Lambda Expressions], page 172).

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 is also called an *anonymous function* (see Section 11.7 [Anonymous Functions], page 182). A named function in Lisp is actually a symbol with a valid function in its function cell (see Section 11.4 [Defining Functions], page 177).

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 11.5 [Calling Functions], page 179.

2.3.13 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 12 [Macros], page 189, for an explanation of how to write a macro.

Warning: Lisp macros and keyboard macros (see Section 20.14 [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.

2.3.14 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 8.1.7 [Special Forms], page 124).

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 2.3.12 [Function Type], page 30, 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.
```

2.3.15 Byte-Code Function Type

The byte compiler produces *byte-code function objects*. Internally, a byte-code function object is much like a vector; however, the evaluator handles this data type specially when it appears as a function to be called. See Chapter 15 [Byte Compilation], page 223, for information about the byte compiler.

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

2.3.16 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 as a placeholder for the real definition; it says that the real definition is found in a file of Lisp code that should be loaded when necessary. The autoload object 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 14.4 [Autoload], page 215, 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 26 [Buffers], page 479). Most buffers hold the contents of a disk file (see Chapter 24 [Files], page 433) 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 27 [Windows], page 495). But a buffer need not be displayed in any window.

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

Each buffer has a designated position called *point* (see Chapter 29 [Positions], page 551). At any time, one buffer is the *current buffer*. Most editing commands act on the contents of the current buffer in the neighborhood of point. Many of the standard Emacs functions manipulate or test the characters in the current buffer; a whole chapter in this manual is devoted to describing these functions (see Chapter 31 [Text], page 575).

Several other data structures are associated with each buffer:

- a local syntax table (see Chapter 34 [Syntax Tables], page 669);
- a local keymap (see Chapter 21 [Keymaps], page 361); and,
- a list of buffer-local variable bindings (see Section 10.10 [Buffer-Local Variables], page 161).
- overlays (see Section 38.8 [Overlays], page 748).
- text properties for the text in the buffer (see Section 31.19 [Text Properties], page 610).

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 26.11 [Indirect Buffers], page 493.

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 30 [Markers], page 565, 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 34.

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 27 [Windows], page 495, for a description of the functions that work on windows.

2.4.4 Frame Type

A *frame* is a rectangle on the screen that contains one or more Emacs windows. A frame initially contains a single main window (plus perhaps a minibuffer window) which you can subdivide vertically or horizontally into smaller windows.

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 28 [Frames], page 525, for a description of the functions that work on frames.

2.4.5 Window Configuration Type

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

Window configurations do not have a read syntax; their print syntax looks like '`#<window-configuration>`'. See Section 27.16 [Window Configurations], page 521, for a description of several functions related to window configurations.

2.4.6 Frame Configuration Type

A *frame configuration* stores information about the positions, sizes, and contents of the windows in all frames. It is actually a list whose CAR is `frame-configuration` and whose CDR is an alist. Each alist element describes one frame, which appears as the CAR of that element.

See Section 28.12 [Frame Configurations], page 541, for a description of several functions related to frame configurations.

2.4.7 Process Type

The word *process* usually means a running program. Emacs itself runs in a process of this sort. However, in Emacs Lisp, a process is a Lisp object that designates a subprocess created by the Emacs process. Programs such as shells, GDB, ftp, and compilers, running in subprocesses of Emacs, extend the capabilities of Emacs.

An Emacs subprocess takes textual input from Emacs and returns textual output to Emacs for further manipulation. Emacs can also send signals to the subprocess.

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

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

See Chapter 36 [Processes], page 689, for information about functions that create, delete, return information about, send input or signals to, and receive output from processes.

2.4.8 Stream Type

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

The object `nil`, in addition to its other meanings, may be used as a stream. It stands for the value of the variable `standard-input` or `standard-output`. Also, the object `t` as a stream specifies input using the minibuffer (see Chapter 19 [Minibuffers], page 295) or output in the echo area (see Section 38.3 [The Echo Area], page 740).

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

See Chapter 18 [Read and Print], page 283, for a description of functions related to streams, including parsing and printing functions.

2.4.9 Keymap Type

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

See Chapter 21 [Keymaps], page 361, for information about creating keymaps, handling prefix keys, local as well as global keymaps, and changing key bindings.

2.4.10 Overlay Type

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

See Section 38.8 [Overlays], page 748, for how to create and use overlays.

2.5 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 5.3 [List-related Predicates], page 77.

`arrayp` See Section 6.3 [Array Functions], page 100.

`bool-vector-p`
See Section 6.7 [Bool-Vectors], page 107.

`bufferp` See Section 26.1 [Buffer Basics], page 479.

`byte-code-function-p`
See Section 2.3.15 [Byte-Code Type], page 32.

`case-table-p`
See Section 4.9 [Case Tables], page 72.

`char-or-string-p`
See Section 4.2 [Predicates for Strings], page 60.

`char-table-p`
See Section 6.6 [Char-Tables], page 104.

`commandp` See Section 20.3 [Interactive Call], page 325.

`consp` See Section 5.3 [List-related Predicates], page 77.

`display-table-p`
See Section 38.14 [Display Tables], page 762.

`floatp` See Section 3.3 [Predicates on Numbers], page 45.

`frame-configuration-p`
See Section 28.12 [Frame Configurations], page 541.

`frame-live-p`
See Section 28.5 [Deleting Frames], page 535.

`framep` See Chapter 28 [Frames], page 525.

`functionp`
: See Chapter 11 [Functions], page 171.

`integer-or-marker-p`
: See Section 30.2 [Predicates on Markers], page 566.

`integerp` See Section 3.3 [Predicates on Numbers], page 45.

`keymapp` See Section 21.3 [Creating Keymaps], page 363.

`listp` See Section 5.3 [List-related Predicates], page 77.

`markerp` See Section 30.2 [Predicates on Markers], page 566.

`wholenump`
: See Section 3.3 [Predicates on Numbers], page 45.

`nlistp` See Section 5.3 [List-related Predicates], page 77.

`numberp` See Section 3.3 [Predicates on Numbers], page 45.

`number-or-marker-p`
: See Section 30.2 [Predicates on Markers], page 566.

`overlayp` See Section 38.8 [Overlays], page 748.

`processp` See Chapter 36 [Processes], page 689.

`sequencep`
: See Section 6.1 [Sequence Functions], page 97.

`stringp` See Section 4.2 [Predicates for Strings], page 60.

`subrp` See Section 11.8 [Function Cells], page 183.

`symbolp` See Chapter 7 [Symbols], page 109.

`syntax-table-p`
: See Chapter 34 [Syntax Tables], page 669.

`user-variable-p`
: See Section 10.5 [Defining Variables], page 152.

`vectorp` See Section 6.4 [Vectors], page 102.

`window-configuration-p`
: See Section 27.16 [Window Configurations], page 521.

`window-live-p`
: See Section 27.3 [Deleting Windows], page 499.

`windowp` See Section 27.1 [Basic Windows], page 495.

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 17). But `type-of` knows nothing about non-primitive types. In most cases, it is more convenient to use type predicates than `type-of`.

type-of *object* Function

This function returns a symbol naming the primitive type of *object*. The value is one of the symbols `symbol`, `integer`, `float`, `string`, `cons`, `vector`, `char-table`, `bool-vector`, `subr`, `compiled-function`, `marker`, `overlay`, `window`, `buffer`, `frame`, `process`, or `window-configuration`.

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

2.6 Equality Predicates

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

eq *object1 object2* Function

This function returns `t` if *object1* and *object2* are the same object, `nil` otherwise. The "same object" means that a change in one will be reflected by the same change in the other.

`eq` returns `t` if *object1* and *object2* are integers with the same value. Also, since symbol names are normally unique, if the arguments are symbols with the same name, they are `eq`. For other types (e.g., lists, vectors, strings), two arguments with the same contents or elements are not necessarily `eq` to each other: they are `eq` only if they are the same object.

```
(eq 'foo 'foo)
     ⇒ t

(eq 456 456)
     ⇒ t

(eq "asdf" "asdf")
     ⇒ nil

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

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

(eq [(1 2) 3] [(1 2) 3])
     ⇒ nil

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

The `make-symbol` function returns an uninterned symbol, distinct from the symbol that is used if you write the name in a Lisp expression. Distinct symbols with the same name are not `eq`. See Section 7.3 [Creating Symbols], page 111.

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

equal *object1 object2* Function

This function returns `t` if *object1* and *object2* have equal components, `nil` otherwise. Whereas `eq` tests if its arguments are the same object, `equal` looks inside nonidentical arguments to see if their elements 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. A unibyte string never equals a multibyte string unless the contents are entirely ASCII (see Section 32.1 [Text Representations], page 629).

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

Two distinct buffers are never `equal`, even if their contents are the same.

The test for equality is implemented recursively, and circular lists may therefore cause infinite recursion (leading to an error).

3 Numbers

GNU Emacs supports two numeric data types: *integers* and *floating point numbers*. Integers are whole numbers such as −3, 0, 7, 13, and 511. Their values are exact. Floating point numbers are numbers with fractional parts, such as −4.5, 0.0, or 2.71828. They can also be expressed in exponential notation: 1.5e2 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 −134217728 to 134217727 (28 bits; i.e., -2^{27} to $2^{27} - 1$), but some machines may provide a wider range. Many examples in this chapter assume an integer has 28 bits.

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

```
  1                ; The integer 1.
  1.               ; The integer 1.
 +1                ; Also the integer 1.
 -1                ; The integer -1.
  268435457        ; Also the integer 1, due to overflow.
  0                ; The integer 0.
 -0                ; The integer 0.
```

To understand how various functions work on integers, especially the bitwise operators (see Section 3.8 [Bitwise Operations], page 52), it is often helpful to view the numbers in their binary form.

In 28-bit binary, the decimal integer 5 looks like this:

```
0000  0000 0000  0000 0000  0000 0101
```

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

The integer −1 looks like this:

```
1111  1111 1111  1111 1111  1111 1111
```

−1 is represented as 28 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  1111 1111  1111 1111  1111 1011
```

In this implementation, the largest 28-bit binary integer value is 134,217,727 in decimal. In binary, it looks like this:

```
0111  1111 1111  1111 1111  1111 1111
```

Since the arithmetic functions do not check whether integers go outside their range, when you add 1 to 134,217,727, the value is the negative integer −134,217,728:

```
(+ 1 134217727)
     ⇒ -134217728
     ⇒ 1000  0000 0000  0000 0000  0000 0000
```

Many of the functions described in this chapter accept markers for arguments in place of numbers. (See Chapter 30 [Markers], page 565.) 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.

3.2 Floating Point Basics

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

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

Most modern computers support the IEEE floating point standard, which provides for positive infinity and negative infinity as floating point values. It also provides for a class of values called NaN or "not-a-number"; numerical functions return such values in cases where there is no correct answer. For example, `(sqrt -1.0)` returns a NaN. For practical purposes, there's no significant difference between different NaN values in Emacs Lisp, and there's no rule for precisely which NaN value should be used in a particular case, so Emacs Lisp doesn't try to distinguish them. 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`'.

In addition, the value `-0.0` is distinguishable from ordinary zero in IEEE floating point (although `equal` and `=` consider them equal values).

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

logb *number* Function

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

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

3.3 Type Predicates for Numbers

The functions in this section test whether the argument is a number or whether it is a certain sort of number. The functions `integerp` and `floatp` can take any type of Lisp object as argument (the predicates 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 30.2 [Predicates on Markers], page 566.

floatp *object* Function

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

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

integerp *object* Function

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

numberp *object* Function

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

wholenump *object* Function

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

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

zerop *number* Function

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

These two forms are equivalent: `(zerop x)` ≡ `(= 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.

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.

`/=` *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.

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

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

```
(min -4 1)
     ⇒ -4
```

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

3.5 Numeric Conversions

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

float *number* Function
This returns *number* converted to floating point. If *number* is already 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. These functions accept integer arguments also, and return such arguments unchanged.

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

floor *number* &optional *divisor* Function

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

If *divisor* is specified, *number* is divided by *divisor* before the floor is taken; this uses the kind of division operation that corresponds to `mod`, rounding downward. An `arith-error` results if *divisor* is 0.

ceiling *number* Function

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

round *number* 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.

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+ 134217727)` may evaluate to −134217728, 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 9.5.3 [Errors], page 138.) 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
```

The result of `(/ -17 6)` could in principle 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 47.

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 134,217,727 produces −2 on a 28-bit machine:

```
(lsh 134217727 1)           ; left shift
     ⇒ -2
```

In binary, in the 28-bit implementation, the argument looks like this:

```
;; Decimal 134,217,727
0111  1111 1111  1111 1111  1111 1111
```

which becomes the following when left shifted:

```
;; Decimal −2
1111  1111 1111  1111 1111  1111 1110
```

ash *integer1 count* Function

`ash` (*arithmetic shift*) shifts the bits in *integer1* to the left *count* places, or to the right if *count* is negative.

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

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

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

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

```
(lsh -6 -1) ⇒ 134217725
;; Decimal −6 becomes decimal 134,217,725.
1111  1111 1111  1111 1111  1111 1010
     ⇒
0111  1111 1111  1111 1111  1111 1101
```

Here are other examples:

```
                        ;           28-bit binary values

(lsh 5 2)               ;   5  =  0000 0000 0000 0000 0000 0101
        20              ;      =  0000 0000 0000 0000 0000 0100
(ash 5 2)
        20
(lsh -5 2)              ;  -5  =  1111 1111 1111 1111 1111 1011
        -20             ;      =  1111 1111 1111 1111 1111 1100
(ash -5 2)
        -20
(lsh 5 -2)              ;   5  =  0000 0000 0000 0000 0000 0101
        1               ;      =  0000 0000 0000 0000 0000 0001
(ash 5 -2)
        1
(lsh -5 -2)             ;  -5  =  1111 1111 1111 1111 1111 1011
        4194302         ;      =  0011 1111 1111 1111 1111 1110
(ash -5 -2)             ;  -5  =  1111 1111 1111 1111 1111 1011
        -2              ;      =  1111 1111 1111 1111 1111 1110
```

logand &rest *ints-or-markers* Function

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

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

Therefore,

```
(logand 13 12)
     ⇒ 12
```

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

```
                        ;           28-bit binary values

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

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

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

logior &rest *ints-or-markers* Function

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

```
                    ;            28-bit binary values

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

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

logxor &rest *ints-or-markers* Function

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

```
                    ;            28-bit binary values

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

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

lognot *integer* Function

This function returns the logical complement of its argument: the *n*th bit is one in the result if, and only if, the *n*th bit is zero in *integer*, and vice-versa.

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

3.9 Standard Mathematical Functions

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

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]), then the result is a NaN.

acos *arg* Function
The value of `(acos` *arg*`)` is a number between 0 and pi (inclusive) whose cosine is *arg*; if, however, *arg* is out of range (outside [-1, 1]), then the result is a NaN.

atan *arg* Function
The value of `(atan` *arg*`)` is a number between −pi/2 and pi/2 (exclusive) whose tangent is *arg*.

exp *arg* Function
This is the exponential function; it returns e to the power *arg*. e is a fundamental mathematical constant also called the base of natural logarithms.

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

log10 *arg* Function
This function returns the logarithm of *arg*, with base 10. If *arg* is negative, the result is a NaN. `(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, it is truncated to fit the range of possible integer values.

sqrt *arg* Function

This returns the square root of *arg*. If *arg* is negative, the value is a NaN.

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 truly unpredictable random numbers, 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 20.5.14 [Strings of Events], page 342, for special considerations for strings of keyboard character events.

4.1 String and Character Basics

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

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

Since strings are arrays, and therefore sequences as well, you can operate on them with the general array and sequence functions. (See Chapter 6 [Sequences Arrays Vectors], page 97.) 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 100).

There are two text representations for non-ASCII characters in Emacs strings (and in buffers): unibyte and multibyte (see Section 32.1 [Text Representations], page 629). ASCII characters always occupy one byte in a string; in fact, there is no real difference between the two representation for a string which is all ASCII. For most Lisp programming, you don't need to be concerned with these two representations.

Sometimes key sequences are represented as strings. When a string is a key sequence, string elements in the range 128 to 255 represent meta characters (which are extremely 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 19, for more information about representation of meta and other modifiers for keyboard input characters.

Strings are useful for holding regular expressions. You can also match regular expressions against strings (see Section 33.3 [Regexp Search],

page 656). The functions `match-string` (see Section 33.6.2 [Simple Match Data], page 662) and `replace-match` (see Section 33.6.1 [Replacing Match], page 661) are useful for decomposing and modifying strings based on regular expression matching.

Like a buffer, a string can contain text properties for the characters in it, as well as the characters themselves. See Section 31.19 [Text Properties], page 610. 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 31 [Text], page 575, for information about functions that display strings or copy them into buffers. See Section 2.3.3 [Character Type], page 19, and Section 2.3.8 [String Type], page 27, for information about the syntax of characters and strings. See Chapter 32 [Non-ASCII Characters], page 629, for functions to convert between text representations and 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 97, and Section 6.2 [Arrays], page 99.

stringp *object* Function

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

char-or-string-p *object* Function

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

4.3 Creating Strings

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

make-string *count character* Function

This function returns a string made up of *count* repetitions of *character*. If *count* is negative, an error is signaled.

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

Other functions to compare with this one include `char-to-string` (see Section 4.6 [String Conversion], page 66), `make-vector` (see Section 6.4 [Vectors], page 102), and `make-list` (see Section 5.5 [Building Lists], page 80).

string &rest *characters* Function

This returns a string containing the characters *characters*.

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

substring *string start* &optional *end* Function

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

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

Here the index for '`a`' is 0, the index for '`b`' is 1, and the index for '`c`' is 2. Thus, three letters, '`abc`', are copied from the string `"abcdefg"`. The index 3 marks the character position up to which the substring is copied. The character whose index is 3 is actually the fourth character in the string.

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

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

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

When `nil` is used as an index, 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 97).

If the characters copied from *string* have text properties, the properties are copied into the new string also. See Section 31.19 [Text Properties], page 610.

`substring` also allows vectors for the first argument. For example:

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

A `wrong-type-argument` error is signaled if either *start* or *end* is not an integer or `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 31.2 [Buffer Contents], page 576), 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.

concat &rest *sequences* Function

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

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

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

When an argument is an integer (not a sequence of integers), it is converted to a string of digits making up the decimal printed representation of the integer. **Don't use this feature; we plan to eliminate it. If you already use this feature, change your programs now!** The proper way to convert an integer to a decimal number in this way is with `format` (see Section 4.7 [Formatting Strings], page 67) or `number-to-string` (see Section 4.6 [String Conversion], page 66).

```
(concat 137)
     ⇒ "137"
(concat 54 321)
     ⇒ "54321"
```

For information about other concatenation functions, see the description of `mapconcat` in Section 11.6 [Mapping Functions], page 180, `vconcat` in Section 6.4 [Vectors], page 102, and `append` in Section 5.5 [Building Lists], page 80.

split-string *string separators* Function

Split *string* into substrings in between matches for the regular expression *separators*. Each match for *separators* defines a splitting point; the substrings between the splitting points are made into a list, which is the value. If *separators* is `nil` (or omitted), the default is `"[ \f\t\n\r\v]+"`.

For example,

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

When there is a match adjacent to the beginning or end of the string, this does not cause a null string to appear at the beginning or end of the list:

```
(split-string "out to moo" "o+")
⇒ ("ut t" " m")
```

Empty matches do count, when not adjacent to another match:

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

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 100). `(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, of if any new character requires a different number of bytes from the character currently present at that point in *string*.

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; case is significant.

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

The function `string=` ignores the text properties of the two strings. When `equal` (see Section 2.6 [Equality Predicates], page 39) compares two strings, it uses `string=`.

If the strings contain non-ASCII characters, and one is unibyte while the other is multibyte, then they cannot be equal. See Section 32.1 [Text Representations], page 629.

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. First it scans both the strings at once to find the first pair of corresponding characters that do not match. If the lesser character of those 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 32.1 [Text Representations], page 629).

```
(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
```

string-lessp *string1 string2* Function

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

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

This function compares a specified part of *string1* with a specified part of *string2*. The specified part of *string1* runs from index *start1* up to index *end1* (default, the end of the string). The specified part of *string2* runs from index *start2* up to index *end2* (default, the end of the string).

The strings are both converted to multibyte for the comparison (see Section 32.1 [Text Representations], page 629) so that a unibyte string can be equal to a multibyte string. 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-ignore-case *key alist* Function

This function works like `assoc`, except that *key* must be a string, and comparison is done using `compare-strings`. Case differences are ignored in this comparison.

assoc-ignore-representation *key alist* Function

This function works like `assoc`, except that *key* must be a string, and comparison is done using `compare-strings`. Case differences are significant.

See also `compare-buffer-substrings` in Section 31.3 [Comparing Text], page 578, 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 33.3 [Regexp Search], page 656.

4.6 Conversion of Characters and Strings

This section describes functions for conversions between characters, strings and integers. `format` and `prin1-to-string` (see Section 18.5 [Output Functions], page 289) can also convert Lisp objects into strings. `read-from-string` (see Section 18.3 [Input Functions], page 286) can "convert" a string representation of a Lisp object into an object. The functions `string-make-multibyte` and `string-make-unibyte` convert the text representation of a string (see Section 32.2 [Converting Representations], page 630).

See Chapter 23 [Documentation], page 423, for functions that produce textual descriptions of text characters and general input events (`single-key-description` and `text-char-description`). These functions are used primarily for making help messages.

char-to-string *character* Function

This function returns a new string containing one character, *character*. This function is semi-obsolete because the function `string` is more general. See Section 4.3 [Creating Strings], page 60.

string-to-char *string* Function

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

```
(string-to-char "ABC")
     ⇒ 65
(string-to-char "xyz")
     ⇒ 120
(string-to-char "")
     ⇒ 0
(string-to-char "\000")
     ⇒ 0
```

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

number-to-string *number* Function

This function returns a string consisting of the printed representation of *number*, which may be an integer or a floating point number. The value starts with a 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 67.

string-to-number *string* &optional *base* Function

This function returns the numeric value of the characters in *string*. If *base* is non-`nil`, integers are converted in that base. If *base* is `nil`, then base ten is used. Floating point conversion always uses base ten; we have not implemented other radices for floating point numbers, because that would be much more work and does not seem useful.

The parsing skips spaces and tabs at the beginning of *string*, then reads as much of *string* as it can interpret as a number. (On some systems it ignores other whitespace at the beginning, not just spaces and tabs.) If the first character after the ignored whitespace is not a digit or a plus or minus sign, 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-int` is an obsolete alias for this function.

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

`concat` `concat` can convert a vector or a list into a string. See Section 4.3 [Creating Strings], page 60.

`vconcat` `vconcat` can convert a string into a vector. See Section 6.5 [Vector Functions], page 103.

`append` `append` can convert a string into a list. See Section 5.5 [Building Lists], page 80.

4.7 Formatting Strings

Formatting means constructing a string by substitution of computed values at various places in a constant string. This 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.

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."
```

If *string* contains more than one format specification, the format specifications correspond with 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 unpredictable behavior. 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 18.5 [Output Functions], page 289). Thus, strings are represented by their contents alone, with no '"' characters, and symbols appear without '\' characters.

If there is no corresponding object, the empty string is used.

'`%S`' Replace the specification with the printed representation of the object, made with quoting (that is, using `prin1`—see Section 18.5 [Output Functions], page 289). Thus, strings are enclosed in '"' characters, and '\' characters appear where necessary before special characters.

If there is no corresponding object, the empty string is used.

'`%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`' Replace the specification with the base-sixteen representation of an integer.

'`%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.

'`%%`' A single '`%`' is placed in the string. 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."
```

All the specification characters allow an optional numeric prefix between the '`%`' and the character. The optional numeric prefix defines the minimum width for the object. If the printed representation of the object contains fewer characters than this, then it is padded. The padding is on the left if the prefix is positive (or starts with zero) and on the right if the prefix is negative. The padding character is normally a space, but if the numeric prefix starts with a zero, zeros are used for padding. Here are some examples of padding:

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

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

`format` never truncates an object's printed representation, no matter what width you specify. Thus, you can use a numeric prefix 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, so 4 blank spaces are inserted for padding. In the second case, the string

"specification" is 13 letters wide but is not truncated. In the third case, the padding is on the right.

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

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

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

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 72.)

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

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

downcase *string-or-char* Function

This function converts a character or a string to lower case.

When the argument to `downcase` is a string, the function creates and returns a new string in which each letter in the argument that is upper case is converted to lower case. When the argument to `downcase` is a character, `downcase` returns the corresponding lower case character. This value is an integer. If the original character is lower case, or is not a letter, then the value equals the original character.

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

(downcase ?X)
     ⇒ 120
```

upcase *string-or-char* Function

This function converts a character or a string to upper case.

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

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

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

(upcase ?x)
     ⇒ 88
```

capitalize *string-or-char* Function

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

The definition of a word is any sequence of consecutive characters that are assigned to the word constituent syntax class in the current syntax table (See Section 34.2.1 [Syntax Class Table], page 670).

When the argument to `capitalize` is a character, `capitalize` has the same result as `upcase`.

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

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

(capitalize ?x)
     ⇒ 88
```

upcase-initials *string* Function

This function capitalizes the initials of the words in *string*. without altering any letters other than the initials. It returns a new string whose contents are a copy of *string*, in which each word has been 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 34.2.1 [Syntax Class Table], page 670).

```
(upcase-initials "The CAT in the hAt")
     ⇒ "The CAT In The HAt"
```

See Section 4.5 [Text Comparison], page 63, 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 31.18 [Case Changes], page 609). 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 104) 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 cyclicly permutes each equivalence class (of characters with the same canonical equivalent). (For ordinary ASCII, this would map '`a`' into '`A`' and '`A`' into '`a`', and likewise for each set of equivalent characters.)

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

Here are the functions for working with case tables:

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

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

standard-case-table Function
This returns the standard case table.

current-case-table Function
This function returns the current buffer's case table.

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

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 34 [Syntax Tables], page 669. Normally you would use these functions to change the standard case table.

set-case-syntax-pair *uc lc case-table* Function
This function specifies a pair of corresponding letters, one upper case and one lower case.

set-case-syntax-delims *l r case-table* Function
This function makes characters *l* and *r* a matching pair of case-invariant delimiters.

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

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

5 Lists

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

5.1 Lists and Cons Cells

Lists in Lisp are not a primitive data type; they are built up from *cons cells*. A cons cell is a data object that represents an ordered pair. It holds, or "points to," two Lisp objects, one labeled as the CAR, and the other labeled as the CDR. These names are traditional; see Section 2.3.6 [Cons Cell Type], page 23. CDR is pronounced "could-er."

A list is a series of cons cells chained together, one cons cell per element of the list. By convention, the CARs of the cons cells are the elements of the list, and the CDRs are used to chain the list: the CDR of each cons cell is the following cons cell. The CDR of the last cons cell is `nil`. This asymmetry between the CAR and the CDR is entirely a matter of convention; at the level of cons cells, the CAR and CDR slots have the same characteristics.

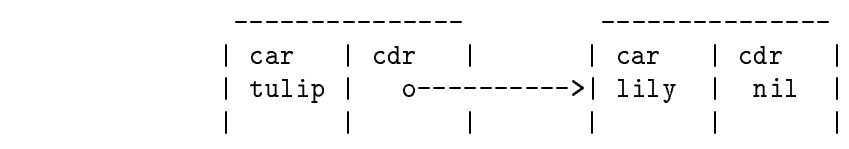
Because most cons cells are used as part of lists, the phrase *list structure* has come to mean any structure made out of cons cells.

The symbol `nil` is considered a list as well as a symbol; it is the list with no elements. For convenience, the symbol `nil` is considered to have `nil` as its CDR (and also as its CAR).

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

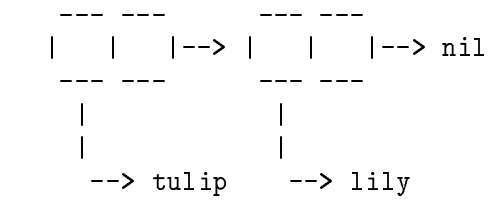
5.2 Lists as Linked Pairs of Boxes

A cons cell can be illustrated as a pair of boxes. The first box represents the CAR and the second box represents the CDR. Here is an illustration of the two-element list, `(tulip lily)`, made from two cons cells:

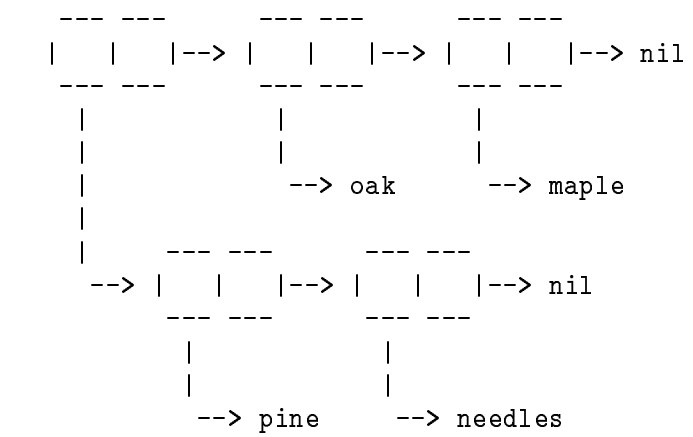


Each pair of boxes represents a cons cell. Each box "refers to", "points to" or "contains" a Lisp object. (These terms are synonymous.) The first box, which describes the CAR of the first cons cell, contains the symbol `tulip`. The arrow from the CDR box of the first cons cell to the second cons cell indicates that the CDR of the first cons cell is the second cons cell.

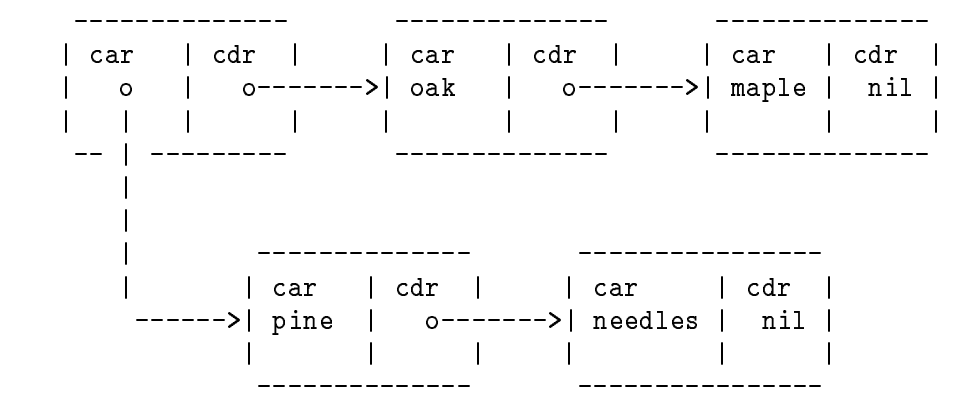
The same list can be illustrated in a different sort of box notation like this:



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



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



See Section 2.3.6 [Cons Cell Type], page 23, for the read and print syntax of cons cells and lists, and for more "box and arrow" illustrations of lists.

5.3 Predicates on Lists

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

consp *object* Function

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

atom *object* Function

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

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

listp *object* Function

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

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

nlistp *object* Function

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

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

null *object* Function

This function returns `t` if *object* is `nil`, and returns `nil` otherwise. This function is identical to `not`, but as a matter of clarity we use `null` when *object* is considered a list and `not` when it is considered a truth value (see `not` in Section 9.3 [Combining Conditions], page 132).

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

5.4 Accessing Elements of Lists

car *cons-cell* Function

This function returns the value pointed to by the first pointer of the cons cell *cons-cell*. Expressed another way, this function returns the CAR of *cons-cell*.

As a special case, if *cons-cell* is `nil`, then `car` is defined to return `nil`; therefore, any list is a valid argument for `car`. An error is signaled if the argument is not a cons cell or `nil`.

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

cdr *cons-cell* Function

This function returns the value pointed to by the second pointer of the cons cell *cons-cell*. Expressed another way, this function returns the CDR of *cons-cell*.

As a special case, if *cons-cell* is `nil`, then `cdr` is defined to return `nil`; therefore, any list is a valid argument for `cdr`. An error is signaled if the argument is not a cons cell or `nil`.

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

car-safe *object* Function

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

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

cdr-safe *object* Function

This function lets you take the CDR of a cons cell while avoiding errors for other data types. It returns the CDR of *object* if *object* is a cons cell,

`nil` otherwise. This is in contrast to `cdr`, which signals an error if *object* is not a list.

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

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 97.

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)
```

safe-length *list* Function

This function returns the length of *list*, with no risk of either an error or an infinite loop.

If *list* is not really a list, `safe-length` returns 0. If *list* is circular, it returns a finite value which is at least the number of distinct elements.

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 97.

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*`)`.

5.5 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 fundamental function used to build 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*. For example:

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

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

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

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

make-list *length object* Function

This function creates a list of length *length*, in which all the elements have the identical value *object*. Compare `make-list` with `make-string` (see Section 4.3 [Creating Strings], page 60).

```
(make-list 3 'pigs)
     ⇒ (pigs pigs pigs)
(make-list 0 'pigs)
     ⇒ nil
```

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 86, 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.

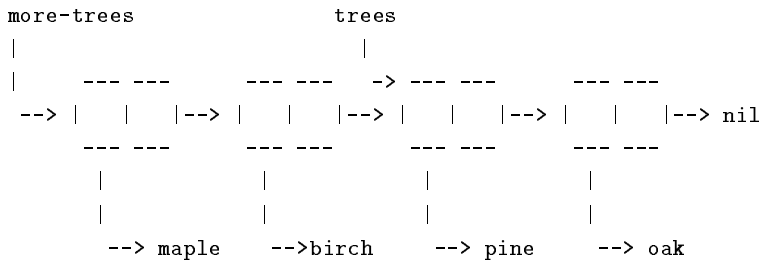
The `append` function also allows integers as arguments. It converts them to strings of digits, making up the decimal print representation of the integer, and then uses the strings instead of the original integers. **Don't use this feature; we plan to eliminate it. If you already use this feature, change your programs now!** The proper way to convert an integer to a decimal number in this way is with `format` (see Section 4.7 [Formatting Strings], page 67) or `number-to-string` (see Section 4.6 [String Conversion], page 66).

Here is an example of using `append`:

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

```
trees
     ⇒ (pine oak)
more-trees
     ⇒ (maple birch pine oak)
(eq trees (cdr (cdr more-trees)))
     ⇒ t
```

You can see how `append` works by looking at a box diagram. The variable `trees` is set to the list `(pine oak)` and then the variable `more-trees` is set to the list `(maple birch pine oak)`. However, the variable `trees` continues to refer to the original list:



An empty sequence contributes nothing to the value returned by `append`. As a consequence of this, a final `nil` argument forces a copy of the previous argument:

```
trees
     ⇒ (pine oak)
(setq wood (append trees nil))
     ⇒ (pine oak)
wood
     ⇒ (pine oak)
(eq wood trees)
     ⇒ nil
```

This once was the usual way to copy a list, before the function `copy-sequence` was invented. See Chapter 6 [Sequences Arrays Vectors], page 97.

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 11.5 [Calling Functions], page 179), 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)
```

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

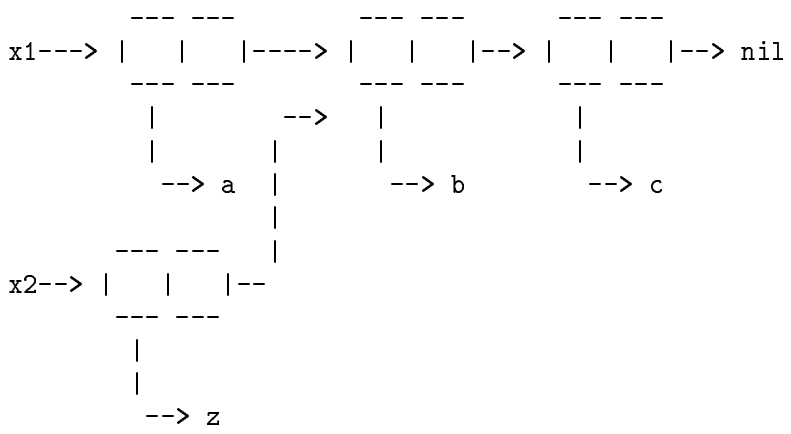
This function stores *object* as the new CAR of *cons*, replacing its previous CAR. In other words, it changes the CAR slot of *cons* to point to *object*. It returns the value *object*. For example:

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

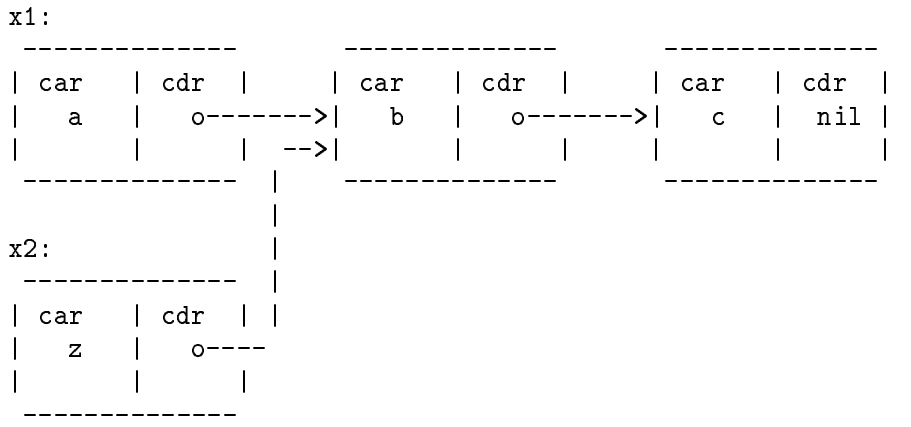
When a cons cell is part of the shared structure of several lists, storing a new CAR into the cons changes one element of each of these lists. Here is an example:

```
;; 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:



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



5.6.2 Altering the CDR of a List

The lowest-level primitive for modifying a CDR is `setcdr`:

setcdr *cons object* Function

This function stores *object* as the new CDR of *cons*, replacing its previous CDR. In other words, it changes the CDR slot of *cons* to point to *object*. It returns the value *object*.

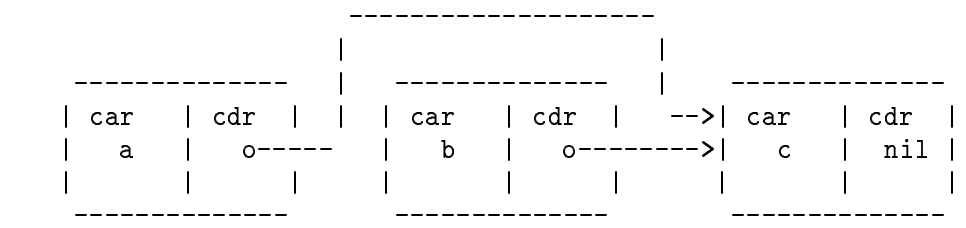
Here is an example of replacing the CDR of a list with a different list. All but the first element of the list are removed in favor of a different sequence of elements. The first element is unchanged, because it resides in the CAR of the list, and is not reached via the CDR.

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

You can delete elements from the middle of a list by altering the CDRs of the cons cells in the list. For example, here we delete the second element, `b`, from the list `(a b c)`, by changing the CDR of the first cons cell:

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

Here is the result in box notation:

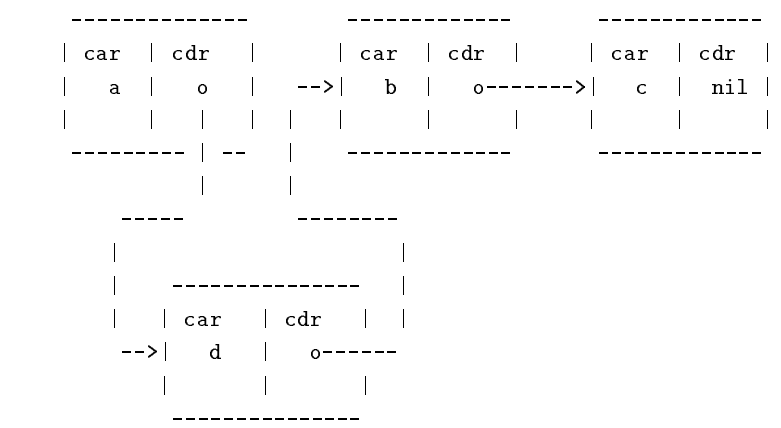


The second cons cell, which previously held the element `b`, still exists and its CAR is still `b`, but it no longer forms part of this list.

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

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

Here is this result in box notation:



5.6.3 Functions that Rearrange Lists

Here are some functions that rearrange lists "destructively" by modifying the CDRs of their component cons cells. We call these functions "destructive" because they chew up the original lists passed to them as arguments, relinking their cons cells to form a new list that is the returned value.

The function `delq` in the following section is another example of destructive list manipulation.

nconc &rest *lists* Function

This function returns a list containing all the elements of *lists*. Unlike `append` (see Section 5.5 [Building Lists], page 80), 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.

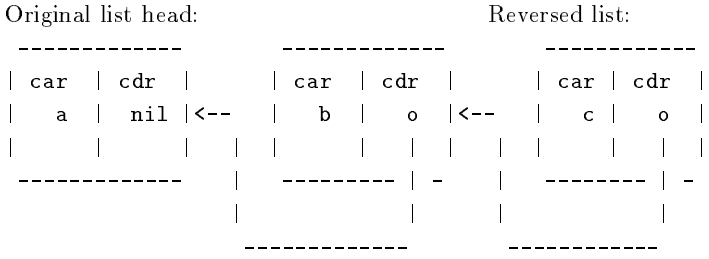
For example:

```
(setq x '(1 2 3 4))
     ⇒ (1 2 3 4)
x
     ⇒ (1 2 3 4)
(nreverse x)
     ⇒ (4 3 2 1)
;; The cons cell that was first is now last.
x
     ⇒ (1)
```

To avoid confusion, we usually store the result of `nreverse` back in the same variable which held the original list:

```
(setq x (nreverse x))
```

Here is the `nreverse` of our favorite example, `(a b c)`, presented graphically:



sort *list predicate* Function

This function sorts *list* stably, though destructively, and returns the sorted list. It compares elements using *predicate*. A stable sort is one in which elements with equal sort keys maintain their relative order before and after the sort. Stability is important when successive sorts are used to order elements according to different criteria.

The argument *predicate* must be a function that accepts two arguments. It is called with two elements of *list*. To get an increasing order sort, the *predicate* should return `t` 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 31.15 [Sorting], page 599, for more functions that perform sorting. See `documentation` in Section 23.2 [Accessing Documentation], page 424, 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). Other useful functions for sets include `memq` and `delq`, and their `equal` versions, `member` and `delete`.

> **Common Lisp note:** Common Lisp has functions `union` (which avoids duplicate elements) and `intersection` for set operations, but GNU Emacs Lisp does not have them. You can write them in Lisp if you wish.

memq *object list* Function

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

```
(memq 'b '(a b c b a))
     ⇒ (b c b a)
(memq '(2) '((1) (2)))    ; (2) and (2) are not eq.
     ⇒ nil
```

delq *object list* Function

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

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 85).

```
(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))
```

The following two functions are like `memq` and `delq` but use `equal` rather than `eq` to compare elements. See Section 2.6 [Equality Predicates], page 39.

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 list* Function

This function destructively removes all elements `equal` to *object* from *list*. It 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 removes the element just as `delq` would. For example:

```
(delete '(2) '((2) (1) (2)))
     ⇒ ((1))
```

Common Lisp note: The functions `member` and `delete` in GNU Emacs Lisp are derived from Maclisp, not Common Lisp. The Common Lisp versions do not use `equal` to compare elements.

See also the function `add-to-list`, in Section 10.8 [Setting Variables], page 156, for another way to add an element to a list stored in a variable.

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*.[1]

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))
```

The associated values in an alist may be any Lisp objects; so may the keys. 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:

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

[1] 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.

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.

Note that 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 7.4 [Property Lists], page 114, for a comparison of property lists and association lists.

assoc *key alist* Function

This function returns the first association for *key* in *alist*. It compares *key* against the alist elements using `equal` (see Section 2.6 [Equality Predicates], page 39). It returns `nil` if no association in *alist* has a CAR `equal` to *key*. For example:

```
(setq trees '((pine . cones) (oak . acorns) (maple . seeds)))
        ((pine . cones) (oak . acorns) (maple . seeds))
(assoc 'oak trees)
        (oak . acorns)
(cdr (assoc 'oak trees))
        acorns
(assoc 'birch trees)
        nil
```

Here is another example, in which the keys and values are not symbols:

```
(setq needles-per-cluster
      '((2 "Austrian Pine" "Red Pine")
        (3 "Pitch Pine")
        (5 "White Pine")))

(cdr (assoc 3 needles-per-cluster))
        ("Pitch Pine")
(cdr (assoc 2 needles-per-cluster))
        ("Austrian Pine" "Red Pine")
```

The functions `assoc-ignore-representation` and `assoc-ignore-case` are much like `assoc` except using `compare-strings` to do the comparison. See Section 4.5 [Text Comparison], page 63.

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 2.6 [Equality Predicates], page 39.

```
(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
```

Note that `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 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 33.3 [Regexp Search], page 656). 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")
```

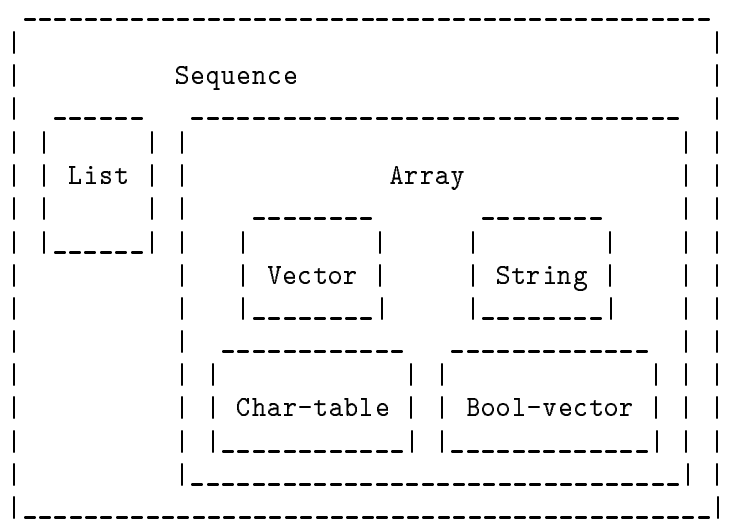
6 Sequences, Arrays, and Vectors

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

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

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

The following diagram shows the relationship between these types:



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

6.1 Sequences

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

sequencep *object* Function

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

length *sequence* Function

This function returns the number of elements in *sequence*. If *sequence* is a cons cell that is not a list (because the final CDR is not `nil`), a `wrong-type-argument` error is signaled.

See Section 5.4 [List Elements], page 78, 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
```

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, then out-of-range values of *index* return `nil`; otherwise, they 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 100) and `nth` (see Section 5.4 [List Elements], page 78).

copy-sequence *sequence* Function

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

Storing a new element into the copy does not affect the original *sequence*, and vice versa. However, the elements of the new sequence are not copies; they are identical (`eq`) to the elements of the original. Therefore, changes

made within these elements, as found via the copied sequence, are also visible in the original sequence.

If the sequence is a string with text properties, the property list in the copy is itself a copy, not shared with the original's property list. However, the actual values of the properties are shared. See Section 31.19 [Text Properties], page 610.

See also `append` in Section 5.5 [Building Lists], page 80, `concat` in Section 4.3 [Creating Strings], page 60, and `vconcat` in Section 6.4 [Vectors], page 102, for others ways to copy sequences.

```
(setq bar '(1 2))
     ⇒ (1 2)
(setq x (vector 'foo bar))
     ⇒ [foo (1 2)]
(setq y (copy-sequence x))
     ⇒ [foo (1 2)]

(eq x y)
     ⇒ nil
(equal x y)
     ⇒ t
(eq (elt x 1) (elt y 1))
     ⇒ t

;; Replacing an element of one sequence.
(aset x 0 'quux)
x ⇒ [quux (1 2)]
y ⇒ [foo (1 2)]

;; Modifying the inside of a shared element.
(setcar (aref x 1) 69)
x ⇒ [quux (69 2)]
y ⇒ [foo (69 2)]
```

6.2 Arrays

An *array* object has slots that hold a number of other Lisp objects, called the elements of the array. Any element of an array may be accessed in constant time. In contrast, an element of a list requires access time that is proportional to the position of the element in the list.

Emacs defines four types of array, all one-dimensional: *strings*, *vectors*, *bool-vectors* and *char-tables*. A vector is a general array; its elements can be any Lisp objects. A string is a specialized array; its elements must be characters (i.e., integers between 0 and 255). Each type of array has its own read syntax. See Section 2.3.8 [String Type], page 27, and Section 2.3.9 [Vector Type], page 29.

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.
- The array is a constant, for evaluation—in other words, it evaluates to itself.
- The elements of an array may be referenced or changed with the functions `aref` and `aset`, respectively (see Section 6.3 [Array Functions], page 100).

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 31.19 [Text Properties], page 610.
- 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 59.

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 20.6.1 [Key Sequence Input], page 344.

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 97.

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. If *array* is a string and *object* is character, but *object* does not use the same number of bytes as the character currently stored in `(aref` *object index*`)`, that is also an error. See Section 32.7 [Splitting Characters], page 633.

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 97.

6.4 Vectors

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

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

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

A vector, like a string or a number, is considered a constant for evaluation: the result of evaluating it is the same vector. This does not evaluate or even examine the elements of the vector. See Section 8.1.1 [Self-Evaluating Forms], page 120.

Here are examples illustrating these principles:

```
(setq avector [1 two '(three) "four" [five]])
     ⇒ [1 two (quote (three)) "four" [five]]
(eval avector)
     ⇒ [1 two (quote (three)) "four" [five]]
(eq avector (eval avector))
     ⇒ t
```

6.5 Functions for Vectors

Here are some functions that relate to vectors:

vectorp *object* Function

This function returns `t` if *object* is a vector.

```
(vectorp [a])
     ⇒ t
(vectorp "asdf")
     ⇒ nil
```

vector &rest *objects* Function

This function creates and returns a vector whose elements are the arguments, *objects*.

```
(vector 'foo 23 [bar baz] "rats")
     ⇒ [foo 23 [bar baz] "rats"]
(vector)
     ⇒ []
```

make-vector *length object* Function

This function returns a new vector consisting of *length* elements, each initialized to *object*.

```
(setq sleepy (make-vector 9 'Z))
     ⇒ [Z Z Z Z Z Z Z Z Z]
```

vconcat &rest *sequences* Function

This function returns a new vector containing all the elements of the *sequences*. The arguments *sequences* may be any kind of arrays, including lists, vectors, or strings. 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 15.6 [Byte-Code Objects], page 229.

The `vconcat` function also allows integers as arguments. It converts them to strings of digits, making up the decimal print representation of the integer, and then uses the strings instead of the original integers. **Don't use this feature; we plan to eliminate it. If you already use this feature, change your programs now!** The proper way to convert an integer to a decimal number in this way is with `format` (see Section 4.7 [Formatting Strings], page 67) or `number-to-string` (see Section 4.6 [String Conversion], page 66).

For other concatenation functions, see `mapconcat` in Section 11.6 [Mapping Functions], page 180, `concat` in Section 4.3 [Creating Strings], page 60, and `append` in Section 5.5 [Building Lists], page 80.

The `append` function provides a way to convert a vector into a list with the same elements (see Section 5.5 [Building Lists], page 80):

```
(setq avector [1 two (quote (three)) "four" [five]])
     ⇒ [1 two (quote (three)) "four" [five]]
(append avector nil)
     ⇒ (1 two (quote (three)) "four" [five])
```

6.6 Char-Tables

A char-table is much like a vector, except that it is indexed by character codes. Any valid character code, without modifiers, can be used as an index in a char-table. You can access a char-table's elements with `aref` and `aset`, as with any array. In addition, a char-table can have *extra slots* to hold additional data not associated with particular character codes. Char-tables are constants when evaluated.

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

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

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

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

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

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

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

set-char-table-default *char-table new-default* Function
This function sets the default value of *char-table* to *new-default*.

There is no special function to access the default value of a char-table. To do that, use `(char-table-range` *char-table* `nil)`.

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

set-char-table-parent *char-table new-parent* Function
This function sets the parent of *char-table* to *new-parent*.

char-table-extra-slot *char-table n* Function
This function returns the contents of extra slot *n* of *char-table*. The number of extra slots in a char-table is determined by its subtype.

set-char-table-extra-slot *char-table n value* Function
This function stores *value* in extra slot *n* of *char-table*.

A char-table can specify an element value for a single character code; it can also specify a value for an entire character set.

char-table-range *char-table range* Function
This returns the value specified in *char-table* for a range of characters *range*. Here are the possibilities for *range*:

`nil`	Refers to the default value.
char	Refers to the element for character *char* (supposing *char* is a valid character code).
charset	Refers to the value specified for the whole character set *charset* (see Section 32.5 [Character Sets], page 632).

generic-char
: A generic character stands for a character set; specifying the generic character as argument is equivalent to specifying the character set name. See Section 32.7 [Splitting Characters], page 633, for a description of generic characters.

set-char-table-range *char-table range value* Function

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

`nil`
: Refers to the default value.

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

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

charset
: Refers to the value specified for the whole character set *charset* (see Section 32.5 [Character Sets], page 632).

generic-char
: A generic character stands for a character set; specifying the generic character as argument is equivalent to specifying the character set name. See Section 32.7 [Splitting Characters], page 633, for a description of generic characters.

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

This function calls *function* for each element of *char-table*. *function* is called with two arguments, a key and a value. The key is a possible *range* argument for `char-table-range`—either a valid character or a generic character—and the value is `(char-table-range` *char-table key*`)`.

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

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

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

6.7 Bool-vectors

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

There are two special functions for working with bool-vectors; aside from that, you manipulate them with same functions used for other kinds of arrays.

make-bool-vector *length initial* Function

Return a new book-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.

7 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 10 [Variables], page 147, and Chapter 11 [Functions], page 171. For the precise read syntax for symbols, see Section 2.3.4 [Symbol Type], page 22.

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.

7.1 Symbol Components

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

Print name
: The *print name cell* holds a string that names the symbol for reading and printing. See `symbol-name` in Section 7.3 [Creating Symbols], page 111.

Value
: The *value cell* holds the current value of the symbol as a variable. When a symbol is used as a form, the value of the form is the contents of the symbol's value cell. See `symbol-value` in Section 10.7 [Accessing Variables], page 155.

Function
: The *function cell* holds the function definition of the symbol. When a symbol is used as a function, its function definition is used in its place. This cell is also used to make a symbol stand for a keymap or a keyboard macro, for editor command execution. Because each symbol has separate value and function cells, variables and function names do not conflict. See `symbol-function` in Section 11.8 [Function Cells], page 183.

Property list
: The *property list cell* holds the property list of the symbol. See `symbol-plist` in Section 7.4 [Property Lists], page 114.

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

The print name cell holds the string that is the name of the symbol. Since symbols are represented textually by their names, it is important not to have two symbols with the same name. The Lisp reader ensures this: every time it reads a symbol, it looks for an existing symbol with the specified name before it creates a new one. (In GNU Emacs Lisp, this lookup uses a hashing algorithm and an obarray; see Section 7.3 [Creating Symbols], page 111.)

In normal usage, the function cell usually contains a function (see Chapter 11 [Functions], page 171) or a macro (see Chapter 12 [Macros], page 189), as that is what the Lisp interpreter expects to see there (see Chapter 8 [Evaluation], page 119). Keyboard macros (see Section 20.14 [Keyboard Macros], page 358), keymaps (see Chapter 21 [Keymaps], page 361) and autoload objects (see Section 8.1.8 [Autoloading], page 125) are also sometimes stored in the function cells of symbols. We often refer to "the function `foo`" when we really mean the function stored in the function cell of the symbol `foo`. We make the distinction only when necessary.

The property list cell normally should hold a correctly formatted property list (see Section 7.4 [Property Lists], page 114), as a number of functions expect to see a property list there.

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

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

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

Because this symbol is the variable which holds the name of the file being visited in the current buffer, the value cell contents we see are the name of the source file of this chapter of the Emacs Lisp Manual. The property list cell contains the list `(variable-documentation 29529)` which tells the documentation functions where to find the documentation string for the variable `buffer-file-name` in the '`DOC-`*version*' file. (29529 is the offset from the beginning of the '`DOC-`*version*' file to where that documentation string begins—see Section 23.1 [Documentation Basics], page 423.) The function cell contains the function for returning the name of the file. `buffer-file-name` names a primitive function, which has no read syntax and prints in hash notation (see Section 2.3.14 [Primitive Function Type], page 31). A symbol naming a function written in Lisp would have a lambda expression (or a byte-code object) in this cell.

7.2 Defining Symbols

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

A definition construct typically specifies a value or meaning for the symbol for one kind of use, plus documentation for its meaning when used in this way. Thus, when you define a symbol as a variable, you can supply an initial value for the variable, plus documentation for the variable.

`defvar` and `defconst` are special forms that define a symbol as a global variable. They are documented in detail in Section 10.5 [Defining Variables], page 152. For defining user option variables that can be customized, use `defcustom` (see Chapter 13 [Customization], page 199).

`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 11 [Functions], page 171.

`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 12 [Macros], page 189.

In Emacs Lisp, a definition is not required in order to use a symbol as a variable or function. Thus, you can make a symbol a global variable with `setq`, whether you define it first or not. The real purpose of definitions is to guide programmers and programming tools. They inform programmers who read the code that certain symbols are *intended* to be used as variables, or as functions. In addition, utilities such as '`etags`' and '`make-docfile`' recognize definitions, and add appropriate information to tag tables and the '`DOC-`*version*' file. See Section 23.2 [Accessing Documentation], page 424.

7.3 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.

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.

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.

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, and you can create an obarray with `(make-vector` *length* `0)`. **This is the only valid way to create an obarray.** Prime numbers as lengths tend to result in good hashing; lengths one less than a power of two are also good.

Do not try to put symbols in an obarray yourself. This does not work—only `intern` can enter a symbol in an obarray properly.

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

Most of the functions below take a name and sometimes an obarray as arguments. A `wrong-type-argument` error is signaled if the name is not a string, or if the obarray is not a vector.

symbol-name *symbol* Function

This function returns the string that is *symbol*'s name. For example:

```
(symbol-name 'foo)
     ⇒ "foo"
```

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

make-symbol *name* Function

This function returns a newly-allocated, uninterned symbol whose name is *name* (which must be a string). Its value and function definition are void, and its property list is `nil`. In the example below, the value of `sym`

is not `eq` to `foo` because it is a distinct uninterned symbol whose name is also '`foo`'.

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

intern *name* &optional *obarray* Function

This function returns the interned symbol whose name is *name*. If there is no such symbol in the obarray *obarray*, `intern` creates a new one, adds it to the obarray, and returns it. If *obarray* is omitted, the value of the global variable `obarray` is used.

```
(setq sym (intern "foo"))
     ⇒ foo
(eq sym 'foo)
     ⇒ t

(setq sym1 (intern "foo" other-obarray))
     ⇒ foo
(eq sym '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.

```
(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 23.2 [Accessing Documentation], page 424, for another example using `mapatoms`.

unintern *symbol* &optional *obarray* Function

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

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

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

7.4 Property Lists

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

Character positions in a string or buffer can also have property lists. See Section 31.19 [Text Properties], page 610.

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.

7.4.1 Property Lists and Association Lists

Association lists (see Section 5.8 [Association Lists], page 91) are very similar to property lists. In contrast to association lists, the order of the pairs in the property list is not significant since the property names must be distinct.

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

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

7.4.2 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.

```
(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 35 [Abbrevs], page 683).

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))
```

7.4.3 Property Lists Outside Symbols

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

plist-get *plist property* Function

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

```
(plist-get '(foo 4) 'foo)
     ⇒ 4
```

plist-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)))
```

8 Evaluation

The *evaluation* of expressions in Emacs Lisp is performed by the *Lisp interpreter*—a program that receives a Lisp object as input and computes its *value as an expression*. How it does this depends on the data type of the object, according to rules described in this chapter. The interpreter runs automatically to evaluate portions of your program, but can also be called explicitly via the Lisp primitive function `eval`.

A Lisp object that is intended for evaluation is called an *expression* or a *form*. The fact that expressions are data objects and not merely text is one of the fundamental differences between Lisp-like languages and typical programming languages. Any object can be evaluated, but in practice only numbers, symbols, lists and strings are evaluated very often.

It is very common to read a Lisp expression and then evaluate the expression, but reading and evaluation are separate activities, and either can be performed alone. Reading per se does not evaluate anything; it converts the printed representation of a Lisp object to the object itself. It is up to the caller of `read` whether this object is a form to be evaluated, or serves some entirely different purpose. See Section 18.3 [Input Functions], page 286.

Do not confuse evaluation with command key interpretation. The editor command loop translates keyboard input into a command (an interactively callable function) using the active keymaps, and then uses `call-interactively` to invoke the command. The execution of the command itself involves evaluation if the command is written in Lisp, but that is not a part of command key interpretation itself. See Chapter 20 [Command Loop], page 319.

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

Evaluation of a function call ultimately calls the function specified in it. See Chapter 11 [Functions], page 171. The execution of the function may itself work by evaluating the function definition; or the function may be a Lisp primitive implemented in C, or it may be a byte-compiled function (see Chapter 15 [Byte Compilation], page 223).

The evaluation of forms takes place in a context called the *environment*, which consists of the current values and bindings of all Lisp variables.[1] Whenever a form refers to a variable without creating a new binding for it, the value of the variable's binding in the current environment is used. See Chapter 10 [Variables], page 147.

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

Evaluation of a form may create new environments for recursive evaluation by binding variables (see Section 10.3 [Local Variables], page 148). These environments are temporary and vanish by the time evaluation of the form is complete. The form may also make changes that persist; these changes are called *side effects*. An example of a form that produces side effects is `(setq foo 1)`.

The details of what evaluation means for each kind of form are described below (see Section 8.1 [Forms], page 120).

8.1 Kinds of Forms

A Lisp object that is intended to be evaluated is called a *form*. How Emacs evaluates a form depends on its data type. Emacs has three different kinds of form that are evaluated differently: symbols, lists, and "all other types". This section describes all three kinds, one by one, starting with the "all other types" which are self-evaluating forms.

8.1.1 Self-Evaluating Forms

A *self-evaluating form* is any form that is not a list or symbol. Self-evaluating forms evaluate to themselves: the result of evaluation is the same object that was evaluated. Thus, the number 25 evaluates to 25, and the string `"foo"` evaluates to the string `"foo"`. Likewise, evaluation of a vector does not cause evaluation of the elements of the vector—it returns the same vector with its contents unchanged.

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

It is common to write numbers, characters, strings, and even vectors in Lisp code, taking advantage of the fact that they self-evaluate. However, it is quite unusual to do this for types that lack a read syntax, because there's no way to write them textually. It is possible to construct Lisp expressions containing these types by means of a Lisp program. Here is an example:

```
;; Build an expression containing a buffer object.
(setq print-exp (list 'print (current-buffer)))
     ⇒ (print #<buffer eval.texi>)
```

```
;; Evaluate it.
(eval print-exp)
     ⊣ #<buffer eval.texi>
     ⇒ #<buffer eval.texi>
```

8.1.2 Symbol Forms

When a symbol is evaluated, it is treated as a variable. The result is the variable's value, if it has one. If it has none (if its value cell is void), an error is signaled. For more information on the use of variables, see Chapter 10 [Variables], page 147.

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 10.2 [Constant Variables], page 148.

8.1.3 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.

8.1.4 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 11.3 [Function Names], page 176, for more information about using a symbol as a name for a function stored in the function cell of the symbol.

One possible consequence of this process is an infinite loop, in the event that a symbol's function cell refers to the same symbol. Or a symbol may have a void function cell, in which case the subroutine `symbol-function` signals a `void-function` error. But if neither of these things happens, we eventually obtain a non-symbol, which ought to be a function or other suitable object.

More precisely, we should now have a Lisp function (a lambda expression), a byte-code function, a primitive function, a Lisp macro, a special form, or an autoload object. Each of these types is a case described in one of the following sections. If the object is not one of these types, the error `invalid-function` is signaled.

The following example illustrates the symbol indirection process. We use `fset` to set the function cell of a symbol and `symbol-function` to get the function cell contents (see Section 11.8 [Function Cells], page 183). Specifically, we store the symbol `car` into the function cell of `first`, and the symbol `first` into the function cell of `erste`.

```
;; Build this function cell linkage:
;;   -------------       -----        -------        -------
;;  | #<subr car> | <-- | car |  <-- | first |  <-- | erste |
;;   -------------       -----        -------        -------

(symbol-function 'car)
       #<subr car>
(fset 'first 'car)
       car
(fset 'erste 'first)
       first
(erste '(1 2 3))    ; Call the function referenced by erste.
       1
```

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

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

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

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

indirect-function *function* 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.

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

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

8.1.5 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 11.5 [Calling Functions], page 179). If the function is written in Lisp, the arguments are used to bind the argument variables of the function (see Section 11.2 [Lambda Expressions], page 172); 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.

8.1.6 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 12 [Macros], page 189, for a complete description of Emacs Lisp macros.

8.1.7 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 9.3 [Combining Conditions], page 132

`catch` see Section 9.5.1 [Catch and Throw], page 135

`cond` see Section 9.2 [Conditionals], page 130

`condition-case`
see Section 9.5.3.3 [Handling Errors], page 140

`defconst` see Section 10.5 [Defining Variables], page 152

`defmacro` see Section 12.4 [Defining Macros], page 191

`defun` see Section 11.4 [Defining Functions], page 177

`defvar` see Section 10.5 [Defining Variables], page 152

`function` see Section 11.7 [Anonymous Functions], page 182

`if` see Section 9.2 [Conditionals], page 130

`interactive`
see Section 20.3 [Interactive Call], page 325

`let`
`let*` see Section 10.3 [Local Variables], page 148

`or` see Section 9.3 [Combining Conditions], page 132

`prog1`
`prog2`
`progn` see Section 9.1 [Sequencing], page 129

`quote` see Section 8.2 [Quoting], page 125

`save-current-buffer`
see Section 26.2 [Current Buffer], page 479

`save-excursion`
see Section 29.3 [Excursions], page 560

`save-restriction`
see Section 29.4 [Narrowing], page 561

`save-window-excursion`
see Section 27.16 [Window Configurations], page 521

`setq` see Section 10.8 [Setting Variables], page 156

`setq-default`
see Section 10.10.2 [Creating Buffer-Local], page 163

`track-mouse`
see Section 28.13 [Mouse Tracking], page 541

`unwind-protect`
see Section 9.5 [Nonlocal Exits], page 135

`while` see Section 9.4 [Iteration], page 134

`with-output-to-temp-buffer`
see Section 38.7 [Temporary Displays], page 746

> **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).

8.1.8 Autoloading

The *autoload* feature allows you to call a function or macro whose function definition has not yet been loaded into Emacs. It specifies which file contains the definition. When an autoload object appears as a symbol's function definition, calling that symbol as a function automatically loads the specified file; then it calls the real definition loaded from that file. See Section 14.4 [Autoload], page 215.

8.2 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 11.7 [Anonymous Functions], page 182), which causes an anonymous lambda expression written in Lisp to be compiled, and ‘‘’ (see Section 12.5 [Backquote], page 192), which is used to quote only part of a list, while computing and substituting other parts.

8.3 Eval

Most often, forms are evaluated automatically, by virtue of their occurrence in a program being run. On rare occasions, you may need to write code that evaluates a form that is computed at run time, such as after reading a form from text being edited or getting one from a property list. On these occasions, use the `eval` function.

The functions and variables described in this section evaluate forms, specify limits to the evaluation process, or record recently returned values. Loading a file also does evaluation (see Chapter 14 [Loading], page 211).

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

eval *form* Function

This is the basic function evaluating an expression. It evaluates *form* in the current environment and returns the result. How the evaluation proceeds depends on the type of the object (see Section 8.1 [Forms], page 120).

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.

If *stream* is non-`nil`, the values that result from evaluating the expressions in the region are printed using *stream*. See Section 18.4 [Output Streams], page 287.

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 Section 14.1 [How Programs Do Loading], page 211) to specify this function, but it is more robust to use the *read-function* argument.

`eval-region` always returns `nil`.

eval-current-buffer &optional *stream* Command

This is like `eval-region` except that it operates on the whole buffer.

max-lisp-eval-depth Variable

This variable defines the maximum depth allowed in calls to `eval`, `apply`, and `funcall` before an error is signaled (with error message "`Lisp`

`nesting exceeds max-lisp-eval-depth`"). This limit, with the associated error when it is exceeded, is one way that Lisp avoids infinite recursion on an ill-defined function.

The depth limit counts internal uses of `eval`, `apply`, and `funcall`, such as for calling the functions mentioned in Lisp expressions, and recursive evaluation of function call arguments and function body forms, as well as explicit calls in Lisp code.

The default value of this variable is 300. If you set it to a value less than 100, Lisp will reset it to 100 if the given value is reached. Entry to the Lisp debugger increases the value, if there is little room left, to make sure the debugger itself has room to execute.

`max-specpdl-size` provides another limit on nesting. See Section 10.3 [Local Variables], page 148.

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. 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
```

9 Control Structures

A Lisp program consists of expressions or *forms* (see Section 8.1 [Forms], page 120). We control the order of execution of the 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*, and the function's value is the value of *b*.

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 12 [Macros], page 189).

9.1 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`.

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 often as it used to be. It is needed now most often inside an `unwind-protect`, `and`, `or`, or in the *then*-part of an `if`.

progn *forms...* Special Form

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

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

Two other control constructs likewise evaluate a series of forms but return a different value:

prog1 *form1 forms...* Special Form

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

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

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

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

prog2 *form1 form2 forms...* Special Form

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

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

9.2 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 9.1 [Sequencing], page 129.)

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

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

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

when *condition then-forms...* Macro

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

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

is entirely equivalent to

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

unless *condition forms...* Macro

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

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

is entirely equivalent to

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

cond *clause...* Special Form

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

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

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

If the value of *condition* is `nil`, the clause "fails", so the `cond` moves on to the following clause, trying its *condition*.

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

A clause may also look like this:

```
(condition)
```

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

The following example has four clauses, which test for the cases where the value of `x` is a number, string, buffer and symbol, respectively:

```
(cond ((numberp x) x)
      ((stringp x) x)
      ((bufferp x)
       (setq temporary-hack x) ; multiple body-forms
       (buffer-name x))        ; in one clause
      ((symbolp x) (symbol-value x)))
```

Often we want to execute the last clause whenever none of the previous clauses was successful. To do this, we use `t` as the *condition* of the last clause, like this: `(t` *body-forms*`)`. The form `t` evaluates to `t`, which is never `nil`, so this clause never fails, provided the `cond` gets to it at all.

For example,

```
(cond ((eq a 'hack) 'foo)
      (t "default"))
⇒ "default"
```

This expression is a `cond` which 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))
```

9.3 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 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.

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` can be expressed in terms of either `if` or `cond`. For example:

```
(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`.

For example, this expression tests whether `x` is either 0 or `nil`:

```
(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.

9.4 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 9.5 [Nonlocal Exits], page 135).

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
```

If you would like to execute something on each iteration before the end-test, put it together with 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).

9.5 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.

9.5.1 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 10.3 [Local Variables], page 148).

Likewise, `throw` restores the buffer and position saved by `save-excursion` (see Section 29.3 [Excursions], page 560), and the narrowing status saved by `save-restriction` and the window selection saved by `save-window-excursion` (see Section 27.16 [Window Configurations], page 521). It also runs any cleanups established with the `unwind-protect` special form when it exits that form (see Section 9.5.4 [Cleanups], page 144).

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 20.11 [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 done within *body* specifying the same value *tag*, the `catch` 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*).

9.5.2 Examples of `catch` and `throw`

One way to use `catch` and `throw` is to exit from a doubly nested loop. (In most languages, this would be done with a "go to".) Here we compute (`foo` *i j*) for *i* and *j* varying from 0 to 9:

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

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

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

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

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

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

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

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

9.5.3 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 9.5.4 [Cleanups], page 144.) 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 9.5.1 [Catch and Throw], page 135.

9.5.3.1 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 20.9 [Quitting], page 351.

error *format-string* &rest *args* Function

This function signals an error with an error message constructed by applying `format` (see Section 4.6 [String Conversion], page 66) to *format-string* and *args*.

These examples show typical uses of `error`:

```
(error "That is an error -- try something else")
     error  That is an error -- try something else

(error "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.

The number and significance of the objects in *data* depends on *error-symbol*. For example, with a `wrong-type-arg` error, there should be two objects in the list: a predicate that describes the type that was expected, and the object that failed to fit that type. See Section 9.5.3.4 [Error Symbols], page 143, for a description of error symbols.

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 9.5.3.3 [Handling Errors], page 140). If the error is not handled, these two values are used in printing the error message.

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

```
(signal 'wrong-number-of-arguments '(x y))
     error  Wrong number of arguments: x, y

(signal 'no-such-error '("My unknown error condition"))
     error  peculiar error: "My unknown error condition"
```

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

9.5.3.2 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, the current command is terminated and control returns to the editor command loop, because 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.

An error that has no explicit handler may call the Lisp debugger. The debugger is enabled if the variable `debug-on-error` (see Section 17.1.1 [Error Debugging], page 247) 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.

9.5.3.3 Writing Code to Handle Errors

The usual effect of signaling an error is to terminate the command that is running and return immediately to the Emacs editor command loop. You can arrange to trap errors occurring in a part of your program by establishing an error handler, with the special form `condition-case`. A simple example looks like this:

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

This deletes the file named *filename*, catching any error and returning `nil` if an error occurs.

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

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

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

If an error is handled by some `condition-case` form, this ordinarily prevents the debugger from being run, even if `debug-on-error` says this error should invoke the debugger. See Section 17.1.1 [Error Debugging], page 247. If you want to be able to debug errors that are caught by a `condition-case`, set the variable `debug-on-signal` to a non-`nil` value.

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

that are exited. Once control arrives at the handler, the body of the handler is executed.

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

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

Error signaling and handling have some resemblance to `throw` and `catch`, but they are entirely separate facilities. An error cannot be caught by a `catch`, and a `throw` cannot be handled by an error handler (though using `throw` when there is no suitable `catch` signals an error that can be handled).

condition-case *var protected-form handlers...* Special Form

This special form establishes the error handlers *handlers* around the execution of *protected-form*. If *protected-form* executes without error, the value it returns becomes the value of the `condition-case` form; in this case, the `condition-case` has no effect. The `condition-case` form makes a difference when an error occurs during *protected-form*.

Each of the *handlers* is a list of the form (*conditions body*...). Here *conditions* is an error condition name to be handled, or a list of condition names; *body* is one or more Lisp expressions to be executed when this handler handles an error. Here are examples of handlers:

```
(error nil)

(arith-error (message "Division by zero"))

((arith-error file-error)
 (message
  "Either division by zero or failure to open a file"))
```

Each error that occurs has an *error symbol* that describes what kind of error it is. The `error-conditions` property of this symbol is a list of condition names (see Section 9.5.3.4 [Error Symbols], page 143). Emacs searches all the active `condition-case` forms for a handler that specifies one or more of these condition names; the innermost matching `condition-case` handles the error. Within this `condition-case`, the first applicable handler handles the error.

After executing the body of the handler, the `condition-case` returns normally, using the value of the last form in the handler body as the overall value.

The argument *var* is a variable. `condition-case` does not bind this variable when executing the *protected-form*, only when it handles an error. At that time, it binds *var* locally to an *error description*, which is a list giving the particulars of the error. The error description has the form (*error-symbol* . *data*). The handler can refer to this list to decide what to do. For example, if the error is for failure opening a file, the file name is the second element of *data*—the third element of the error description.

If *var* is `nil`, that means no variable is bound. Then the error symbol and associated data are not available to the handler.

error-message-string *error-description* Function

This function returns the error message string for a given error descriptor. It is useful if you want to handle an error by printing the usual error message for that error.

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

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

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

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

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

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

```
(setq baz 34)
       34
```

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

9.5.3.4 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. 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.

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-message` property exists, but is not a string, the error message '`peculiar error`' is used.

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 Section 9.5.3.1 [Signaling Errors], page 138) 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 C [Standard Errors], page 809, for a list of all the standard error symbols and their conditions.

9.5.4 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.

unwind-protect *body cleanup-forms...* Special Form

`unwind-protect` executes the *body* with a guarantee that the *cleanup-forms* will be evaluated if control leaves *body*, no matter how that happens. The *body* 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 the *body* forms finish normally, `unwind-protect` returns the value of the last *body* form, after it evaluates the *cleanup-forms*. If the *body* forms do not finish, `unwind-protect` does not return any value in the normal sense.

Only the *body* is actually protected by the `unwind-protect`. If any of the *cleanup-forms* themselves exits nonlocally (e.g., 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 Section 10.3 [Local Variables], page 148).

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

```
(save-excursion
  (let ((buffer (get-buffer-create " *temp*")))
    (set-buffer buffer)
    (unwind-protect
        body
      (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* happens to get an error after switching to a different buffer! (Alternatively, you could write another `save-excursion` around the body, 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 Section 26.2 [Current Buffer], page 479). Several of the macros defined in this manual use `unwind-protect` in this way.

Here is an actual example taken from the file '`ftp.el`'. It creates a process (see Chapter 36 [Processes], page 689) 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 actually 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.

10 Variables

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

In Lisp, unlike most programming languages, programs are represented primarily as Lisp objects and only secondarily as text. The Lisp objects used for variables are symbols: the symbol name is the variable name, and the variable's value is stored in the value cell of the symbol. The use of a symbol as a variable is independent of its use as a function name. See Section 7.1 [Symbol Components], page 109.

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

10.1 Global Variables

The simplest way to use a variable is *globally*. This means that the variable has just one value at a time, and this value is in effect (at least for the moment) throughout the Lisp system. The value remains in effect until you specify a new one. When a new value replaces the old one, no trace of the old value remains in the variable.

You specify a value for a symbol with `setq`. For example,

```
(setq x '(a b))
```

gives the variable `x` the value `(a b)`. Note that `setq` does not evaluate its first argument, the name of the variable, but it does evaluate the second argument, the new value.

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

```
x ⇒ (a b)
```

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

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

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

10.2 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 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 symbol whose name starts with `:`, except that you are allowed to set such a symbol to itself.

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

keyword-symbols-constant-flag Variable

If this variable is `nil`, you are allowed to set and bind symbols whose names start with `:` as you wish. This is to make it possible to run old Lisp programs which do that.

10.3 Local Variables

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

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

Establishing a local value saves away the previous value (or lack of one) of the variable. When the life span of the local value is over, the previous value is restored. In the mean time, we say that the previous value is *shadowed* and *not visible*. Both global and local values may be shadowed (see Section 10.9.1 [Scope], page 159).

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

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

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

A variable can have more than one local binding at a time (for example, if there are nested `let` forms that bind it). In such a case, the most recently created local binding that still exists is the *current binding* of the variable. (This rule is called *dynamic scoping*; see Section 10.9 [Variable Scoping], page 158.) If there are no local bindings, the variable's global binding is its current binding. We sometimes call the current binding the *most-local existing binding*, for emphasis. Ordinary evaluation of a symbol always returns the value of its current binding.

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

let (*bindings...*) *forms...* Special Form

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

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

All of the *value-forms* in *bindings* are evaluated in the order they appear and *before* binding any of the symbols to them. Here is an example of this: `Z` is bound to the old value of `Y`, which is 2, not the new value of `Y`, which is 1.

```
(setq Y 2)
     ⇒ 2
(let ((Y 1)
      (Z Y))
  (list Y Z))
     ⇒ (1 2)
```

let* (*bindings...*) *forms...* Special Form

This special form is like `let`, but it binds each variable right after computing its local value, before computing the local value for the next variable. Therefore, an expression in *bindings* can reasonably refer to the preceding symbols bound in this `let*` form. Compare the following example with the example above for `let`.

```
(setq Y 2)
     ⇒ 2
(let* ((Y 1)
       (Z Y))     ; Use the just-established value of Y.
  (list Y Z))
     ⇒ (1 1)
```

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

- Function calls (see Chapter 11 [Functions], page 171).

- Macro calls (see Chapter 12 [Macros], page 189).
- `condition-case` (see Section 9.5.3 [Errors], page 138).

Variables can also have buffer-local bindings (see Section 10.10 [Buffer-Local Variables], page 161) and frame-local bindings (see Section 10.11 [Frame-Local Variables], page 168); a few variables have terminal-local bindings (see Section 28.2 [Multiple Displays], page 526). These kinds of bindings work somewhat like ordinary local bindings, but they are localized depending on "where" you are in Emacs, rather than localized in time.

max-specpdl-size Variable

This variable defines the limit on the total number of local variable bindings and `unwind-protect` cleanups (see Section 9.5 [Nonlocal Exits], page 135) that are allowed before signaling an error (with data `"Variable binding depth exceeds max-specpdl-size"`).

This limit, with the associated error when it is exceeded, is one way that Lisp avoids infinite recursion on an ill-defined function. `max-lisp-eval-depth` provides another limit on depth of nesting. See Section 8.3 [Eval], page 126.

The default value is 600. Entry to the Lisp debugger increases the value, if there is little room left, to make sure the debugger itself has room to execute.

10.4 When a Variable is "Void"

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

Note that a value of `nil` is not the same as void. The symbol `nil` is a Lisp object and can be the value of a variable just as any other object can be; but it is *a value*. A void variable does not have any value.

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

makunbound *symbol* Function

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

`makunbound` returns *symbol*.

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

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

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

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

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

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

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

boundp *variable* Function

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

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

```
(boundp 'abracadabra)
        t
```

10.5 Defining Global Variables

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

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

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

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

This special form defines *symbol* as a variable and can also initialize and document it. The definition informs a person reading your code that *symbol* is used as a variable that might be set or changed. Note that *symbol* is not evaluated; the symbol to be defined must appear explicitly in the `defvar`.

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

If *symbol* has a buffer-local binding in the current buffer, `defvar` operates on the default value, which is buffer-independent, not the current (buffer-local) binding. It sets the default value if the default value is void. See Section 10.10 [Buffer-Local Variables], page 161.

When you evaluate a top-level `defvar` form with *C-M-x* in Emacs Lisp mode (`eval-defun`), a special feature of `eval-defun` arranges to set the variable unconditionally, without testing whether its value is void.

If the *doc-string* argument appears, it specifies the documentation for the variable. (This opportunity to specify documentation is one of the main benefits of defining the variable.) The documentation is stored in the symbol's `variable-documentation` property. The Emacs help functions (see Chapter 23 [Documentation], page 423) look for this property.

If the first character of *doc-string* is '`*`', it means that this variable is considered a user option. This lets users set the variable conveniently using

the commands `set-variable` and `edit-options`. However, it is better to use `defcustom` instead of `defvar` for user option variables, so you can specify customization information. See Chapter 13 [Customization], page 199.

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

```
(defvar foo)
     ⇒ foo
```

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

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

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

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

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

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

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

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

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

`defconst` always evaluates *value*, and sets the value of *symbol* to the result if *value* is given. If *symbol* does have a buffer-local binding in the current buffer, `defconst` sets the default value, not the buffer-local value.

(But you should not be making buffer-local bindings for a symbol that is defined with `defconst`.)

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

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

user-variable-p *variable* Function

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

User option variables are distinguished from other variables by the first character of the `variable-documentation` property. If the property exists and is a string, and its first character is '`*`', then the variable is a user option.

If a user option variable has a `variable-interactive` property, the `set-variable` command uses that value to control reading the new value for the variable. The property's value is used as if it were to `interactive` (see Section 20.2.1 [Using Interactive], page 320). However, this feature is largely obsoleted by `defcustom` (see Chapter 13 [Customization], page 199).

Warning: If the `defconst` and `defvar` special forms are used while the variable has a local binding, they set the local binding's value; the global binding is not changed. This is not what we really want. To prevent it, use these special forms at top level in a file, where normally no local binding is in effect, and make sure to load the file before making a local binding for the variable.

10.6 Tips for Defining Variables Robustly

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

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

But be careful not to write the code like this:

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

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

10.7 Accessing Variable Values

The usual way to reference a variable is to write the symbol which names it (see Section 8.1.2 [Symbol Forms], page 121). This requires you to specify the variable name when you write the program. Usually that is exactly what you want to do. Occasionally you need to choose at run time which variable to reference; then you can use `symbol-value`.

symbol-value *symbol* Function

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

```
(setq abracadabra 5)
     ⇒ 5
(setq foo 9)
     ⇒ 9

;; Here the symbol abracadabra
;;    is the symbol whose value is examined.
(let ((abracadabra 'foo))
  (symbol-value 'abracadabra))
     ⇒ foo

;; Here the value of abracadabra,
;;    which is foo,
;;    is the symbol whose value is examined.
(let ((abracadabra 'foo))
  (symbol-value abracadabra))
     ⇒ 9

(symbol-value 'abracadabra)
     ⇒ 5
```

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

10.8 How to Alter a Variable Value

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

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

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

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

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

```
(setq x (1+ 2))
     ⇒ 3
x                    ; x now has a global value.
```

```
     ⇒ 3
(let ((x 5))
  (setq x 6)          ; The local binding of x is set.
  x)
     ⇒ 6
x                     ; The global value is unchanged.
     ⇒ 3
```

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

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

set *symbol value* Function

This function sets *symbol*'s value to *value*, then returns *value*. Since `set` is a function, the expression written for *symbol* is evaluated to obtain the symbol to set.

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

```
(set one 1)
error  Symbol's value as variable is void: one
(set 'one 1)
     ⇒ 1
(set 'two 'one)
     ⇒ one
(set two 2)           ; two evaluates to symbol one.
     ⇒ 2
one                   ; So it is one that was set.
     ⇒ 2
(let ((one 1))        ; This binding of one is set,
  (set 'one 3)        ;    not the global value.
  one)
     ⇒ 3
one
     ⇒ 2
```

If *symbol* is not actually a symbol, a `wrong-type-argument` error is signaled.

```
(set '(x y) 'z)
error  Wrong type argument: symbolp, (x y)
```

Logically speaking, `set` is a more fundamental primitive than `setq`. Any use of `setq` can be trivially rewritten to use `set`; `setq` could even be defined as a macro, given the availability of `set`. However, `set` itself is rarely used; beginners hardly need to know about it. It is useful only for

choosing at run time which variable to set. For example, the command `set-variable`, which reads a variable name from the user and then sets the variable, needs to use `set`.

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

One other function for setting a variable is designed to add an element to a list if it is not already present in the list.

add-to-list *symbol element* 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.

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)))
```

10.9 Scoping Rules for Variable Bindings

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

Local bindings in Emacs Lisp have *indefinite scope* and *dynamic extent*. *Scope* refers to *where* textually in the source code the binding can be accessed. Indefinite scope means that any part of the program can potentially

access the variable binding. *Extent* refers to *when*, as the program is executing, the binding exists. Dynamic extent means that the binding lasts as long as the activation of the construct that established it.

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

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

10.9.1 Scope

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

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

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

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

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

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

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

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

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

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

10.9.2 Extent

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

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

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

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

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

10.9.3 Implementation of Dynamic Scoping

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

Suppose there is a stack of bindings, which are variable-value pairs. At entry to a function or to a `let` form, we can push bindings onto the stack for the arguments or local variables created there. We can pop those bindings from the stack at exit from the binding construct.

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

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

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

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

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

10.9.4 Proper Use of Dynamic Scoping

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

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

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

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

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

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

10.10 Buffer-Local Variables

Global and local variable bindings are found in most programming languages in one form or another. Emacs also supports additional, unusual kinds of variable binding: *buffer-local* bindings, which apply only in one buffer, and frame-local bindings, which apply only in one frame. Having different values for a variable in different buffers and/or frames is an important customization method.

This section describes buffer-local bindings; for frame-local bindings, see the following section, Section 10.11 [Frame-Local Variables], page 168. (A few variables have bindings that are local to each terminal; see Section 28.2 [Multiple Displays], page 526.)

10.10.1 Introduction to Buffer-Local Variables

A buffer-local variable has a buffer-local binding associated with a particular buffer. The binding is in effect when that buffer is current; otherwise, it is not in effect. If you set the variable while a buffer-local binding is in effect, the new value goes in that binding, so its other bindings are unchanged. This means that the change is visible only in the buffer where you made it.

The variable's ordinary binding, which is not associated with any specific buffer, is called the *default binding*. In most cases, this is the global binding.

A variable can have buffer-local bindings in some buffers but not in other buffers. The default binding is shared by all the buffers that don't have their own bindings for the variable. (This includes all newly created buffers.) If you set the variable in a buffer that does not have a buffer-local binding for it, this sets the default binding (assuming there are no frame-local bindings to complicate the matter), so the new value is visible in all the buffers that see the default binding.

The most common use of buffer-local bindings is for major modes to change variables that control the behavior of commands. For example, C mode and Lisp mode both set the variable `paragraph-start` to specify that only blank lines separate paragraphs. They do this by making the variable buffer-local in the buffer that is being put into C mode or Lisp mode, and then setting it to the new value for that mode. See Section 22.1 [Major Modes], page 391.

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 values in one or more buffers, you can get Emacs very confused by binding the variable with `let`, changing

to a different current buffer in which a different binding is in effect, and then exiting the `let`. This can scramble the values of the buffer-local and default bindings.

To preserve your sanity, avoid using a variable in that way. If you use `save-excursion` around each piece of code that changes to a different current buffer, you will not have this problem (see Section 29.3 [Excursions], page 560). Here is an example of what to avoid:

```
(setq foo 'b)
(set-buffer "a")
(make-local-variable 'foo)
(setq foo 'a)
(let ((foo 'temp))
  (set-buffer "b")
  body...)
foo ⇒ 'a       ; The old buffer-local value from buffer 'a'
               ;    is now the default value.
(set-buffer "a")
foo ⇒ 'temp    ; The local let value that should be gone
               ;    is now the buffer-local value in buffer 'a'.
```

But `save-excursion` as shown here avoids the problem:

```
(let ((foo 'temp))
  (save-excursion
    (set-buffer "b")
    body...))
```

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*.

10.10.2 Creating and Deleting Buffer-Local Bindings

make-local-variable *variable* Command

This function creates a buffer-local binding in the current buffer for *variable* (a symbol). Other buffers are not affected. The value returned is *variable*.

The buffer-local value of *variable* starts out as the same value *variable* previously had. If *variable* was void, it remains void.

```
;; In buffer 'b1':
(setq foo 5)                ; Affects all buffers.
     ⇒ 5
(make-local-variable 'foo)  ; Now it is local in 'b1'.
     ⇒ foo
```

```
foo                             ; That did not change
     ⇒ 5                        ;    the value.
(setq foo 6)                    ; Change the value
     ⇒ 6                        ;    in 'b1'.
foo
     ⇒ 6

;; In buffer 'b2', the value hasn't changed.
(save-excursion
  (set-buffer "b2")
  foo)
     ⇒ 5
```

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

If the variable is terminal-local, this function signals an error. Such variables cannot have buffer-local bindings as well. See Section 28.2 [Multiple Displays], page 526.

Note: do not use `make-local-variable` for a hook variable. Instead, use `make-local-hook`. See Section 22.6 [Hooks], page 420.

make-variable-buffer-local *variable* Command

This function marks *variable* (a symbol) automatically buffer-local, so that any subsequent attempt to set it will make it local to the current buffer at the time.

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

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`.

buffer-local-variables &optional *buffer* Function

This function returns a list describing the buffer-local variables in buffer *buffer*. (If *buffer* is omitted, the current buffer is used.) It returns an association list (see Section 5.8 [Association Lists], page 91) in which each element contains one buffer-local variable and its value. However, when a variable's buffer-local binding in *buffer* is void, then the variable appears directly in the resulting list.

```
(make-local-variable 'foobar)
(makunbound 'foobar)
(make-local-variable 'bind-me)
(setq bind-me 69)
(setq lcl (buffer-local-variables))
    ;; First, built-in variables local in all buffers:
⇒ ((mark-active . nil)
    (buffer-undo-list . nil)
    (mode-name . "Fundamental")
    ...
    ;; Next, non-built-in buffer-local variables.
    ;; This one is buffer-local and void:
    foobar
    ;; This one is buffer-local and nonvoid:
    (bind-me . 69))
```

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

kill-local-variable *variable* Command

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

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

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

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

kill-all-local-variables Function

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

This function also resets certain other information pertaining to the buffer: it sets the local keymap to `nil`, the syntax table to the value of `(standard-syntax-table)`, the case table to `(standard-case-table)`, and the abbrev table to the value of `fundamental-mode-abbrev-table`.

The very first thing this function does is run the normal hook `change-major-mode-hook` (see below).

Every major mode command begins by calling this function, which has the effect of switching to Fundamental mode and erasing most of the effects of the previous major mode. To ensure that this does its job, the variables that major modes set should not be marked permanent.

`kill-all-local-variables` returns `nil`.

change-major-mode-hook Variable

The function `kill-all-local-variables` runs this normal hook before it does anything else. This gives major modes a way to arrange for something special to be done if the user switches to a different major mode. 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 22.6 [Hooks], page 420.

A buffer-local variable is *permanent* if the variable name (a symbol) has a `permanent-local` property that is non-`nil`. Permanent locals are appropriate for data pertaining to where the file came from or how to save it, rather than with how to edit the contents.

10.10.3 The Default Value of a Buffer-Local Variable

The global value of a variable with buffer-local bindings is also called the *default* value, because it is the value that is in effect whenever neither the current buffer nor the selected frame has its own binding for the variable.

The functions `default-value` and `setq-default` access and change a variable's default value regardless of whether the current buffer has a buffer-local binding. For example, you could use `setq-default` to change the default setting of `paragraph-start` for most buffers; and this would work even when you are in a C or Lisp mode buffer that has a buffer-local value for this variable.

The special forms `defvar` and `defconst` also set the default value (if they set the variable at all), rather than any buffer-local or frame-local value.

default-value *symbol* Function

This function returns *symbol*'s default value. This is the value that is seen in buffers and frames that do not have their own values for this variable. If *symbol* is not buffer-local, this is equivalent to `symbol-value` (see Section 10.7 [Accessing Variables], page 155).

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
```

10.11 Frame-Local Variables

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

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

make-variable-frame-local *variable* Command

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

If the variable is terminal-local, this function signals an error, because such variables cannot have frame-local bindings as well. See Section 28.2 [Multiple Displays], page 526. A few variables that are implemented specially in Emacs can be (and usually are) buffer-local, but can never be frame-local.

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

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

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

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

;; Make a frame-local binding for foo in a new frame.
```

```
;; Store that frame in f2.
(setq f2 (make-frame))
(modify-frame-parameters f2 '((foo . (f 2))))
```

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

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

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

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

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

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

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

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

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

10.12 Possible Future Local Variables

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

It would also be possible to implement window-local bindings. We don't know of many situations where they would be useful, and it seems that indirect buffers (see Section 26.11 [Indirect Buffers], page 493) with buffer-local bindings offer a way to handle these situations more robustly.

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

11 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.

11.1 What Is a Function?

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

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

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

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

Usually the reason we implement a function as a primitive is either because it is fundamental, because it provides a low-level interface to operating system services, or because it needs to run fast. Primitives can be modified or added only by changing the C sources and recompiling the editor. See Section B.5 [Writing Emacs Primitives], page 799.

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

special form
A *special form* is a primitive that is like a function but does not evaluate all of its arguments in the usual way. It may evaluate only some of the arguments, or may evaluate them in an unusual order, or several times. Many special forms are described in Chapter 9 [Control Structures], page 129.

macro A *macro* is a construct defined in Lisp by the programmer. It differs from a function in that it translates a Lisp expression that you write into an equivalent expression to be evaluated instead of the original expression. Macros enable Lisp programmers to do the sorts of things that special forms can do. See Chapter 12 [Macros], page 189, for how to define and use macros.

command A *command* is an object that `command-execute` can invoke; it is a possible definition for a key sequence. Some functions are commands; a function written in Lisp is a command if it contains an interactive declaration (see Section 20.2 [Defining Commands], page 320). Such a function can be called from Lisp expressions like other functions; in this case, the fact that the function is a command makes no difference.

Keyboard macros (strings and vectors) are commands also, even though they are not functions. A symbol is a command if its function definition is a command; such symbols can be invoked with `M-x`. The symbol is a function as well if the definition is a function. See Section 20.1 [Command Overview], page 319.

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

byte-code function
A *byte-code function* is a function that has been compiled by the byte compiler. See Section 2.3.15 [Byte-Code Type], page 32.

functionp *object* Function
This function returns `t` if *object* is any kind of function, or a special form or macro.

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
```

11.2 Lambda Expressions

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

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

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

11.2.1 Components of a Lambda Expression

The first element of a lambda expression is always the symbol `lambda`. This indicates that the list represents a function. The reason functions are defined to start with `lambda` is so that other lists, intended for other uses, will not accidentally be valid as functions.

The second element is a list of symbols—the argument variable names. This is called the *lambda list*. When a Lisp function is called, the argument values are matched up against the variables in the lambda list, which are given local bindings with the values provided. See Section 10.3 [Local Variables], page 148.

The documentation string is a Lisp string object placed within the function definition to describe the function for the Emacs help facilities. See Section 11.2.4 [Function Documentation], page 175.

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 20.2 [Defining Commands], page 320, 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.

11.2.2 A Simple Lambda-Expression Example

Consider for example the following function:

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

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

```
((lambda (a b c) (+ a b c))
 1 2 3)
```

This call evaluates the body of the lambda expression with the variable `a` bound to 1, `b` bound to 2, and `c` bound to 3. Evaluation of the body

adds these three numbers, producing the result 6; therefore, this call to the function returns the value 6.

Note that the arguments can be the results of other function calls, as in this example:

```
((lambda (a b c) (+ a b c))
 1 (* 2 3) (- 5 4))
```

This evaluates the arguments `1`, `(* 2 3)`, and `(- 5 4)` from left to right. Then it applies the lambda expression to the argument values 1, 6 and 1 to produce the value 8.

It is not often useful to write a lambda expression as the CAR of a form in this way. You can get the same result, of making local variables and giving them values, using the special form `let` (see Section 10.3 [Local Variables], page 148). And `let` is clearer and easier to use. In practice, lambda expressions are either stored as the function definitions of symbols, to produce named functions, or passed as arguments to other functions (see Section 11.7 [Anonymous Functions], page 182).

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

11.2.3 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? Similarly, it makes no sense to have any more arguments (either required or optional) after a `&rest` argument.

Here are some examples of argument lists and proper calls:

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

11.2.4 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 23 [Documentation], page 423, 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. *This 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.

11.3 Naming a Function

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

However, a symbol can serve as the name of a function. This happens when you put the function in the symbol's *function cell* (see Section 7.1 [Symbol Components], page 109). Then the symbol itself becomes a valid, callable function, equivalent to the list or subr-object that its function cell refers to. The contents of the function cell are also called the symbol's *function definition*. The procedure of using a symbol's function definition in place of the symbol is called *symbol function indirection*; see Section 8.1.4 [Function Indirection], page 121.

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

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

We often identify functions with the symbols used to name them. For example, we often speak of "the function `car`", not distinguishing between the symbol `car` and the primitive subr-object that is its function definition. For most purposes, there is no need to distinguish.

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

A symbol used as a function name may also be used as a variable; these two uses of a symbol are independent and do not conflict. (Some Lisp dialects, such as Scheme, do not distinguish between a symbol's value and its function definition; a symbol's value as a variable is also its function definition.) If you have not given a symbol a function definition, you cannot use it as a function; whether the symbol has a value as a variable makes no difference to this.

11.4 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 (see Section 11.2 [Lambda Expressions], page 172), *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.

There is no conflict if the same symbol *name* is also used as a variable, since the symbol's value cell is independent of the function cell. See Section 7.1 [Symbol Components], page 109.

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 a word."
  (interactive)
  (backward-word 1)
  (forward-word 1)
  (backward-char 1)
  (capitalize-word 1))
     ⇒ capitalize-backwards
```

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

defalias *name definition* Function

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

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 14.7 [Unloading], page 221).

By contrast, in programs that manipulate function definitions for other purposes, it is better to use `fset`, which does not keep such records.

See also `defsubst`, which defines a function like `defun` and tells the Lisp compiler to open-code it. See Section 11.9 [Inline Functions], page 186.

11.5 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 8 [Evaluation], page 119, for a description of evaluation.

When you write a list as an expression in your program, the function name it calls is written in your program. This means that you choose which function to call, and how many arguments to give it, when you write 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*; `funcall` enters the normal procedure for calling a function at the place where the 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 example 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 the description of `mapcar`, in Section 11.6 [Mapping Functions], page 180.

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`.

11.6 Mapping Functions

A *mapping function* applies a given function to each element of a list or other collection. Emacs Lisp has several such functions; `mapcar` and `mapconcat`, which scan a list, are described here. See Section 7.3 [Creating Symbols], page 111, for the function `mapatoms` which maps over the symbols in an obarray.

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 104).

mapcar *function sequence* Function

`mapcar` applies *function* to each element of *sequence* in turn, and returns a list of the results.

The argument *sequence* can be any kind of sequence except a char-table; that is, a list, a vector, a bool-vector, or a string. The result is always a list. The length of the result is the same as the length of *sequence*.

For example:

```
(mapcar 'car '((a b) (c d) (e f)))
        (a c e)
(mapcar '1+ [1 2 3])
        (2 3 4)
(mapcar 'char-to-string "abc")
        ("a" "b" "c")

;; Call each function in my-hooks.
(mapcar 'funcall my-hooks)

(defun mapcar* (function &rest args)
  "Apply FUNCTION to successive cars of all ARGS.
Return the list of results."
  ;; If no list is exhausted,
  (if (not (memq 'nil args))
      ;; apply function to CARs.
      (cons (apply function (mapcar 'car args))
            (apply 'mapcar* function
                   ;; Recurse for rest of elements.
                   (mapcar 'cdr args)))))

(mapcar* 'cons '(a b c) '(1 2 3 4))
        ((a . 1) (b . 2) (c . 3))
```

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"
```

11.7 Anonymous Functions

In Lisp, a function is a list that starts with `lambda`, a byte-code function compiled from such a list, or alternatively a primitive subr-object; names are "extra". Although usually functions are defined with `defun` and given names at the same time, it is occasionally more concise to use an explicit lambda expression—an anonymous function. Such a list is valid wherever a function name is.

Any method of creating such a list makes a valid function. Even this:

```
(setq silly (append '(lambda (x)) (list (list '+ (* 3 4) 'x))))
   (lambda (x) (+ 12 x))
```

This computes a list that looks like `(lambda (x) (+ 12 x))` and makes it the value (*not* the function definition!) of `silly`.

Here is how we might call this function:

```
(funcall silly 1)
⇒ 13
```

(It does *not* work to write `(silly 1)`, because this function is not the *function definition* of `silly`. We have not given `silly` any function definition, just a value as a variable.)

Most of the time, anonymous functions are constants that appear in your program. For example, you might want to pass one as an argument to the function `mapcar`, which applies any given function to each element of a list.

Here we define a function `change-property` which uses a function as its third argument:

```
(defun change-property (symbol prop function)
  (let ((value (get symbol prop)))
    (put symbol prop (funcall function value))))
```

Here we define a function that uses `change-property`, passing it a function to double a number:

```
(defun double-property (symbol prop)
  (change-property symbol prop '(lambda (x) (* 2 x))))
```

In such cases, we usually use the special form `function` instead of simple quotation to quote the anonymous function, like this:

```
(defun double-property (symbol prop)
  (change-property symbol prop
                   (function (lambda (x) (* 2 x)))))
```

Using `function` instead of `quote` makes a difference if you compile the function `double-property`. For example, if you compile the second definition of `double-property`, the anonymous function is compiled as well. By contrast, if you compile the first definition which uses ordinary `quote`, the argument passed to `change-property` is the precise list shown:

```
(lambda (x) (* x 2))
```

The Lisp compiler cannot assume this list is a function, even though it looks like one, since it does not know what `change-property` will do with the list. Perhaps it will check whether the CAR of the third element is the symbol `*`! Using `function` tells the compiler it is safe to go ahead and compile the constant function.

We sometimes write `function` instead of `quote` when quoting the name of a function, but this usage is just a sort of comment:

`(function` *symbol*`)` ≡ `(quote` *symbol*`)` ≡ `'`*symbol*

The read syntax `#'` is a short-hand for using `function`. For example,

```
#'(lambda (x) (* x x))
```

is equivalent to

```
(function (lambda (x) (* x x)))
```

function *function-object* Special Form

This special form returns *function-object* without evaluating it. In this, it is equivalent to `quote`. However, it serves as a note to the Emacs Lisp compiler that *function-object* is intended to be used only as a function, and therefore can safely be compiled. Contrast this with `quote`, in Section 8.2 [Quoting], page 125.

See `documentation` in Section 23.2 [Accessing Documentation], page 424, for a realistic example using `function` and an anonymous function.

11.8 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` in Section 8.1.4 [Function Indirection], page 121.

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. (See also `makunbound`, in Section 10.4 [Void Variables], page 150.)

```
(defun foo (x) x)
     ⇒ foo
(foo 1)
     ⇒1
(fmakunbound 'foo)
     ⇒ foo
(foo 1)
error  Symbol's function definition is void: foo
```

fset *symbol definition* Function

This function stores *definition* in the function cell of *symbol*. The result is *definition*. Normally *definition* should be a function or the name of a function, but this is not checked. The argument *symbol* is an ordinary evaluated argument.

There are three normal uses of this function:

- Copying one symbol's function definition to another—in other words, making an alternate name for a function. (If you think of this as the

definition of the new name, you should use `defalias` instead of `fset`; see Section 11.4 [Defining Functions], page 177.)

- Giving a symbol a function definition that is not a list and therefore cannot be made with `defun`. For example, you can use `fset` to give a symbol `s1` a function definition which is another symbol `s2`; then `s1` serves as an alias for whatever definition `s2` presently has. (Once again use `defalias` instead of `fset` if you think of this as the definition of `s1`.)
- In constructs for defining or altering functions. If `defun` were not a primitive, it could be written in Lisp (as a macro) using `fset`.

Here are examples of these uses:

```
;; Save foo's definition in old-foo.
(fset 'old-foo (symbol-function 'foo))

;; Make the symbol car the function definition of xfirst.
;; (Most likely, defalias would be better than fset here.)
(fset 'xfirst 'car)
     ⇒ car
(xfirst '(1 2 3))
     ⇒ 1
(symbol-function 'xfirst)
     ⇒ car
(symbol-function (symbol-function 'xfirst))
     ⇒ #<subr car>

;; Define a named keyboard macro.
(fset 'kill-two-lines "\^u2\^k")
     ⇒ "\^u2\^k"

;; Here is a function that alters other functions.
(defun copy-function-definition (new old)
  "Define NEW with the same function definition as OLD."
  (fset new (symbol-function old)))
```

When writing a function that extends a previously defined function, the following idiom is sometimes used:

```
(fset 'old-foo (symbol-function 'foo))
(defun foo ()
  "Just like old-foo, except more so."
  (old-foo)
  (more-so))
```

This does not work properly if `foo` has been defined to autoload. In such a case, when `foo` calls `old-foo`, Lisp attempts to define `old-foo` by loading a file. Since this presumably defines `foo` rather than `old-foo`, it does not produce the proper results. The only way to avoid this problem is to make sure the file is loaded before moving aside the old definition of `foo`.

But it is unmodular and unclean, in any case, for a Lisp file to redefine a function defined elsewhere. It is cleaner to use the advice facility (see Chapter 16 [Advising Functions], page 235).

11.9 Inline Functions

You can define an *inline function* by using `defsubst` instead of `defun`. An inline function works just like an ordinary function except for one thing: when you compile a call to the function, the function's definition is open-coded into the caller.

Making a function inline makes explicit calls run faster. But it also has disadvantages. For one thing, it reduces flexibility; if you change the definition of the function, calls already inlined still use the old definition until you recompile them. Since the flexibility of redefining functions is an important feature of Emacs, you should not make a function inline unless its speed is really crucial.

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.

It's possible to define a macro to expand into the same code that an inline function would execute. (See Chapter 12 [Macros], page 189.) But the macro would be limited to direct use in expressions—a macro cannot be called with `apply`, `mapcar` and so on. Also, it takes some work to convert an ordinary function into a macro. To convert it into an inline function is very easy; simply replace `defun` with `defsubst`. Since each argument of an inline function is evaluated exactly once, you needn't worry about how many times the body uses the arguments, as you do for macros. (See Section 12.6.1 [Argument Evaluation], page 193.)

Inline functions can be used and open-coded later on in the same file, following the definition, just like macros.

11.10 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 11.5 [Calling Functions], page 179.

`autoload` See Section 14.4 [Autoload], page 215.

`call-interactively`
See Section 20.3 [Interactive Call], page 325.

`commandp` See Section 20.3 [Interactive Call], page 325.

`documentation`
: See Section 23.2 [Accessing Documentation], page 424.

`eval`
: See Section 8.3 [Eval], page 126.

`funcall`
: See Section 11.5 [Calling Functions], page 179.

`function`
: See Section 11.7 [Anonymous Functions], page 182.

`ignore`
: See Section 11.5 [Calling Functions], page 179.

`indirect-function`
: See Section 8.1.4 [Function Indirection], page 121.

`interactive`
: See Section 20.2.1 [Using Interactive], page 320.

`interactive-p`
: See Section 20.3 [Interactive Call], page 325.

`mapatoms`
: See Section 7.3 [Creating Symbols], page 111.

`mapcar`
: See Section 11.6 [Mapping Functions], page 180.

`map-char-table`
: See Section 6.6 [Char-Tables], page 104.

`mapconcat`
: See Section 11.6 [Mapping Functions], page 180.

`undefined`
: See Section 21.7 [Key Lookup], page 370.

12 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 11.9 [Inline Functions], page 186.

12.1 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`.

12.2 Expansion of a Macro Call

A macro call looks just like a function call in that it is a list which starts with the name of the macro. The rest of the elements of the list are the arguments of the macro.

Evaluation of the macro call begins like evaluation of a function call except for one crucial difference: the macro arguments are the actual expressions appearing in the macro call. They are not evaluated before they are given to the macro definition. By contrast, the arguments of a function are results of evaluating the elements of the function call list.

Having obtained the arguments, Lisp invokes the macro definition just as a function is invoked. The argument variables of the macro are bound to the argument values from the macro call, or to a list of them in the case of a `&rest` argument. And the macro body executes and returns its value just as a function body does.

The second crucial difference between macros and functions is that the value returned by the macro body is not the value of the macro call. Instead, it is an alternate expression for computing that value, also known as the *expansion* of the macro. The Lisp interpreter proceeds to evaluate the expansion as soon as it comes back from the macro.

Since the expansion is evaluated in the normal manner, it may contain calls to other macros. It may even be a call to the same macro, though this is unusual.

You can see the expansion of a given macro call by calling `macroexpand`.

macroexpand *form* &optional *environment* Function

This function expands *form*, if it is a macro call. If the result is another macro call, it is expanded in turn, until something which is not a macro call results. That is the value returned by `macroexpand`. If *form* is not a macro call to begin with, it is returned as given.

Note that `macroexpand` does not look at the subexpressions of *form* (although some macro definitions may do so). Even if they are macro calls themselves, `macroexpand` does not expand them.

The function `macroexpand` does not expand calls to inline functions. Normally there is no need for that, since a call to an inline function is no harder to understand than a call to an ordinary function.

If *environment* is provided, it specifies an alist of macro definitions that shadow the currently defined macros. Byte compilation uses this feature.

```
(defmacro inc (var)
    (list 'setq var (list '1+ var)))
        inc

(macroexpand '(inc r))
        (setq r (1+ r))

(defmacro inc2 (var1 var2)
    (list 'progn (list 'inc var1) (list 'inc var2)))
        inc2

(macroexpand '(inc2 r s))
        (progn (inc r) (inc s))  ; inc not expanded here.
```

12.3 Macros and Byte Compilation

You might ask why we take the trouble to compute an expansion for a macro and then evaluate the expansion. Why not have the macro body produce the desired results directly? The reason has to do with compilation.

When a macro call appears in a Lisp program being compiled, the Lisp compiler calls the macro definition just as the interpreter would, and receives an expansion. But instead of evaluating this expansion, it compiles the expansion as if it had appeared directly in the program. As a result, the

compiled code produces the value and side effects intended for the macro, but executes at full compiled speed. This would not work if the macro body computed the value and side effects itself—they would be computed at compile time, which is not useful.

In order for compilation of macro calls to work, the macros must already be defined in Lisp when the calls to them are compiled. The compiler has a special feature to help you do this: if a file being compiled contains a `defmacro` form, the macro is defined temporarily for the rest of the compilation of that file. To make this feature work, you must put the `defmacro` in the same file where it is used, and before its first use.

Byte-compiling a file executes any `require` calls at top-level in the file. This is in case the file needs the required packages for proper compilation. One way to ensure that necessary macro definitions are available during compilation is to require the files that define them (see Section 14.6 [Named Features], page 219). To avoid loading the macro definition files when someone *runs* the compiled program, write `eval-when-compile` around the `require` calls (see Section 15.5 [Eval During Compile], page 228).

12.4 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 11.2.3 [Argument List], page 174). Macros may have a documentation string, but any `interactive` declaration is ignored since macros cannot be called interactively.

12.5 Backquote

Macros often need to construct large list structures from a mixture of constants and nonconstant parts. To make this easier, use the ‘`’ syntax (usually called *backquote*).

Backquote allows you to quote a list, but selectively evaluate elements of that list. In the simplest case, it is identical to the special form `quote` (see Section 8.2 [Quoting], page 125). For example, these two forms yield identical results:

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

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

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

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

```
(defmacro t-becomes-nil (variable)
  ‘(if (eq ,variable t)
       (setq ,variable nil)))

(t-becomes-nil foo)
     ≡ (if (eq foo t) (setq foo nil))
```

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

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

(setq list ’(hack foo bar))
     ⇒ (hack foo bar)
```

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

In old Emacs versions, before version 19.29, ‘`’ used a different syntax which required an extra level of parentheses around the entire backquote construct. Likewise, each ‘,’ or ‘,@’ substitution required an extra level of parentheses surrounding both the ‘,’ or ‘,@’ and the following expression. The old syntax required whitespace between the ‘`’, ‘,’ or ‘,@’ and the following expression.

This syntax is still accepted, for compatibility with old Emacs versions, but we recommend not using it in new programs.

12.6 Common Problems Using Macros

The basic facts of macro expansion have counterintuitive consequences. This section describes some important consequences that can lead to trouble, and rules to follow to avoid trouble.

12.6.1 Evaluating Macro Arguments Repeatedly

When defining a macro you must pay attention to the number of times the arguments will be evaluated when the expansion is executed. The following macro (used to facilitate iteration) illustrates the problem. This macro allows us to write a simple “for” loop such as one might find in Pascal.

```
(defmacro for (var from init to final do &rest body)
  "Execute a simple \"for\" loop.
For example, (for i from 1 to 10 do (print i))."
  (list 'let (list (list var init))
        (cons 'while (cons (list '<= var final)
                           (append body (list (list 'inc var)))))))
   for

(for i from 1 to 3 do
   (setq square (* i i))
   (princ (format "\n%d %d" i square)))

(let ((i 1))
  (while (<= i 3)
    (setq square (* i i))
    (princ (format "%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.

12.6.2 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 7.3 [Creating Symbols], page 111). 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.

12.6.3 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 8.3 [Eval], page 126). 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.

12.6.4 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 Writing Customization Definitions

This chapter describes how to declare user options for customization, and also customization groups for classifying them. We use the term *customization item* to include both kinds of customization definitions—as well as face definitions (see Section 38.10.2 [Defining Faces], page 754).

13.1 Common Keywords for All Kinds of Items

All kinds of customization declarations (for variables and groups, and for faces) accept keyword arguments for specifying various information. This section describes some keywords that apply to all kinds.

All of these keywords, except `:tag`, can be used more than once in a given item. Each use of the keyword has an independent effect. The keyword `:tag` is an exception because any given item can only display one name.

`:tag` *name*
: Use *name*, a string, instead of the item's name, to label the item in customization menus and buffers.

`: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. Be careful not to overdo this!

`: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 three 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.

  `(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*.

You can specify the text to use in the customization buffer by adding `:tag` *name* after the first element of the *link-data*; for example, `(info-link :tag "foo" "(emacs)Top")` makes a link to the Emacs manual which appears in the buffer as '`foo`'.

An item can have more than one external link; however, most items have none at all.

`:load` *file* Load file *file* (a string) before displaying this customization item. Loading is done with `load-library`, and only if the file is not already loaded.

`:require` *feature*
Require feature *feature* (a symbol) when installing a value for this item (an option or a face) that was saved using the customization feature. This is done by calling `require`.

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.

13.2 Defining Custom 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.

In addition to the common keywords (see Section 13.1 [Common Keywords], page 199), you can use this keyword in `defgroup`:

`:prefix` *prefix*
: If the name of an item in the group starts with *prefix*, then the tag for that item is constructed (by default) by omitting *prefix*.

  One group can have any number of prefixes.

The prefix-discarding feature is currently turned off, which means that `:prefix` currently has no effect. We did this because we found that discarding the specified prefixes often led to confusing names for options. This happened because the people who wrote the `defgroup` definitions for various groups added `:prefix` keywords whenever they make logical sense—that is, whenever the variables in the library have a common prefix.

In order to obtain good results with `:prefix`, it would be necessary to check the specific effects of discarding a particular prefix, given the specific items in a group and their names and documentation. If the resulting text is not clear, then `:prefix` should not be used in that case.

It should be possible to recheck all the customization groups, delete the `:prefix` specifications which give unclear results, and then turn this feature back on, if someone would like to do the work.

13.3 Defining Customization Variables

Use `defcustom` to declare user-editable variables.

defcustom *option default doc* [*keyword value*]... Macro

Declare *option* as a customizable user option variable. Do not quote *option*. The argument *doc* specifies the documentation string for the variable.

If *option* is void, `defcustom` initializes it to *default*. *default* should be an expression to compute the value; be careful in writing it, because it can be evaluated on more than one occasion.

`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 13.4 [Customization Types], page 203, for more information.

`:options` *list*
: Specify *list* as the list of reasonable values for use in this option.

Currently this is meaningful only when the type is `hook`. In that case, the elements of *list* should be functions that are useful as elements of the hook value. The user is not restricted to using only these functions, but they are offered as convenient alternatives.

`:version` *version*
: This option specifies that the variable was first introduced, or its default value was changed, in Emacs version *version*. The value *version* must be a string. For example,

```
(defcustom foo-max 34
  "*Maximum number of foo's allowed."
  :type 'integer
  :group 'foo
  :version "20.3")
```

`:set` *setfunction*
: Specify *setfunction* as the way to change the value of this option. The function *setfunction* should take two arguments, a symbol and the new value, and should do whatever is necessary to update the value properly for this option (which may not mean simply setting the option as a Lisp variable). The default for *setfunction* is `set-default`.

`:get` *getfunction*
: Specify *getfunction* as the way to extract the value of this option. The function *getfunction* should take one argument, a symbol, and should return the "current value" for that symbol (which need not be the symbol's Lisp value). The default is `default-value`.

`:initialize` *function*
: *function* should be a function used to initialize the variable when the `defcustom` is evaluated. It should take two arguments, the symbol and 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. This is the default `:initialize` function.

  `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).

`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`.

The `:require` option is useful for an option that turns on the operation of a certain feature. Assuming that the package is coded to check the value of the option, you still need to arrange for the package to be loaded. You can do that with `:require`. See Section 13.1 [Common Keywords], page 199. Here is an example, from the library '`paren.el`':

```
(defcustom show-paren-mode nil
  "Toggle Show Paren mode...."
  :set (lambda (symbol value)
         (show-paren-mode (or value 0)))
  :initialize 'custom-initialize-default
  :type 'boolean
  :group 'paren-showing
  :require 'paren)
```

Internally, `defcustom` uses the symbol property `standard-value` to record the expression for the default value, and `saved-value` to record the value saved by the user with the customization buffer. The `saved-value` property is actually a list whose car is an expression which evaluates to the value.

13.4 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; since types that vary at run time are rarely useful, normally you 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 13.4.4 [Type Keywords], page 208).

Some of the type symbols do not use any arguments; those are called *simple types*. For a simple type, if you do not use any keyword-value pairs, you can omit the parentheses around the type symbol. For example just `string` as a customization type is equivalent to `(string)`.

13.4.1 Simple Types

This section describes all the simple customization types.

`sexp` The value may be any Lisp object that can be printed and read back. You can use `sexp` as a fall-back for any option, if you don't want to take the time to work out a more specific type to use.

`integer` The value must be an integer, and is represented textually in the customization buffer.

`number` The value must be a number, and is represented textually in the customization buffer.

`string` The value must be a string, and the customization buffer shows just the contents, with no delimiting '"' characters and no quoting with '\'.

`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, and you can do completion with *M-*TAB.

`(file :must-match t)`
The value must be a file name for an existing file, and you can do completion with *M-*TAB.

`directory`
The value must be a directory name, and you can do completion with *M-*TAB.

`hook` The value must be a list of functions (or a single function, but that is obsolete usage). This customization type is used for hook

variables. You can use the `:options` keyword in a hook variable's `defcustom` to specify a list of functions recommended for use in the hook; see Section 13.3 [Variable Definitions], page 201.

`symbol` The value must be a symbol. It appears in the customization buffer as the name of the symbol.

`function` The value must be either a lambda expression or a function name. When it is a function name, you can do completion with *M-*TAB .

`variable` The value must be a variable name, and you can do completion with *M-*TAB .

`face` The value must be a symbol which is a face name, and you can do completion with *M-*TAB .

`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.

13.4.2 Composite Types

When none of the simple types is appropriate, you can use composite types, which build new types from other types. Here are several ways of doing that:

`(restricted-sexp :match-alternatives` *criteria*`)`
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.

`(cons` *car-type cdr-type*`)`
The value must be a cons cell, its CAR must fit *car-type*, and its CDR must fit *cdr-type*. For example, `(cons string symbol)`

is a customization type which matches values such as `("foo" . foo)`.

In the customization buffer, the CAR and the CDR are displayed and edited separately, each according to the type that you specify for it.

`(list` *element-types*`...)`
: The value must be a list with exactly as many elements as the *element-type*s you have specified; and each element must fit the corresponding *element-type*.

  For example, `(list integer string function)` describes a list of three elements; the first element must be an integer, the second a string, and the third a function.

  In the customization buffer, each element is displayed and edited separately, according to the type specified for it.

`(vector` *element-types*`...)`
: Like `list` except that the value must be a vector instead of a list. The elements work the same as in `list`.

`(choice` *alternative-types*`...)`
: The value must fit at least one of *alternative-types*. For example, `(choice integer string)` allows either an integer or a string.

  In the customization buffer, the user selects one of the alternatives 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 13.4.4 [Type Keywords], page 208.

`(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` *elements*`...)`
: The value must be a list and each element of the list must be one of the *elements* specified. This appears in the customization buffer as a checklist.

`(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.

13.4.3 Splicing into Lists

The `:inline` feature lets you splice a variable number of elements into the middle of a list or vector. You use it in a `set`, `choice` or `repeat` type which appears among the element-types of a `list` or `vector`.

Normally, each of the element-types in a `list` or `vector` describes one and only one element of the list or vector. Thus, if an element-type is a `repeat`, that specifies a list of unspecified length which appears as one element.

But when the element-type uses `:inline`, the value it matches is merged directly into the containing sequence. For example, if it matches a list with three elements, those become three elements of the overall sequence. This is analogous to using '`,@`' in the backquote construct.

For example, to specify a list whose first element must be `t` and whose remaining arguments should be zero or more of `foo` and `bar`, use this customization type:

```
(list (const t) (set :inline t foo bar))
```

This matches values such as `(t)`, `(t foo)`, `(t bar)` and `(t 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.

13.4.4 Type Keywords

You can specify keyword-argument pairs in a customization type after the type name symbol. Here are the keywords you can use, and their meanings:

`:value` *default*
: This is used for a type that appears as an alternative inside of `choice`; it specifies the default value to use, at first, if and when the user selects this alternative with the menu in the customization buffer.

  Of course, if the actual value of the option fits this alternative, it will appear showing the actual value, not *default*.

  If `nil` is not a valid value for the alternative, then it is essential to specify a valid default with `:value`.

`:format` *format-string*
: This string will be inserted in the buffer to represent the value corresponding to the type. The following '`%`' escapes are available for use in *format-string*:

'`%[`*button*`%]`'
: Display the text *button* marked as a button. The `:action` attribute specifies what the button will do if the user invokes it; its value is a function which takes two arguments—the widget which the button appears in, and the event.

  There is no way to specify two different buttons with different actions.

'`%{`*sample*`%}`'
: Show *sample* in a special face specified by `:sample-face`.

'`%v`'
: Substitute the item's value. How the value is represented depends on the kind of item, and (for variables) on the customization type.

'`%d`'
: Substitute the item's documentation string.

'`%h`'
: Like '`%d`', but if the documentation string is more than one line, add an active field to control whether to show all of it or just the first line.

'`%t`'
: Substitute the tag here. You specify the tag with the `:tag` keyword.

'`%%`'
: Display a literal '`%`'.

`:action` *action*
: Perform *action* if the user clicks on a button.

`:button-face` *face*
: Use the face *face* (a face name or a list of face names) for button text displayed with '`%[...%]`'.

`:button-prefix` *prefix*
`:button-suffix` *suffix*
: These specify the text to display before and after a button. Each can be:

  `nil`
  : No text is inserted.

  a string
  : The string is inserted literally.

  a symbol
  : The symbol's value is used.

`:tag` *tag*
: Use *tag* (a string) as the tag for the value (or part of the value) that corresponds to this type.

`:doc` *doc*
: Use *doc* as the documentation string for this value (or part of the value) that corresponds to this type. In order for this to work, you must specify a value for `:format`, and use '`%d`' or '`%h`' in that value.

The usual reason to specify a documentation string for a type is to provide more information about the meanings of alternatives inside a `:choice` type or the parts of some other composite type.

`:help-echo` *motion-doc*
: When you move to this item with `widget-forward` or `widget-backward`, it will display the string *motion-doc* in the echo area.

`:match` *function*
: Specify how to decide whether a value matches the type. The corresponding value, *function*, should be a function that accepts two arguments, a widget and a value; it should return non-`nil` if the value is acceptable.

14 Loading

Loading a file of Lisp code means bringing its contents into the Lisp environment in the form of Lisp objects. Emacs finds and opens the file, reads the text, evaluates each form, and then closes the file.

The load functions evaluate all the expressions in a file just as the `eval-current-buffer` function evaluates all the expressions in a buffer. The difference is that the load functions read and evaluate the text in the file as found on disk, not the text in an Emacs buffer.

The loaded file must contain Lisp expressions, either as source code or as byte-compiled code. Each form in the file is called a *top-level form*. There is no special format for the forms in a loadable file; any form in a file may equally well be typed directly into a buffer and evaluated there. (Indeed, most code is tested this way.) Most often, the forms are function definitions and variable definitions.

A file containing Lisp code is often called a *library*. Thus, the "Rmail library" is a file containing code for Rmail mode. Similarly, a "Lisp library directory" is a directory of files containing Lisp code.

14.1 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 14.4 [Autoload], page 215). `require` loads a file if it isn't already loaded (see Section 14.6 [Named Features], page 219). 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 '`.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 the optional argument *nosuffix* is non-`nil`, then the suffixes '`.elc`' and '`.el`' are not tried. In this case, you must specify the precise file name you want. By specifying the precise file name and using `t` for *nosuffix*, you can prevent perverse file names such as '`foo.el.el`' from being tried.

If the optional argument *must-suffix* is non-`nil`, then `load` insists that the file name used must end in either '`.el`' or '`.elc`', unless it contains an explicit directory name. If *filename* does not contain an explicit directory name, and does not end in a suffix, then `load` insists on adding one.

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 14.2 [Library Search], page 213.

If you get a warning that '`foo.elc`' is older than '`foo.el`', it means you should consider recompiling '`foo.el`'. See Chapter 15 [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 32.10 [Coding Systems], page 636.

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. `load-path` is not used, and suffixes are not appended. Use this command if you wish to specify precisely the file name to load.

load-library *library* Command

This command loads the library named *library*. It is equivalent to `load`, except in how it reads its argument interactively.

load-in-progress Variable

This variable is non-`nil` if Emacs is in the process of loading a file, and it is `nil` otherwise.

load-read-function Variable

This variable specifies an alternate expression-reading function for `load` and `eval-region` to use instead of `read`. The function should accept one argument, just as `read` does.

Normally, the variable's value is `nil`, which means those functions should use `read`.

Note: 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 Section 8.3 [Eval], page 126.

For information about how `load` is used in building Emacs, see Section B.1 [Building Emacs], page 793.

14.2 Library Search

When Emacs loads a Lisp library, it searches for the library in a list of directories specified by the variable `load-path`.

load-path User Option

The value of this variable is a list of directories to search when loading files with `load`. Each element is a string (which must be a directory name) or `nil` (which stands for the current working directory).

The value of `load-path` is initialized from the environment variable `EMACSLOADPATH`, if that exists; otherwise its default value is specified in '`emacs/src/paths.h`' when Emacs is built. Then the list is expanded by adding subdirectories of the directories in the list.

The syntax of `EMACSLOADPATH` is the same as used for `PATH`; '`:`' (or '`;`', according to the operating system) separates directory names, and '`.`' is used for the current default directory. Here is an example of how to set your `EMACSLOADPATH` variable from a `csh` '`.login`' file:

```
setenv EMACSLOADPATH .:/user/bil/emacs:/usr/local/share/emacs/20.3/lisp
```

Here is how to set it using `sh`:

```
export EMACSLOADPATH
EMACSLOADPATH=.:/user/bil/emacs:/usr/local/share/emacs/20.3/lisp
```

Here is an example of code you can place in a '`.emacs`' file to add several directories to the front of your default `load-path`:

```
(setq load-path
      (append (list nil "/user/bil/emacs"
                    "/usr/local/lisplib"
                    "~/emacs")
              load-path))
```

In this example, the path searches the current working directory first, followed then by the '`/user/bil/emacs`' directory, the '`/usr/local/lisplib`'

directory, and the '`~/emacs`' directory, which are then followed by the standard directories for Lisp code.

Dumping Emacs uses a special value of `load-path`. If the value of `load-path` at the end of dumping is unchanged (that is, still the same special value), the dumped Emacs switches to the ordinary `load-path` value when it starts up, as described above. But if `load-path` has any other value at the end of dumping, that value is used for execution of the dumped Emacs also.

Therefore, if you want to change `load-path` temporarily for loading a few libraries in '`site-init.el`' or '`site-load.el`', you should bind `load-path` locally with `let` around the calls to `load`.

The default value of `load-path`, when running an Emacs which has been installed on the system, includes two special directories (and their subdirectories as well):

"/usr/local/share/emacs/*version*/site-lisp"

and

"/usr/local/share/emacs/site-lisp"

The first one is for locally installed packages for a particular Emacs version; the second is for locally installed packages meant for use with all installed Emacs versions.

There are several reasons why a Lisp package that works well in one Emacs version can cause trouble in another. Sometimes packages need updating for incompatible changes in Emacs; sometimes they depend on undocumented internal Emacs data that can change without notice; sometimes a newer Emacs version incorporates a version of the package, and should be used only with that version.

Emacs finds these directories' subdirectories and adds them to `load-path` when it starts up. Both immediate subdirectories and subdirectories multiple levels down are added to `load-path`.

Not all subdirectories are included, though. Subdirectories whose names do not start with a letter or digit are excluded. Subdirectories named '`RCS`' are excluded. Also, a subdirectory which contains a file named '`.nosearch`' is excluded. You can use these methods to prevent certain subdirectories of the '`site-lisp`' directories from being searched.

If you run Emacs from the directory where it was built—that is, an executable that has not been formally installed—then `load-path` normally contains two additional directories. These are the `lisp` and `site-lisp` subdirectories of the main build directory. (Both are represented as absolute file names.)

locate-library *library* &optional *nosuffix path* *interactive-call* Command

This command finds the precise file name for library *library*. It searches for the library in the same way `load` does, and the argument *nosuffix* has

the same meaning as in `load`: don't add suffixes '`.elc`' or '`.el`' to the specified name *library*.

If the *path* is non-`nil`, that list of directories is used instead of `load-path`.

When `locate-library` is called from a program, it returns the file name as a string. When the user runs `locate-library` interactively, the argument *interactive-call* is `t`, and this tells `locate-library` to display the file name in the echo area.

14.3 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 32.1 [Text Representations], page 629). 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 32.10 [Coding Systems], page 636.

To make the results more predictable, Emacs always performs decoding into the multibyte representation when loading Lisp files, even if it was started with the '`--unibyte`' option. This means that string constants with non-ASCII characters translate into multibyte strings. The only exception is when a particular file specifies no decoding.

The reason Emacs is designed this way is so that Lisp programs give predictable results, regardless of how Emacs was started. In addition, this enables programs that depend on using multibyte text to work even in a unibyte Emacs. Of course, such programs should be designed to notice whether the user prefers unibyte or multibyte text, by checking `default-enable-multibyte-characters`, and convert representations appropriately.

In most Emacs Lisp programs, the fact that non-ASCII strings are multibyte strings should not be noticeable, since inserting them in unibyte buffers converts them to unibyte automatically. However, if this does make a difference, you can force a particular Lisp file to be interpreted as unibyte by writing '`-*-unibyte: t;-*-`' in a comment on the file's first line. With that designator, the file will be unconditionally be interpreted as unibyte, even in an ordinary multibyte Emacs session.

14.4 Autoload

The *autoload* facility allows you to make a function or macro known in Lisp, but put off loading the file that defines it. The first call to the function automatically reads the proper file to install the real definition and other associated code, then runs the real definition as if it had been loaded all along.

There are two ways to set up an autoloaded function: by calling `autoload`, and by writing a special "magic" comment in the source before the real definition. `autoload` is the low-level primitive for autoloading; any Lisp program can call `autoload` at any time. Magic comments are the most convenient way to make a function autoload, for packages installed along with Emacs. These comments do nothing on their own, but they serve as a guide for the command `update-file-autoloads`, which constructs calls to `autoload` and arranges to execute them when Emacs is built.

autoload *function filename* &optional *docstring interactive type* Function

This function defines the function (or macro) named *function* so as to load automatically from *filename*. The string *filename* specifies the file to load to get the real definition of *function*.

If *filename* does not contain either a directory name, or the suffix `.el` or `.elc`, then `autoload` insists on adding one of these suffixes, and it will not load from a file whose name is just *filename* with no added suffix.

The argument *docstring* is the documentation string for the function. Normally, this should be identical to the documentation string in the function definition itself. Specifying the documentation string in the call to `autoload` makes it possible to look at the documentation without loading the function's real definition.

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 23.1 [Documentation Basics], page 423), `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 consists of '`;;;###autoload`', on a line by itself, just before the real definition of the function in its autoloadable source file. The command *`M-x update-file-autoloads`* writes a corresponding `autoload` call into '`loaddefs.el`'. Building Emacs loads '`loaddefs.el`' and thus calls `autoload`. *`M-x update-directory-autoloads`* is even more powerful; it updates autoloads for all files in the current directory.

The same magic comment can copy any kind of form into '`loaddefs.el`'. If the form following the magic comment is not a function definition, it is copied verbatim. 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 'doctor "doctor"
  "\
Switch to *doctor* buffer and start giving psychotherapy."
```

```
t)
```

The backslash and newline immediately following the double-quote are a convention used only in the preloaded Lisp files such as '`loaddefs.el`'; they tell `make-docfile` to put the documentation string in the '`etc/DOC`' file. See Section B.1 [Building Emacs], page 793.

14.5 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 10.5 [Defining Variables], page 152.)

The simplest way to add an element to an alist is like this:

```
(setq minor-mode-alist
      (cons '(leif-mode " Leif") minor-mode-alist))
```

But this would add multiple elements if the library is reloaded. To avoid the problem, write this:

```
(or (assq 'leif-mode minor-mode-alist)
    (setq minor-mode-alist
          (cons '(leif-mode " Leif") minor-mode-alist)))
```

To add an element to a list just once, you can also use `add-to-list` (see Section 10.8 [Setting Variables], page 156).

Occasionally you will want to test explicitly whether a library has already been loaded. Here's one way to test, in a library, whether it has been loaded before:

```
(defvar foo-was-loaded nil)

(unless foo-was-loaded
  execute-first-time-only
  (setq foo-was-loaded t))
```

If the library uses `provide` to provide a named feature, you can use `featurep` earlier in the file to test whether the `provide` call has been executed before.

14.6 Features

`provide` and `require` are an alternative to `autoload` for loading files automatically. They work in terms of named *features*. Autoloading is triggered by calling a specific function, but a feature is loaded the first time another program asks for it by name.

A feature name is a symbol that stands for a collection of functions, variables, etc. The file that defines them should *provide* the feature. Another program that uses them may ensure they are defined by *requiring* the feature. This loads the file of definitions if it hasn't been loaded already.

To require the presence of a feature, call `require` with the feature name as argument. `require` looks in the global variable `features` to see whether the desired feature has been provided already. If not, it loads the feature from the appropriate file. This file should call `provide` at the top level to add the feature to `features`; if it fails to do so, `require` signals an error.

For example, in '`emacs/lisp/prolog.el`', the definition for `run-prolog` includes the following code:

```
(defun run-prolog ()
  "Run an inferior Prolog process, with I/O via buffer *prolog*."
  (interactive)
  (require 'comint)
  (switch-to-buffer (make-comint "prolog" prolog-program-name))
  (inferior-prolog-mode))
```

The expression `(require 'comint)` loads the file '`comint.el`' if it has not yet been loaded. This ensures that `make-comint` is defined. Features are normally named after the files that provide them, so that `require` need not be given the file name.

The '`comint.el`' file contains the following top-level expression:

```
(provide 'comint)
```

This adds `comint` to the global `features` list, so that `(require 'comint)` will henceforth know that nothing needs to be done.

When `require` is used at top level in a file, it takes effect when you byte-compile that file (see Chapter 15 [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.

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* Function

This function announces that *feature* is now loaded, or being loaded, into the current Emacs session. This means that the facilities associated with *feature* are or will be available for other Lisp programs.

The direct effect of calling `provide` is to add *feature* to the front of the list `features` if it is not already in the list. The argument *feature* must be a symbol. `provide` returns *feature*.

```
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 evaluating its contents, any function definitions or `provide` calls that occurred during the load are undone. See Section 14.4 [Autoload], page 215.

require *feature* &optional *filename* 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 suffix; a file whose name is just *feature* won't be used.

If loading the file fails to provide *feature*, `require` signals an error, '`Required feature` *feature* `was not provided`'.

featurep *feature* Function

This function returns `t` if *feature* has been provided in the current Emacs session (i.e., if *feature* is a member of `features`.)

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.

14.7 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 `loadhist-special-hooks`. This is to prevent Emacs from ceasing to function because important hooks refer to functions that are no longer defined.

If these measures are not sufficient to prevent malfunction, a library can define an explicit unload hook. If *feature*`-unload-hook` is defined, it is run as a normal hook before restoring the previous definitions, *instead of* the usual hook-removing actions. The unload hook ought to undo all the global state changes made by the library that might cease to work once the library is unloaded.

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`.

load-history Variable

This variable's value is an alist connecting library names with the names of functions and variables they define, the features they provide, and the features they require.

Each element is a list and describes one library. The CAR of the list is the name of the library, as a string. The rest of the list is composed of these kinds of objects:

- Symbols that were defined by this library.
- Lists of the form (`require` . *feature*) indicating features that were required.

- Lists of the form `(provide . feature)` indicating features that were 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 8.3 [Eval], page 126.

Preloaded libraries don't contribute to `load-history`.

loadhist-special-hooks Variable

This variable holds a list of hooks to be scanned before unloading a library, to remove functions defined in the library.

14.8 Hooks for Loading

You can ask for code to be executed if and when a particular library is loaded, by calling `eval-after-load`.

eval-after-load *library form* Function

This function arranges to evaluate *form* at the end of loading the library *library*, if and when *library* is loaded. If *library* is already loaded, it evaluates *form* right away.

The library name *library* must exactly match the argument of `load`. To get the proper results when an installed library is found by searching `load-path`, you should not include any directory names in *library*.

An error in *form* does not undo the load, but does prevent execution of the rest of *form*.

In general, well-designed Lisp programs should not use this feature. The clean and modular ways to interact with a Lisp library are (1) examine and set the library's variables (those which are meant for outside use), and (2) call the library's functions. If you wish to do (1), you can do it immediately—there is no need to wait for when the library is loaded. To do (2), you must load the library (preferably with `require`).

But it is OK to use `eval-after-load` in your personal customizations if you don't feel they must meet the design standards for programs meant for wider use.

after-load-alist Variable

An alist of expressions to evaluate if and when particular libraries are loaded. Each element looks like this:

```
(filename forms...)
```

The function `load` checks `after-load-alist` in order to implement `eval-after-load`.

15 Byte Compilation

Emacs Lisp has a *compiler* that translates functions written in Lisp into a special representation called *byte-code* that can be executed more efficiently. The compiler replaces Lisp function definitions with byte-code. When a byte-code function is called, its definition is evaluated by the *byte-code interpreter*.

Because the byte-compiled code is evaluated by the byte-code interpreter, instead of being executed directly by the machine's hardware (as true compiled code is), byte-code is completely transportable from machine to machine without recompilation. It is not, however, as fast as true compiled code.

Compiling a Lisp file with the Emacs byte compiler always reads the file as multibyte text, even if Emacs was started with '`--unibyte`', unless the file specifies otherwise. This is so that compilation gives results compatible with running the same file without compilation. See Section 14.3 [Loading Non-ASCII], page 215.

In general, any version of Emacs can run byte-compiled code produced by recent earlier versions of Emacs, but the reverse is not true. A major incompatible change was introduced in Emacs version 19.29, and files compiled with versions since that one will definitely not run in earlier versions unless you specify a special option. See Section 15.3 [Docs and Compilation], page 226. In addition, the modifier bits in keyboard characters were renumbered in Emacs 19.29; as a result, files compiled in versions before 19.29 will not work in subsequent versions if they contain character constants with modifier bits.

See Section 17.4 [Compilation Errors], page 281, for how to investigate errors occurring in byte compilation.

15.1 Performance of Byte-Compiled Code

A byte-compiled function is not as efficient as a primitive function written in C, but runs much faster than the version written in Lisp. Here is an example:

```
(defun silly-loop (n)
  "Return time before and after N iterations of a loop."
  (let ((t1 (current-time-string)))
    (while (> (setq n (1- n))
              0))
    (list t1 (current-time-string))))
⇒ silly-loop

(silly-loop 100000)
⇒ ("Fri Mar 18 17:25:57 1994"
    "Fri Mar 18 17:26:28 1994")  ; 31 seconds
```

```
(byte-compile 'silly-loop)
⇒ [Compiled code not shown]

(silly-loop 100000)
⇒ ("Fri Mar 18 17:26:52 1994"
    "Fri Mar 18 17:26:58 1994")  ; 6 seconds
```

In this example, the interpreted code required 31 seconds to run, whereas the byte-compiled code required 6 seconds. These results are representative, but actual results will vary greatly.

15.2 The Compilation Functions

You can byte-compile an individual function or macro definition with the `byte-compile` function. You can compile a whole file with `byte-compile-file`, or several files with `byte-recompile-directory` or `batch-byte-compile`.

The byte compiler produces error messages and warnings about each file in a buffer called '`*Compile-Log*`'. These report things in your program that suggest a problem but are not necessarily erroneous.

Be careful when writing macro calls in files that you may someday byte-compile. Macro calls are expanded when they are compiled, so the macros must already be defined for proper compilation. For more details, see Section 12.3 [Compiling Macros], page 190.

Normally, compiling a file does not evaluate the file's contents or load the file. But it does execute any `require` calls at top level in the file. One way to ensure that necessary macro definitions are available during compilation is to require the file that defines them (see Section 14.6 [Named Features], page 219). To avoid loading the macro definition files when someone *runs* the compiled program, write `eval-when-compile` around the `require` calls (see Section 15.5 [Eval During Compile], page 228).

byte-compile *symbol* Function

This function byte-compiles the function definition of *symbol*, replacing the previous definition with the compiled one. The function definition of *symbol* must be the actual code for the function; i.e., the compiler does not follow indirection to another symbol. `byte-compile` returns the new, compiled definition of *symbol*.

If *symbol*'s definition is a byte-code function object, `byte-compile` does nothing and returns `nil`. Lisp records only one function definition for any symbol, and if that is already compiled, non-compiled code is not available anywhere. So there is no way to "compile the same definition again."

```
(defun factorial (integer)
  "Compute factorial of INTEGER."
  (if (= 1 integer) 1
    (* integer (factorial (1- integer)))))
⇒ factorial

(byte-compile 'factorial)
⇒
#[(integer)
  "^H\301U\203^H^@\301\207\302^H\303^HS!\"\207"
  [integer 1 * factorial]
  4 "Compute factorial of INTEGER."]
```

The result is a byte-code function object. The string it contains is the actual byte-code; each character in it is an instruction or an operand of an instruction. The vector contains all the constants, variable names and function names used by the function, except for certain primitives that are coded as special instructions.

compile-defun 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.

byte-compile-file *filename* Command

This function compiles a file of Lisp code named *filename* into a file of byte-code. The output file's name is made by changing the '`.el`' suffix into '`.elc`'; if *filename* does not end in '`.el`', it adds '`.elc`' to the end of *filename*.

Compilation works by reading the input file one form at a time. If it is a definition of a function or macro, the compiled function or macro definition is written out. Other forms are batched together, then each batch is compiled, and written so that its compiled code will be executed when the file is read. All comments are discarded when the input file is read.

This command returns `t`. When called interactively, it prompts for the file name.

```
% 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 flag* Command

This function recompiles every '`.el`' file in *directory* 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`, these files are ignored. If it is non-`nil`, the user is asked whether to compile each such file.

The returned value of this command is unpredictable.

batch-byte-compile 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.

```
% emacs -batch -f batch-byte-compile *.el
```

byte-code *code-string data-vector max-stack* Function

This function actually interprets byte-code. A byte-compiled function is actually defined with a body that calls `byte-code`. Don't call this function yourself—only the byte compiler knows how to generate valid calls to this function.

In Emacs version 18, byte-code was always executed by way of a call to the function `byte-code`. Nowadays, byte-code is usually executed as part of a byte-code function object, and only rarely through an explicit call to `byte-code`.

15.3 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 give nonsense results.

If your site installs Emacs following the usual procedures, these problems will never normally occur. Installing a new version uses a new directory with

a different name; as long as the old version remains installed, its files will remain unmodified in the places where they are expected to be.

However, if you have built Emacs yourself and use it from the directory where you built it, you will experience this problem occasionally if you edit and recompile Lisp files. When it happens, you can cure the problem by reloading the file after recompiling it.

Byte-compiled files made with recent versions of Emacs (since 19.29) will not load into older versions because the older versions don't support this feature. You can turn off this feature at compile time by setting `byte-compile-dynamic-docstrings` to `nil`; then you can compile files that will load into older Emacs versions. You can do this globally, or for one source file by specifying a file-local binding for the variable. One way to do that is by adding this string to the file's first line:

```
-*-byte-compile-dynamic-docstrings: nil;-*-
```

byte-compile-dynamic-docstrings Variable

If this is non-`nil`, the byte compiler generates compiled files that are set up for dynamic loading of documentation strings.

The dynamic documentation string feature writes compiled files that use a special Lisp reader construct, '`#@`*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.

15.4 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 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

This immediately finishes loading the definition of *function* from its byte-compiled file, if it is not fully loaded already. The argument *function* may be a byte-code function object or a function name.

15.5 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.

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.

Common Lisp Note: At top level, this 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.

15.6 Byte-Code Function Objects

Byte-compiled functions have a special data type: they are *byte-code function objects*.

Internally, a byte-code function object is much like a vector; however, the evaluator handles this data type specially when it appears as a function to be called. The printed representation for a byte-code function object is like that for a vector, with an additional '`#`' before the opening '`[`'.

A byte-code function object must have at least four elements; there is no maximum number, but only the first six elements have any normal use. They are:

arglist The list of argument symbols.

byte-code The string containing the byte-code instructions.

constants The vector of Lisp objects referenced by the byte code. These include symbols used as function names and variable names.

stacksize The maximum stack size this function needs.

docstring The documentation string (if any); otherwise, `nil`. The value may be a number or a list, in case the documentation string is stored in a file. Use the function `documentation` to get the real documentation string (see Section 23.2 [Accessing Documentation], page 424).

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).

You can access the elements of a byte-code object using `aref`; you can also use `vconcat` to create a vector with the same elements.

15.7 Disassembled Byte-Code

People do not write byte-code; that job is left to the byte compiler. But we provide a disassembler to satisfy a cat-like curiosity. The disassembler converts the byte-compiled code into humanly readable form.

The byte-code interpreter is implemented as a simple stack machine. It pushes values onto a stack of its own, then pops them off to use them in calculations whose results are themselves pushed back on the stack. When a byte-code function returns, it pops a value off the stack and returns it as the value of the function.

In addition to the stack, byte-code functions can use, bind, and set ordinary Lisp variables, by transferring values between variables and the stack.

disassemble *object* &optional *stream* Command

This function prints the disassembled code for *object*. If *stream* is supplied, then output goes there. Otherwise, the disassembled code is printed to the stream `standard-output`. The argument *object* can be a function name or a lambda expression.

As a special exception, if this function is used interactively, it outputs to a buffer named '`*Disassemble*`'.

Here are two examples of using the `disassemble` function. We have added explanatory comments to help you relate the byte-code to the Lisp source; these do not appear in the output of `disassemble`. These examples show unoptimized byte-code. Nowadays byte-code is usually optimized, but we did not want to rewrite these examples, since they still serve their purpose.

```
(defun factorial (integer)
  "Compute factorial of an integer."
  (if (= 1 integer) 1
    (* integer (factorial (1- integer)))))
     ⇒ factorial

(factorial 4)
     ⇒ 24

(disassemble 'factorial)
     ⊣ byte-code for factorial:
 doc: Compute factorial of an integer.
 args: (integer)
```

```
0   constant 1              ; Push 1 onto stack.

1   varref   integer        ; Get value of integer
                            ;    from the environment
                            ;    and push the value
                            ;    onto the stack.

2   eqlsign                 ; Pop top two values off stack,
                            ;    compare them,
                            ;    and push result onto stack.

3   goto-if-nil 10          ; Pop and test top of stack;
                            ;    if nil, go to 10,
                            ;    else continue.

6   constant 1              ; Push 1 onto top of stack.

7   goto     17             ; Go to 17 (in this case, 1 will be
                            ;    returned by the function).

10  constant *              ; Push symbol * onto stack.

11  varref   integer        ; Push value of integer onto stack.

12  constant factorial      ; Push factorial onto stack.

13  varref   integer        ; Push value of integer onto stack.

14  sub1                    ; Pop integer, decrement value,
                            ;    push new value onto stack.

                            ; Stack now contains:
                            ;    - decremented value of integer
                            ;    - factorial
                            ;    - value of integer
                            ;    - *

15  call     1              ; Call function factorial using
                            ;    the first (i.e., the top) element
                            ;    of the stack as the argument;
                            ;    push returned value onto stack.

                            ; Stack now contains:
                            ;    - result of recursive
                            ;          call to factorial
                            ;    - value of integer
                            ;    - *
```

```
16  call     2              ; Using the first two
                            ;    (i.e., the top two)
                            ;    elements of the stack
                            ;    as arguments,
                            ;    call the function *,
                            ;    pushing the result onto the stack.
17  return                 ; Return the top element
                            ;    of the stack.
     ⇒ nil
```

The `silly-loop` function is somewhat more complex:

```
(defun silly-loop (n)
  "Return time before and after N iterations of a loop."
  (let ((t1 (current-time-string)))
    (while (> (setq n (1- n))
              0))
    (list t1 (current-time-string))))
     ⇒ silly-loop

(disassemble 'silly-loop)
     ⊣ byte-code for silly-loop:
 doc: Return time before and after N iterations of a loop.
 args: (n)

0   constant current-time-string  ; Push
                                  ;    current-time-string
                                  ;    onto top of stack.
1   call     0              ; Call current-time-string
                            ;     with no argument,
                            ;     pushing result onto stack.
2   varbind  t1             ; Pop stack and bind t1
                            ;    to popped value.
3   varref   n              ; Get value of n from
                            ;    the environment and push
                            ;    the value onto the stack.
4   sub1                    ; Subtract 1 from top of stack.

5   dup                     ; Duplicate the top of the stack;
                            ;    i.e., copy the top of
                            ;    the stack and push the
                            ;    copy onto the stack.
```

```
6   varset   n                ; Pop the top of the stack,
                              ;   and bind n to the value.

                              ; In effect, the sequence dup varset
                              ;   copies the top of the stack
                              ;   into the value of n
                              ;   without popping it.
7   constant 0                ; Push 0 onto stack.

8   gtr                       ; Pop top two values off stack,
                              ;   test if n is greater than 0
                              ;   and push result onto stack.
9   goto-if-nil-else-pop 17   ; Goto 17 if n <= 0
                              ;   (this exits the while loop).
                              ;   else pop top of stack
                              ;   and continue
12  constant nil              ; Push nil onto stack
                              ;   (this is the body of the loop).
13  discard                   ; Discard result of the body
                              ;   of the loop (a while loop
                              ;   is always evaluated for
                              ;   its side effects).
14  goto     3                ; Jump back to beginning
                              ;   of while loop.
17  discard                   ; Discard result of while loop
                              ;   by popping top of stack.
                              ;   This result is the value nil that
                              ;   was not popped by the goto at 9.
18  varref   t1               ; Push value of t1 onto stack.

19  constant current-time-string  ; Push
                                  ;   current-time-string
                                  ;   onto top of stack.
20  call     0                ; Call current-time-string again.

21  list2                     ; Pop top two elements off stack,
                              ;   create a list of them,
                              ;   and push list onto stack.
```

```
22  unbind   1              ; Unbind t1 in local environment.
23  return                  ; Return value of the top of stack.
      ⇒ nil
```

16 Advising Emacs Lisp Functions

The *advice* feature lets you add to the existing definition of a function, by *advising the function*. This is a clean method for a library to customize functions defined by other parts of Emacs—cleaner than redefining the whole function.

Each function can have multiple *pieces of advice*, separately defined. Each defined piece of advice can be enabled or disabled explicitly. The enabled pieces of advice for any given function actually take effect when you *activate* advice for that function, or when that function is subsequently defined or redefined.

Usage Note: Advice is useful for altering the behavior of existing calls to an existing function. If you want the new behavior for new calls, or for key bindings, it is cleaner to define a new function (or a new command) which uses the existing function.

16.1 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`).

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. 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 (arg))
  "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`. `(arg)` 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.

16.2 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.

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.

In place of the argument list in an ordinary definition, an advice definition calls for several different pieces of information.

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 16.10 [Combined Definition], page 244). Therefore, the advice expressions can use the argument variables in this list to access argument values.

This argument list must be compatible with the argument list of the original function, so that it can handle the ways the function is actually called. If more than one piece of advice specifies an argument list, then the first one (the one with the smallest position) found in the list of all classes of advice is used.

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 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 9.5.4 [Cleanups], page 144.

`compile` Compile the combined definition that is used to run the advice. This flag is ignored unless `activate` is also specified. See Section 16.10 [Combined Definition], page 244.

`disable` Initially disable this piece of advice, so that it will not be used unless subsequently explicitly enabled. See Section 16.6 [Enabling Advice], page 241.

`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.

  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.

16.3 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, but it is somewhat used like one in around-advice. It specifies 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).

16.4 Computed Advice

The macro `defadvice` resembles `defun` in that the code for the advice, and all other information about it, are explicitly stated in the source code. You can also create advice whose details are computed, using the function `ad-add-advice`.

ad-add-advice *function advice class position* Function

Calling `ad-add-advice` adds *advice* as a piece of advice to *function* in class *class*. The argument *advice* has this form:

(*name protected enabled definition*)

Here *protected* and *enabled* are flags, and *definition* is the expression that says what the advice should do. If *enabled* is `nil`, this piece of advice is initially disabled (see Section 16.6 [Enabling Advice], page 241).

If *function* already has one or more pieces of advice in the specified *class*, then *position* specifies where in the list to put the new piece of advice. The value of *position* can either be `first`, `last`, or a number (counting from 0 at the beginning of the list). Numbers outside the range are mapped to the closest extreme position.

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.

16.5 Activation of Advice

By default, advice does not take effect when you define it—only when you *activate* advice for the function that was advised. You can request the activation of advice for a function when you define the advice, by specifying the `activate` flag in the `defadvice`. But normally you activate the advice for a function by calling the function `ad-activate` or one of the other activation commands listed below.

Separating the activation of advice from the act of defining it permits you to add several pieces of advice to one function efficiently, without redefining the function over and over as each advice is added. More importantly, it permits defining advice for a function before that function is actually defined.

When a function's advice is first activated, the function's original definition is saved, and all enabled pieces of advice for that function are combined with the original definition to make a new definition. (Pieces of advice that are currently disabled are not used; see Section 16.6 [Enabling Advice], page 241.) 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`, the command also compiles the combined definition which implements the advice.

ad-activate *function* &optional *compile* Command
This command activates the advice for *function*.

To activate advice for a function whose advice is already active is not a no-op. It is a useful operation which puts into effect any changes in that function's advice since the previous activation of advice for that function.

ad-deactivate *function* Command
This command deactivates the advice for *function*.

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-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 16.6 [Enabling Advice], page 241) since the last time it was activated.

ad-start-advice Command
Turn on automatic advice activation when a function is defined or redefined. If you turn on this mode, then advice really does take effect immediately when defined.

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.

If the advised definition was constructed during "preactivation" (see Section 16.7 [Preactivation], page 242), 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.

16.6 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.

16.7 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.

16.8 Argument Access in Advice

The simplest way to access the arguments of an advised function in the body of a piece of advice is to use the same names that the function definition

uses. To do this, you need to know the names of the argument variables of the original function.

While this simple method is sufficient in many cases, it has a disadvantage: it is not robust, because it hard-codes the argument names into the advice. If the definition of the original function changes, the advice might break.

Another method is to specify an argument list in the advice itself. This avoids the need to know the original function definition's argument names, but it has a limitation: all the advice on any particular function must use the same argument list, because the argument list actually used for all the advice comes from the first piece of advice for that function.

A more robust method is to use macros that are translated into the proper access forms at activation time, i.e., when constructing the advised definition. Access macros access actual arguments by position regardless of how these actual arguments get distributed onto the argument variables of a function. This is robust because in Emacs Lisp the meaning of an argument is strictly determined by its position in the argument list.

ad-get-arg *position* Macro
This returns the actual argument that was supplied at *position*.

ad-get-args *position* Macro
This returns the list of actual arguments supplied starting at *position*.

ad-set-arg *position value* Macro
This sets the value of the actual argument at *position* to *value*

ad-set-args *position value-list* Macro
This sets the list of actual arguments starting at *position* to *value-list*.

Now an example. Suppose the function `foo` is defined as

```
(defun foo (x y &optional z &rest r) ...)
```

and is then called with

```
(foo 0 1 2 3 4 5 6)
```

which means that x is 0, y is 1, z is 2 and r is `(3 4 5 6)` within the body of `foo`. Here is what `ad-get-arg` and `ad-get-args` return in this case:

```
(ad-get-arg 0) ⇒ 0
(ad-get-arg 1) ⇒ 1
(ad-get-arg 2) ⇒ 2
(ad-get-arg 3) ⇒ 3
(ad-get-args 2) ⇒ (2 3 4 5 6)
(ad-get-args 4) ⇒ (4 5 6)
```

Setting arguments also makes sense in this example:

```
(ad-set-arg 5 "five")
```

has the effect of changing the sixth argument to `"five"`. If this happens in advice executed before the body of `foo` is run, then *r* will be `(3 4 "five" 6)` within that body.

Here is an example of setting a tail of the argument list:

```
(ad-set-args 0 '(5 4 3 2 1 0))
```

If this happens in advice executed before the body of `foo` is run, then within that body, *x* will be 5, *y* will be 4, *z* will be 3, and *r* will be `(2 1 0)` inside the body of `foo`.

These argument constructs are not really implemented as Lisp macros. Instead they are implemented specially by the advice mechanism.

16.9 Definition of Subr Argument Lists

When the advice facility constructs the combined definition, it needs to know the argument list of the original function. This is not always possible for primitive functions. When advice cannot determine the argument list, it uses `(&rest ad-subr-args)`, which always works but is inefficient because it constructs a list of the argument values. You can use `ad-define-subr-args` to declare the proper argument names for a primitive function:

ad-define-subr-args *function arglist* Function

This function specifies that *arglist* should be used as the argument list for function *function*.

For example,

```
(ad-define-subr-args 'fset '(sym newdef))
```

specifies the argument list for the function `fset`.

16.10 The Combined Definition

Suppose that a function has *n* pieces of before-advice, *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)
        other-around-m-1-body-form...
            ....
    other-around-1-body-form...
other-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, a special method is used: to call 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.

17 Debugging Lisp Programs

There are three ways to investigate a problem in an Emacs Lisp program, depending on what you are doing with the program when the problem appears.

- If the problem occurs when you run the program, you can use a Lisp debugger to investigate what is happening during execution. In addition to the ordinary debugger, Emacs comes with a source level debugger, Edebug. This chapter describes both of them.
- If the problem is syntactic, so that Lisp cannot even read the program, you can use the Emacs facilities for editing Lisp to localize it.
- If the problem occurs when trying to compile the program with the byte compiler, you need to know how to examine the compiler's input buffer.

Another useful debugging tool is the dribble file. When a dribble file is open, Emacs copies all keyboard input characters to that file. Afterward, you can examine the file to find out what input was used. See Section 37.8 [Terminal Input], page 729.

For debugging problems in terminal descriptions, the `open-termscript` function can be useful. See Section 37.9 [Terminal Output], page 734.

17.1 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 20.11 [Recursive Editing], page 355.

17.1.1 Entering the Debugger on an Error

The most important time to enter the debugger is when a Lisp error happens. This allows you to investigate the immediate causes of the error.

However, entry to the debugger is not a normal consequence of an error. Many commands frequently cause Lisp errors when invoked inappropriately (such as `C-f` at the end of the buffer), and during ordinary editing it would be very inconvenient to enter the debugger each time this happens. So if you want errors to enter the debugger, set the variable `debug-on-error` to 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`). If it is `nil`, none call the debugger.

The value can also be a list of error conditions that should call the debugger. For example, if you set it to the list `(void-variable)`, then only errors about a variable that has no value invoke the debugger.

When this variable is non-`nil`, Emacs does not create an error handler around process filter functions and sentinels. Therefore, errors in these functions also invoke the debugger. See Chapter 36 [Processes], page 689.

debug-ignored-errors User Option

This variable specifies certain kinds of errors that should not enter the debugger. Its value is a list of error condition symbols and/or regular expressions. If the error has any of those condition symbols, or if the error message matches any of the regular expressions, then that error does not enter the debugger, regardless of the value of `debug-on-error`.

The normal value of this variable lists several errors that happen often during editing but rarely result from bugs in Lisp programs. However, "rarely" is not "never"; if your program fails with an error that matches this list, you will need to change this list in order to debug the error. The easiest way is usually to set `debug-ignored-errors` to `nil`.

debug-on-signal User Option

Normally, errors that are caught by `condition-case` never run the debugger, even if `debug-on-error` is non-`nil`. In other words, `condition-case` gets a chance to handle the error before the debugger gets a chance.

If you set `debug-on-signal` to a non-`nil` value, then the debugger gets the first chance at every error; an error will invoke the debugger regardless of any `condition-case`, if it fits the criteria specified by the values of `debug-on-error` and `debug-ignored-errors`.

Warning: This variable is strong medicine! Various parts of Emacs handle errors in the normal course of affairs, and you may not even realize that errors happen there. If you set `debug-on-signal` to a non-`nil` value, those errors will enter the debugger.

Warning: `debug-on-signal` has no effect when `debug-on-error` is `nil`.

To debug an error that happens during loading of the '`.emacs`' file, use the option '`--debug-init`', which binds `debug-on-error` to `t` while loading '`.emacs`', and bypasses the `condition-case` which normally catches errors in the init file.

If your '`.emacs`' file sets `debug-on-error`, the effect may not last past the end of loading '`.emacs`'. (This is an undesirable byproduct of the code that implements the '`--debug-init`' command line option.) The best way to make '`.emacs`' set `debug-on-error` permanently is with `after-init-hook`, like this:

```
(add-hook 'after-init-hook
          '(lambda () (setq debug-on-error t)))
```

17.1.2 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 quit.

Ordinary quitting gives no information about why the program was looping. To get more information, you can set the variable `debug-on-quit` to non-`nil`. Quitting with *C-g* is not considered an error, and `debug-on-error` has no effect on the handling of *C-g*. Likewise, `debug-on-quit` has no effect on errors.

Once you have the debugger running in the middle of the infinite loop, you can proceed from the debugger using the stepping commands. If you step through the entire loop, you will probably get enough information to solve the problem.

debug-on-quit User Option

This variable determines whether the debugger is called when `quit` is signaled and not handled. If `debug-on-quit` is non-`nil`, then the debugger is called whenever you quit (that is, type *C-g*). If `debug-on-quit` is `nil`, then the debugger is not called when you quit. See Section 20.9 [Quitting], page 351.

17.1.3 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 `(debug 'debug)` into the function definition as the first form.

Any function 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't debug primitive functions (i.e., those written in C) this way.

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*.

Note: 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.

```
(defun fact (n)
  (if (zerop n) 1
      (* n (fact (1- n)))))
     ⇒ fact
(debug-on-entry 'fact)
     ⇒ fact
(fact 3)

------ Buffer: *Backtrace* ------
Entering:
* fact(3)
  eval-region(4870 4878 t)
  byte-code("...")
  eval-last-sexp(nil)
  (let ...)
  eval-insert-last-sexp(nil)
* call-interactively(eval-insert-last-sexp)
------ Buffer: *Backtrace* ------

(symbol-function 'fact)
     ⇒ (lambda (n)
          (debug (quote debug))
          (if (zerop n) 1 (* n (fact (1- n)))))
```

cancel-debug-on-entry *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 `nil` or the empty string, 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. It always returns *function-name*.

17.1.4 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*. **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 9.1 [Sequencing], page 129).

17.1.5 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 20.11 [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.

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.

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.

17.1.6 Debugger Commands

Inside the debugger (in Debugger mode), these special commands are available in addition to the usual cursor motion commands. (Keep in mind that all the usual facilities of Emacs, such as switching windows or buffers, are still available.)

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.)

Here is a list of Debugger mode commands:

`c` Exit the debugger and continue execution. When continuing is possible, it resumes execution of the program as if the debugger had never been entered (aside from any side-effects that you caused by changing variable values or data structures while inside the debugger).

Continuing is possible after entry to the debugger due to function entry or exit, explicit invocation, or quitting. You cannot continue if the debugger was entered because of an error.

`d` Continue execution, but enter the debugger the next time any Lisp function is called. This allows you to step through the subexpressions of an expression, seeing what values the subexpressions compute, and what else they do.

The stack frame made for the function call which enters the debugger in this way will be flagged automatically so that the debugger will be called again when the frame is exited. You can use the `u` command to cancel this flag.

`b` Flag the current frame so that the debugger will be entered when the frame is exited. Frames flagged in this way are marked with stars in the backtrace buffer.

`u` Don't enter the debugger when the current frame is exited. This cancels a `b` command on that frame. The visible effect is to remove the star from the line in the backtrace buffer.

`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.

17.1.7 Invoking the Debugger

Here we describe in full detail the function `debug` that is used to invoke the debugger.

debug &rest *debugger-args* Function

This function enters the debugger. It switches buffers to a buffer named '`*Backtrace*`' (or '`*Backtrace*<2>`' if it is the second recursive entry to the debugger, etc.), and fills it with information about the stack of Lisp function calls. It then enters a recursive edit, showing the backtrace buffer in Debugger mode.

The Debugger mode `c` 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 '`Entering:`' as a line of text at the top of the buffer.

`debug` `debug` as first argument indicates a call to `debug` because of entry to a function that was set to debug on entry. The debugger displays '`Entering:`', 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 list form when `debug-on-next-call` is non-`nil`. The debugger displays the following as the top line in the buffer:

```
Beginning evaluation of function call form:
```

`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 '`Return 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 '`Signaling:`' followed by the error signaled and any arguments to `signal`. For example,

```
(let ((debug-on-error t))
  (/ 1 0))

------ Buffer: *Backtrace* ------
Signaling: (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.

17.1.8 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). Presumably this function will enter 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`.

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`: in this case, to 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)))))))

       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-region(1973 2142 #<buffer *scratch*>)
  byte-code("...  for eval-print-last-sexp ...")
  eval-print-last-sexp(nil)
* call-interactively(eval-print-last-sexp)
---------- Buffer: backtrace-output ------------
```

The character '`*`' indicates a frame whose debug-on-exit flag is set.

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, for the debugger, 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 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`.

17.2 Edebug

Edebug is a source-level debugger for Emacs Lisp programs with which you can:

- Step through evaluation, stopping before and after each expression.
- Set conditional or unconditional breakpoints.
- Stop when a specified condition is true (the global break event).

- Trace slow or fast, stopping briefly at each stop point, or at each breakpoint.
- Display expression results and evaluate expressions as if outside of Edebug.
- Automatically re-evaluate a list of expressions and display their results each time Edebug updates the display.
- Output trace info on function enter and exit.
- Stop when an error occurs.
- Display a backtrace, omitting Edebug's own frames.
- Specify argument evaluation for macros and defining forms.
- Obtain rudimentary coverage testing and frequency counts.

The first three sections below should tell you enough about Edebug to enable you to use it.

17.2.1 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 17.2.2 [Instrumenting], page 258, for alternative ways to instrument code.

Once a function is instrumented, any call to the function activates Edebug. 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, depending on which Edebug execution mode you have selected. The default execution mode is step, which does stop execution. See Section 17.2.3 [Edebug Execution Modes], page 259.

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*. This buffer is temporarily read-only.

An arrow at the left margin 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 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 show with periods the stop points found 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.

17.2.2 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.

Once you have loaded Edebug, the command *C-M-x* (`eval-defun`) is redefined so that when invoked with a prefix argument on a definition, it instruments the definition before evaluating it. (The source code itself is not modified.) If the variable `edebug-all-defs` is non-`nil`, that inverts the meaning of the prefix argument: then *C-M-x* instruments the definition *unless* it has a prefix argument. The default value of `edebug-all-defs` is `nil`. The command *M-x edebug-all-defs* toggles the value of the variable `edebug-all-defs`.

If `edebug-all-defs` is non-`nil`, then the commands `eval-region`, `eval-current-buffer`, and `eval-buffer` also instrument any definitions they evaluate. Similarly, `edebug-all-forms` controls whether `eval-region` should instrument *any* form, even non-defining forms. This doesn't apply to loading or evaluations in the minibuffer. The command *M-x edebug-all-forms* toggles this option.

Another command, *M-x edebug-eval-top-level-form*, is available to instrument any top-level form regardless of the values of `edebug-all-defs` and `edebug-all-forms`.

While Edebug is active, the command *I* (`edebug-instrument-callee`) instruments the definition of the function or macro called by the list form after point, if is not already instrumented. This is possible only if Edebug knows where to find the source for that function; 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 17.2.4 [Jumping], page 260), 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. Edebug cannot know what a user-defined macro will do with the arguments of a macro call, so you must tell it; see Section 17.2.15 [Instrumenting Macro Calls], page 272, for details.

When Edebug is about to instrument code for the first time in a session, it runs the hook `edebug-setup-hook`, then sets it to `nil`. You can use this to arrange to load Edebug specifications (see Section 17.2.15 [Instrumenting Macro Calls], page 272) associated with a package you are using, but actually load them only if 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 17.2.9 [Edebug Eval], page 265, for other evaluation functions available inside of Edebug.

17.2.3 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 for now, just 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 17.2.5 [Edebug Misc], page 262.

`t` Trace: pause 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 17.2.6 [Breakpoints], page 262.

`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. But see the `edebug-continue-kbd-macro` option (see Section 17.2.16 [Edebug Options], page 278).

When you enter a new Edebug level, the initial execution mode comes from the value of the variable `edebug-initial-mode`. 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.

17.2.4 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 17.2.6 [Breakpoints], page 262, for the details on breakpoints.

These commands may fail to work as expected in case of nonlocal exit, because a nonlocal exit 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 forward over one expression (`edebug-forward-sexp`).

`o` Run the program until the end of the containing sexp.

`i` Step into the function or macro called by the form after point.

The `h` command proceeds to the stop point near the current location of point, using a temporary breakpoint. See Section 17.2.6 [Breakpoints], page 262, for more information about breakpoints.

The `f` command runs the program forward over one expression. More precisely, it sets a temporary breakpoint at the position that `C-M-f` would reach, then executes in go mode so that the program will stop at breakpoints.

With a prefix argument *n*, the temporary breakpoint is placed *n* sexps beyond point. If the containing list ends before *n* more elements, then the place to stop is after the containing expression.

Be careful that the position `C-M-f` finds is a place that the program will really get to; this may not be true in a `cond`, for example.

The `f` command does `forward-sexp` starting at point, rather than at the stop point, for flexibility. If you want to execute one expression *from the current stop point*, type `w` first, 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.

17.2.5 Miscellaneous Edebug Commands

Some miscellaneous Edebug commands are described here.

`?` Display the help message for Edebug (`edebug-help`).

`C-]` Abort one level back to the previous command level (`abort-recursive-edit`).

`q` Return to the top level editor command loop (`top-level`). This exits all recursive editing levels, including all levels of Edebug activity. However, instrumented code protected with `unwind-protect` or `condition-case` forms may resume debugging.

`Q` Like `q` but don't stop even for protected code (`top-level-nonstop`).

`r` Redisplay the most recently known expression result in the echo area (`edebug-previous-result`).

`d` Display a backtrace, excluding Edebug's own functions for clarity (`edebug-backtrace`).

You cannot use debugger commands in the backtrace buffer in Edebug as you would in the standard debugger.

The backtrace buffer is killed automatically when you continue execution.

From the Edebug recursive edit, you may invoke commands that activate Edebug again recursively. Any time 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`.

17.2.6 Breakpoints

Edebug's step mode stops execution at the next stop point reached. There are three other ways to stop Edebug execution once it has started: breakpoints, the global break condition, and source breakpoints.

While using Edebug, you can specify *breakpoints* in the program you are testing: points where execution should stop. You can set a breakpoint at any stop point, as defined in Section 17.2.1 [Using Edebug], page 257. 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 *condition* evaluates to 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 forgets all its 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.

17.2.6.1 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 (`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.

17.2.6.2 Source Breakpoints

All breakpoints in a definition are forgotten each time you reinstrument it. 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, 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`.

17.2.7 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 17.2.16 [Edebug Options], page 278.

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, 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 17.2.5 [Edebug Misc], page 262).

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.

17.2.8 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` Temporarily view the outside window configuration (`edebug-view-outside`).

`p` Temporarily display the outside current buffer with point at its outside position (`edebug-bounce-point`). 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 with `w` from a source code buffer.

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.

17.2.9 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 things that Edebug explicitly saves and restores. See Section 17.2.14 [The Outside Context], page 270, for details on this process.

`e` *exp* RET
Evaluate expression *exp* in the context outside of Edebug (`edebug-eval-expression`). That is, Edebug tries to minimize its interference with the evaluation.

`M-:` *exp* RET
Evaluate expression *exp* in the context of Edebug itself.

`C-x C-e` Evaluate the expression before point, in the context outside of Edebug (`edebug-eval-last-sexp`).

Edebug supports evaluation of expressions containing references to lexically bound symbols created by the following constructs in '`cl.el`' (version 2.03 or later): `lexical-let`, `macrolet`, and `symbol-macrolet`.

17.2.10 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 that use variables not currently valid do not interrupt your debugging.

Here is an example of what the evaluation list window looks like after several expressions have been added to it:

```
(current-buffer)
#<buffer *scratch*>
;----------------------------------------------------------------
(selected-window)
#<window 16 on *scratch*>
;----------------------------------------------------------------
(point)
196
;----------------------------------------------------------------
bad-var
"Symbol's value as variable is void: bad-var"
;----------------------------------------------------------------
(recursion-depth)
0
;----------------------------------------------------------------
this-command
eval-last-sexp
;----------------------------------------------------------------
```

To delete a group, move point into it and type *C-c C-d*, or simply delete the text for the group and update the evaluation list with *C-c C-u*. To add a new expression to the evaluation list, insert the expression at a suitable place, and insert a new comment line. (You need not insert dashes in the comment line—its contents don't matter.) Then type *C-c C-u*.

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.

17.2.11 Printing in Edebug

If an expression in your program produces a value containing circular list structure, you may get an error when Edebug attempts to print it.

One way to cope with circular structure is to set `print-length` or `print-level` to truncate the printing. Edebug does this for you; it binds `print-length` and `print-level` to 50 if they were `nil`. (Actually, the variables `edebug-print-length` and `edebug-print-level` specify the values to use within Edebug.) See Section 18.6 [Output Variables], page 292.

edebug-print-length User Option

If non-`nil`, bind `print-length` to this while printing results in Edebug. The default value is `50`.

edebug-print-level User Option

If non-`nil`, bind `print-level` to this while printing results in Edebug. The default value is `50`.

You can also print circular structures and structures that share elements more informatively by using the '`cust-print`' package.

To load '`cust-print`' and activate custom printing only for Edebug, simply use the command *M-x edebug-install-custom-print*. To restore the standard print functions, use *M-x edebug-uninstall-custom-print*.

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`, bind `print-circle` to this while printing results in Edebug. The default value is `nil`.

Other programs can also use custom printing; see '`cust-print.el`' for details.

17.2.12 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 17.2.3 [Edebug Execution Modes], page 259).

When trace recording is enabled, each function entry and exit adds lines to the trace buffer. A function entry record looks like '`::::{`' followed by the function name and argument values. A function exit record looks like '`::::}`' 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. All the arguments are evaluated. `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.

17.2.13 Coverage Testing

Edebug provides rudimentary coverage testing and display of execution frequency.

Coverage testing works by comparing the result of each expression with the previous result; each form in the program is considered "covered" if it has returned two different values since you began testing coverage in the current Emacs session. Thus, to do coverage testing on your program, execute it under various conditions and note whether it behaves correctly; Edebug will tell you when you have tried enough different conditions that each form has returned two different values.

Coverage testing makes execution slower, so it is only done if `edebug-test-coverage` is non-`nil`. Frequency counting is performed for all execution of an instrumented function, even if the execution mode is Go-nonstop, and regardless of whether coverage testing is enabled.

Use *M-x edebug-display-freq-count* to display both the coverage information and the frequency counts for a definition.

edebug-display-freq-count Command
This command displays the frequency count data for each line of the current definition.

The frequency counts appear as comment lines after each line of code, and you can undo all insertions with one `undo` command. The counts appear under the '`(`' before an expression or the '`)`' after an expression, or on the last character of a variable. To simplify the display, a count is not shown if it is equal to the count of an earlier expression on the same line.

The character '`=`' following the count for an expression says that the expression has returned the same value each time it was evaluated. In other words, it is not yet "covered" for coverage testing purposes.

To clear the frequency count and coverage data for a definition, simply reinstrument it with `eval-defun`.

For example, after evaluating `(fac 5)` with a source breakpoint, and setting `edebug-test-coverage` to `t`, when the breakpoint is reached, the frequency data looks like this:

```
(defun fac (n)
  (if (= n 0) (edebug))
;#6           1      0 =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.

17.2.14 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.

17.2.14.1 Checking Whether to Stop

Whenever Edebug is entered, it needs to save and restore certain data before even deciding whether to make trace information or stop the program.

- `max-lisp-eval-depth` and `max-specpdl-size` are both incremented once to reduce Edebug's impact on the stack. You could, however, still run out of stack space when using Edebug.
- The state of keyboard macro execution is saved and restored. While Edebug is active, `executing-macro` is bound to `edebug-continue-kbd-macro`.

17.2.14.2 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 27.16 [Window Configurations], page 521). When you exit Edebug (by continuing the program), it restores the previous window configuration.

Emacs redisplays only when it pauses. Usually, when you continue execution, the program comes back into 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. 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, but some of these 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 17.2.14.2 [Edebug Display Update], page 270).

  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.

17.2.14.3 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 33.6 [Match Data], page 660.
- `last-command`, `this-command`, `last-command-char`, `last-input-char`, `last-input-event`, `last-command-event`, `last-event-frame`, `last-nonmenu-event`, and `track-mouse`. Commands used within Edebug do not affect these variables outside of Edebug.

  The key sequence returned by `this-command-keys` is changed by executing commands within Edebug 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`.

17.2.15 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, use `def-edebug-spec`.

def-edebug-spec *macro specification* Macro

Specify which expressions of a call to macro *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.

The *macro* argument can actually be any symbol, not just a macro name.

Here is a simple example that defines the specification for the `for` example macro (see Section 12.6.1 [Argument Evaluation], page 193), followed by an alternative, equivalent specification.

```
(def-edebug-spec for
  (symbolp "from" form "to" form "do" &rest form))

(def-edebug-spec for
  (symbolp ['from form] ['to form] ['do body]))
```

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.

17.2.15.1 Specification List

A *specification list* is required for an Edebug specification if some arguments of a macro call are evaluated while others are not. Some elements in a specification list match one or more arguments, but others modify the processing of all following elements. The latter, called *specification keywords*, are symbols beginning with '`&`' (such as `&optional`).

A specification list may contain sublists which match arguments that are themselves lists, or it may contain vectors used for grouping. Sublists and groups thus subdivide the specification list into a hierarchy of levels. Specification keywords apply only to the remainder of the sublist or group they are contained in.

When a specification list involves alternatives or repetition, matching it against an actual macro call may require backtracking. See Section 17.2.15.2 [Backtracking], page 276, for more details.

Edebug specifications provide the power of regular expression matching, plus some context-free grammar constructs: the matching of sublists with balanced parentheses, recursive processing of forms, and recursion via indirect specifications.

Here's a table of the possible elements of a specification list, with their meanings:

`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 below.

`&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 below.

`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 17.2.15.2 [Backtracking], page 276, for more details. Also see the `let` example below.

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 below.

Otherwise, the symbol should be a predicate. The predicate is called with the argument and the specification fails if the predicate returns `nil`. In either case, that argument is not instrumented.

Some suitable predicates include `symbolp`, `integerp`, `stringp`, `vectorp`, and `atom`.

`[`*elements*`...]`
: A vector of elements groups the elements into a single *group specification*. Its meaning has nothing to do with vectors.

"*string*"
: The argument should be a symbol named *string*. This specification is equivalent to the quoted symbol, `'`*symbol*, where the name of *symbol* is the *string*, but the string form is preferred.

`(vector` *elements*`...)`
: The argument should be a vector whose elements must match the *elements* in the specification. See the backquote example below.

`(`*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 below. 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 below.

`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 use this 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 below.

17.2.15.2 Backtracking in Specifications

If a specification fails to match at some point, this does not necessarily mean a syntax error will be signaled; instead, *backtracking* will take place until all alternatives have been exhausted. Eventually every element of the argument list must be matched by some element in the specification, and every required element in the specification must match some argument.

When a syntax error is detected, it might not be reported until much later after higher-level alternatives have been exhausted, and with the point positioned further from the real error. But if backtracking is disabled when an error occurs, it can be reported immediately. Note that backtracking is also reenabled automatically in several situations; it is reenabled when a new alternative is established by `&optional`, `&rest`, or `&or`, or at the start of processing a sublist, group, or indirect specification. The effect of enabling or disabling backtracking is limited to the remainder of the level currently being processed and lower levels.

Backtracking is disabled while matching any of the form specifications (that is, `form`, `body`, `def-form`, and `def-body`). These specifications will match any form so any error must be in the form itself rather than at a higher level.

Backtracking is also disabled after successfully matching a quoted symbol or string specification, since this usually indicates a recognized construct. But if you have a set of alternative constructs that all begin with the same symbol, you can usually work around this constraint by factoring the symbol out of the alternatives, e.g., `["foo" &or [first case] [second case] ...]`.

Most needs are satisfied by these two ways that bactracking 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.

17.2.15.3 Specification Examples

It may be easier to understand Edebug specifications by studying the examples provided here.

A `let` special form has a sequence of bindings and a body. Each of the bindings is either a symbol or a sublist with a symbol and optional expression. In the specification below, notice the `gate` inside of the sublist to prevent backtracking once a sublist is found.

```
(def-edebug-spec let
  ((&rest
    &or symbolp (gate symbolp &optional form))
   body))
```

Edebug uses the following specifications for `defun` and `defmacro` and the associated argument list and `interactive` specifications. It is necessary to handle interactive forms specially since an expression argument it is actually evaluated outside of the function body.

```
(def-edebug-spec defmacro defun) ; Indirect ref to defun spec.
(def-edebug-spec defun
  (&define name lambda-list
           [&optional stringp]   ; Match the doc string, if present.
           [&optional ("interactive" interactive)]
           def-body))

(def-edebug-spec lambda-list
  (([&rest arg]
    [&optional ["&optional" arg &rest arg]]
    &optional ["&rest" arg]
    )))

(def-edebug-spec interactive
  (&optional &or stringp def-form))     ; Notice: def-form
```

The specification for backquote below illustrates how to match dotted lists and use `nil` to terminate recursion. It also illustrates how components of a vector may be matched. (The actual specification defined by Edebug does not support dotted lists because doing so causes very deep recursion that could fail.)

```
(def-edebug-spec ‘ (backquote-form))    ; Alias just for clarity.

(def-edebug-spec backquote-form
```

```
(&or ([&or "," ",@"] &or ("quote" backquote-form) form)
     (backquote-form . [&or nil backquote-form])
     (vector &rest backquote-form)
     sexp))
```

17.2.16 Edebug Options

These options affect the behavior of Edebug:

edebug-setup-hook User Option

Functions to call before Edebug is used. Each time it is set to a new value, Edebug will call those functions once and then `edebug-setup-hook` is reset to `nil`. You could use this to load up Edebug specifications associated with a package you are using but only when you also use Edebug. See Section 17.2.2 [Instrumenting], page 258.

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 17.2.2 [Instrumenting], page 258.

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 17.2.2 [Instrumenting], page 258.

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 17.2.14.2 [Edebug Display Update], page 270.

edebug-save-displayed-buffer-points User Option

If this is non-`nil`, Edebug saves and restores point in all displayed buffers.

Saving and restoring point in other buffers is necessary if you are debugging code that changes the point of a buffer which is displayed in a non-selected window. If Edebug or the user then selects the window, point in that buffer will move to the window's value of point.

Saving and restoring point in all buffers is expensive, since it requires selecting each window twice, so enable this only if you need it. See Section 17.2.14.2 [Edebug Display Update], page 270.

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 17.2.3 [Edebug Execution Modes], page 259.

edebug-trace User Option

Non-`nil` means display a trace of 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.

The default value is `nil`.

Also see `edebug-tracing`, in Section 17.2.12 [Trace Buffer], page 268.

edebug-test-coverage User Option

If non-`nil`, Edebug tests coverage of all expressions debugged. See Section 17.2.13 [Coverage Testing], page 269.

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 17.2.3 [Edebug Execution Modes], page 259.

edebug-on-error User Option

Edebug binds `debug-on-error` to this value, if `debug-on-error` was previously `nil`. See Section 17.2.7 [Trapping Errors], page 264.

edebug-on-quit User Option

Edebug binds `debug-on-quit` to this value, if `debug-on-quit` was previously `nil`. See Section 17.2.7 [Trapping Errors], page 264.

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 17.2.6.1 [Global Break Condition], page 263.

17.3 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.)

17.3.1 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 insert a close parenthesis at the end of the file and type `C-M-b` (`backward-sexp`). This will move you to the beginning of the defun that is unbalanced. (Then type `C-SPC C-_ C-u C-SPC` to set the mark there, undo the insertion of the close parenthesis, and finally return to the mark.)

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.

17.3.2 Excess Close Parentheses

To deal with an excess close parenthesis, first insert an open parenthesis at the beginning of the file, back up over it, and type *C-M-f* to find the end of the unbalanced defun. (Then type *C-*SPC *C-_ C-u C-*SPC to set the mark there, undo the insertion of the open parenthesis, and finally return to the mark.)

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 fit the intended nesting of parentheses, and you have put back those parentheses, *C-M-q* should not change anything.

17.4 Debugging Problems in Compilation

When an error happens during byte compilation, it is normally due to invalid syntax in the program you are compiling. The compiler prints a suitable error message in the '`*Compile-Log*`' buffer, and then stops. The message may state a function name in which the error was found, or it may not. Either way, here is how to find out where in the file the error occurred.

What you should do is switch to the buffer '`*Compiler Input*`'. (Note that the buffer name starts with a space, so it does not show up in *M-x list-buffers*.) This buffer contains the program being compiled, and point shows how far the byte compiler was able to read.

If the error was due to invalid Lisp syntax, point shows exactly where the invalid syntax was *detected*. The cause of the error is not necessarily near by! Use the techniques in the previous section to find the error.

If the error was detected while compiling a form that had been read successfully, then point is located at the end of the form. In this case, this technique can't localize the error precisely, but can still show you which function to check.

18 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 17.

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).

18.1 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 17). 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 inverses. 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.

18.2 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.

`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 white-space preceding the significant text.

Here is an example of reading from a stream that is a marker, initially positioned at the beginning of the buffer shown. The value read is the symbol `This`.

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

(setq m (set-marker (make-marker) 1 (get-buffer "foo")))
     ⇒ #<marker at 1 in foo>
(read m)
     ⇒ This
m
     ⇒ #<marker at 5 in foo>   ;; Before the first space.
```

Here we read from the contents of a string:

```
(read "(When in) the course")
     ⇒ (When in)
```

The following example reads from the minibuffer. The prompt is: '`Lisp expression: `'. (That is always the prompt used when you read from the stream `t`.) The user's input is shown following the prompt.

```
(read t)
     ⇒ 23
---------- Buffer: Minibuffer ----------
Lisp expression: 23 RET
---------- Buffer: Minibuffer ----------
```

Finally, here is an example of a stream that is a function, named `useless-stream`. Before we use the stream, we initialize the variable `useless-list` to a list of characters. Then each call to the function `useless-stream` obtains the next character in the list or unreads a character by adding it to the front of the list.

```
(setq useless-list (append "XY()" nil))
     ⇒ (88 89 40 41)
```

```
(defun useless-stream (&optional unread)
  (if unread
      (setq useless-list (cons unread useless-list))
    (prog1 (car useless-list)
           (setq useless-list (cdr useless-list)))))
     ⇒ useless-stream
```

Now we read using the stream thus constructed:

```
(read 'useless-stream)
     ⇒ XY

useless-list
     ⇒ (40 41)
```

Note that the open and close parentheses remain in the list. The Lisp reader encountered the open parenthesis, decided that it ended the input, and unread it. Another attempt to read from the stream at this point would read '`()`' and return `nil`.

get-file-char Function

This function is used internally as an input stream to read from the input file opened by the function `load`. Don't use this function yourself.

18.3 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`.

18.4 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.

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 5.5 [Building Lists], page 80). 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.

18.5 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 17, 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 4.6 [String Conversion], page 66, for other ways to obtain the printed representation of a Lisp object as a string.

with-output-to-string *body...* Macro

This macro executes the *body* forms with `standard-output` set up to feed output into a string. Then it returns that string.

For example, if the current buffer name is '`foo`',

```
(with-output-to-string
  (princ "The buffer is ")
  (princ (buffer-name)))
```

returns `"The buffer is foo"`.

18.6 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`.

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.

19 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, but only while it is in use for reading an argument.

19.1 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 28.8 [Minibuffers and Frames], page 537.)

The minibuffer's window is normally a single line. You can resize it temporarily with the window sizing commands; it reverts to its normal size when the minibuffer is exited. You can resize it permanently by using the window sizing commands in the frame's other window, when the minibuffer is not active. If the frame contains just a minibuffer, you can change the minibuffer's size by changing the frame's size.

If a command uses a minibuffer while there is an active minibuffer, this is called a *recursive minibuffer*. The first minibuffer is named ‘ `*Minibuf-0*`’. Recursive minibuffers are named by incrementing the number at the end of the name. (The names begin with a space so that they won't show up in normal buffer lists.) Of several recursive minibuffers, the innermost (or most recently entered) is the active minibuffer. We usually call this “the” minibuffer. You can permit or forbid recursive minibuffers by setting the variable `enable-recursive-minibuffers` or by putting properties of that name on command symbols (see Section 19.9 [Minibuffer Misc], page 316).

Like other buffers, a minibuffer may use any of several local keymaps (see Chapter 21 [Keymaps], page 361); these contain various exit commands and in some cases completion commands (see Section 19.5 [Completion], page 301).

- `minibuffer-local-map` is for ordinary input (no completion).
- `minibuffer-local-ns-map` is similar, except that SPC exits just like RET. This is used mainly for Mocklisp compatibility.
- `minibuffer-local-completion-map` is for permissive completion.
- `minibuffer-local-must-match-map` is for strict completion and for cautious completion.

19.2 Reading Text Strings with the Minibuffer

Most often, the minibuffer is used to read text as a string. It can also be used to read a Lisp object in textual form. The most basic primitive for minibuffer input is `read-from-minibuffer`; it can do either one.

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 20.2 [Defining Commands], page 320.

read-from-minibuffer *prompt-string* &optional *initial-contents keymap read hist default inherit-input-method* Function

This function is the most general way to get input through the minibuffer. By default, it accepts arbitrary text and returns it as a string; however, if *read* is non-`nil`, then it uses `read` to convert the text into a Lisp object (see Section 18.3 [Input Functions], page 286).

The first thing this function does is to activate a minibuffer and display it with *prompt-string* as the prompt. This value must be a string. Then the user can edit text in the minibuffer.

When the user types a command to exit the minibuffer, `read-from-minibuffer` constructs the return value from the text in the minibuffer. Normally it returns a string containing that text. However, if *read* is non-`nil`, `read-from-minibuffer` reads the text and returns the resulting Lisp object, unevaluated. (See Section 18.3 [Input Functions], page 286, for information about reading.)

The argument *default* specifies a default value to make available through the history commands. It should be a string, or `nil`. If *read* is non-`nil`, then *default* is also used as the input to `read`, if the user enters empty input. However, in the usual case (where *read* is `nil`), `read-from-minibuffer` does not return *default* when the user enters empty input; it returns an empty string, `""`. In this respect, it is different from all the other minibuffer input functions in this chapter.

If *keymap* is non-`nil`, that keymap is the local keymap to use in the minibuffer. If *keymap* is omitted or `nil`, the value of `minibuffer-local-map` is used as the keymap. Specifying a keymap is the most important way to customize the minibuffer for various applications such as completion.

The argument *hist* specifies which history list variable to use for saving the input and for history commands used in the minibuffer. It defaults to `minibuffer-history`. See Section 19.4 [Minibuffer History], page 300.

If the variable `minibuffer-allow-text-properties` is non-`nil`, then the string which is returned includes whatever text properties were present in the minibuffer. Otherwise all the text properties are stripped when the value is returned.

If the argument *inherit-input-method* is non-`nil`, then the minibuffer inherits the current input method (see Section 32.11 [Input Methods],

page 645) and the setting of `enable-multibyte-characters` (see Section 32.1 [Text Representations], page 629) from whichever buffer was current before entering the minibuffer.

If *initial-contents* is a string, `read-from-minibuffer` inserts it into the minibuffer, leaving point at the end, before the user starts to edit the text. The minibuffer appears with this text as its initial contents.

Alternatively, *initial-contents* can be a cons cell of the form `(`*string* `.` *position*`)`. This means to insert *string* in the minibuffer but put point *position* characters from the beginning, rather than at the end.

Usage note: The *initial-contents* argument and the *default* argument are two alternative features for more or less the same job. It does not make sense to use both features in a single call to `read-from-minibuffer`. In general, we recommend using *default*, since this permits the user to insert the default value when it is wanted, but does not burden the user with deleting it from the minibuffer on other occasions.

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* and *initial* are used as in `read-from-minibuffer`. The keymap used is `minibuffer-local-map`.

The optional argument *history*, if non-nil, specifies a history list and optionally the initial position in the list. The optional argument *default* specifies a default value to return if the user enters null input; it should be a string. The optional argument *inherit-input-method* specifies whether to inherit the current buffer's input method.

This function is a simplified interface to the `read-from-minibuffer` function:

```
(read-string prompt initial history default inherit)

(let ((value
       (read-from-minibuffer prompt initial nil nil
                             history default inherit)))
  (if (equal value "")
      default
    value))
```

minibuffer-allow-text-properties Variable

If this variable is `nil`, then `read-from-minibuffer` strips all text properties from the minibuffer input before returning it. Since all minibuffer input uses `read-from-minibuffer`, this variable applies to all minibuffer input.

Note that the completion functions 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	`next-history-element`
M-p	`previous-history-element`
M-r	`next-matching-history-element`
M-s	`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.

(`read-no-blanks-input` *prompt initial*)

(`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`

19.3 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)

(read-from-minibuffer prompt initial nil t)
```

Here is an example in which we supply the string `"(testing)"` as initial input:

```
(read-minibuffer
 "Enter an expression: " (format "%s" '(testing)))

;; Here is how the minibuffer is displayed:

---------- Buffer: Minibuffer ----------
Enter an expression: (testing)*
---------- Buffer: Minibuffer ----------
```

The user can type RET immediately to use the initial input as a default, or can edit the input.

eval-minibuffer *prompt* &optional *initial* Function

This function reads a Lisp expression using the minibuffer, evaluates it, then returns the result. The arguments *prompt* and *initial* are used as in `read-from-minibuffer`.

This function simply evaluates the result of a call to `read-minibuffer`:

```
(eval-minibuffer prompt initial)

(eval (read-minibuffer prompt initial))
```

edit-and-eval-command *prompt form* Function

This function reads a Lisp expression in the minibuffer, and then evaluates it. The difference between this command and `eval-minibuffer` is that here the initial *form* is not optional and it is treated as a Lisp object to be converted to printed representation rather than as a string of text. It is printed with `prin1`, so if it is a string, double-quote characters (‘"’) appear in the initial text. See Section 18.5 [Output Functions], page 289.

The first thing `edit-and-eval-command` does is to activate the minibuffer with *prompt* as the prompt. Then it inserts the printed representation of *form* in the minibuffer, and lets the user edit it. When the user exits the minibuffer, the edited text is read with `read` and then evaluated. The resulting value becomes the value of `edit-and-eval-command`.

In the following example, we offer the user an expression with initial text which is a valid form already:

```
(edit-and-eval-command "Please edit: " '(forward-word 1))

;; After evaluation of the preceding expression,
;;    the following appears in the minibuffer:

---------- Buffer: Minibuffer ----------
Please edit: (forward-word 1)*
---------- Buffer: Minibuffer ----------
```

Typing RET right away would exit the minibuffer and evaluate the expression, thus moving point forward one word. `edit-and-eval-command` returns `nil` in this example.

19.4 Minibuffer History

A *minibuffer history list* records previous minibuffer inputs so the user can reuse them conveniently. A history list is actually a symbol, not a list; it is a variable whose value is a list of strings (previous inputs), most recent first.

There are many separate history lists, used for different kinds of inputs. It's the Lisp programmer's job to specify the right history list for each use of the minibuffer.

The basic minibuffer input functions `read-from-minibuffer` and `completing-read` both accept an optional argument named *hist* which is how you specify the history list. Here are the possible values:

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* (an integer, counting from zero which specifies the most recent element of the history).

  If you specify *startpos*, then you should also specify that element of the history as the initial minibuffer contents, for consistency.

If you don't specify *hist*, then the default history list `minibuffer-history` is used. For other standard history lists, see below. You can also create your own history list variable; just initialize it to `nil` before the first use.

Both `read-from-minibuffer` and `completing-read` add new elements to the history list automatically, and provide commands to allow the user to reuse items on the list. The only thing your program needs to do to use a history list is to initialize it and to pass its name to the input functions when you wish. But it is safe to modify the list by hand when the minibuffer input functions are not using it.

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.

19.5 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.

19.5.1 Basic Completion Functions

The two functions `try-completion` and `all-completions` have nothing in themselves to do with minibuffers. We describe them in this chapter so as to keep them near the higher-level completion features that do use the minibuffer.

try-completion *string collection* &optional *predicate* Function

This function returns the longest common substring of all possible completions of *string* in *collection*. The value of *collection* must be an alist, an obarray, or a function that implements a virtual set of strings (see below).

Completion compares *string* against each of the permissible completions specified by *collection*; if the beginning of the permissible completion equals *string*, it matches. If no permissible completions match, `try-completion` returns `nil`. If only one permissible completion matches, and the match is exact, then `try-completion` returns `t`. Otherwise, the value is the longest initial sequence common to all the permissible completions that match.

If *collection* is an alist (see Section 5.8 [Association Lists], page 91), the CARs of the alist elements form the set of permissible completions.

If *collection* is an obarray (see Section 7.3 [Creating Symbols], page 111), the names of all symbols in the obarray form the set of permissible completions. The global variable `obarray` holds an obarray containing the names of all interned Lisp symbols.

Note that the only valid way to make a new obarray is to create it empty and then add symbols to it one by one using `intern`. Also, you cannot intern a given symbol in more than one obarray.

If the argument *predicate* is non-`nil`, then it must be a function of one argument. 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 cons cell from the alist (the CAR of which is a string) or else it is a symbol (*not* a symbol name) from the obarray.

You can also use a symbol that is a function as *collection*. Then the function is solely responsible for performing completion; `try-completion` returns whatever this function returns. The function is called with three arguments: *string*, *predicate* and `nil`. (The reason for the third argument is so that the same function can be used in `all-completions` and do the appropriate thing in either case.) See Section 19.5.6 [Programmed Completion], page 311.

In the first of the following examples, the string '`foo`' is matched by three of the alist CARs. All of the matches begin with the characters '`fooba`', so that is the result. In the second example, there is only one possible match, and it is exact, so the value is `t`.

```
(try-completion
 "foo"
 '(("foobar1" 1) ("barfoo" 2) ("foobaz" 3) ("foobar2" 4)))
       "fooba"

(try-completion "foo" '(("barfoo" 2) ("foo" 3)))
       t
```

In the following example, numerous symbols begin with the characters '`forw`', and all of them begin with the word '`forward`'. In most of the symbols, this is followed with a '`-`', but not in all, so no more than '`forward`' can be completed.

```
(try-completion "forw" obarray)
       "forward"
```

Finally, in the following example, only two of the three possible matches pass the predicate `test` (the string '`foobaz`' is too short). Both of those begin with the string '`foobar`'.

```
(defun test (s)
  (> (length (car s)) 6))
       test
(try-completion
 "foo"
 '(("foobar1" 1) ("barfoo" 2) ("foobaz" 3) ("foobar2" 4))
 'test)
       "foobar"
```

all-completions *string collection* &optional *predicate nospace* Function

This function returns a list of all possible completions of *string*. The arguments to this function are the same as those of `try-completion`.

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 19.5.6 [Programmed Completion], page 311.

If *nospace* is non-`nil`, completions that start with a space are ignored unless *string* also starts with a space.

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")
```

completion-ignore-case Variable

If the value of this variable is non-`nil`, Emacs does not consider case significant in completion.

19.5.2 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 hist 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 *collection* and *predicate* to the function `try-completion`. This happens in certain commands bound in the local keymaps used for completion.

If *require-match* is `nil`, the exit commands work regardless of the input in the minibuffer. If *require-match* is `t`, the usual minibuffer exit commands won't exit unless the input completes to an element of *collection*. If *require-match* is neither `nil` nor `t`, then the exit commands won't exit unless the input already in the buffer matches an element of *collection*.

However, empty input is always permitted, regardless of the value of *require-match*; in that case, `completing-read` returns *default*. The value of *default* (if non-`nil`) is also available to the user through the history commands.

The user can exit with null input by typing RET with an empty minibuffer. Then `completing-read` returns "". This is how the user requests whatever default the command uses for the value being read. The user can return using RET in this way regardless of the value of *require-match*, and regardless of whether the empty string is included in *collection*.

The function `completing-read` works by calling `read-minibuffer`. It 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 19.5.3 [Completion Commands], page 305.

The argument *hist* specifies which history list variable to use for saving the input and for minibuffer history commands. It defaults to `minibuffer-history`. See Section 19.4 [Minibuffer History], page 300.

If *initial* is non-`nil`, `completing-read` inserts it into the minibuffer as part of the input. Then it allows the user to edit the input, providing several commands to attempt completion. In most cases, we recommend using *default*, and not *initial*.

If the argument *inherit-input-method* is non-`nil`, then the minibuffer inherits the current input method (see Section 32.11 [Input Methods],

page 645) and the setting of `enable-multibyte-characters` (see Section 32.1 [Text Representations], page 629) from whichever buffer was current before entering the minibuffer.

Completion ignores case when comparing the input against the possible matches, if the built-in variable `completion-ignore-case` is non-`nil`. See Section 19.5.1 [Basic Completion], page 302.

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 three variables to pass information to the commands that actually do completion. These variables are `minibuffer-completion-table`, `minibuffer-completion-predicate` and `minibuffer-completion-confirm`. For more information about them, see Section 19.5.3 [Completion Commands], page 305.

19.5.3 Minibuffer Commands That Do Completion

This section describes the keymaps, commands and user options used in the minibuffer to do completion.

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`

with other characters bound as in `minibuffer-local-map` (see Section 19.2 [Text from Minibuffer], page 296).

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:

?	`minibuffer-completion-help`
SPC	`minibuffer-complete-word`
TAB	`minibuffer-complete`
C-j	`minibuffer-complete-and-exit`
RET	`minibuffer-complete-and-exit`

with other characters bound as in `minibuffer-local-map`.

minibuffer-completion-table Variable

The value of this variable is the alist or obarray 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-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 34 [Syntax Tables], page 669.

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-confirm Variable

When the value of this variable is non-`nil`, Emacs asks for confirmation of a completion before exiting the minibuffer. The function `minibuffer-complete-and-exit` checks the value of this variable before it exits.

minibuffer-completion-help Command

This function creates a list of the possible completions of the current minibuffer contents. It works by calling `all-completions` using the value of the variable `minibuffer-completion-table` as the *collection* argument, and the value of `minibuffer-completion-predicate` as the *predicate* argument. The list of completions is displayed as text in a buffer named '`*Completions*`'.

display-completion-list *completions* Function

This function displays *completions* to the stream in `standard-output`, usually a buffer. (See Chapter 18 [Read and Print], page 283, 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, or a list of two strings, which is printed as if the strings were concatenated.

This function is called by `minibuffer-completion-help`. The most common way to use it is together with `with-output-to-temp-buffer`, like this:

```
(with-output-to-temp-buffer "*Completions*"
  (display-completion-list
    (all-completions (buffer-string) my-alist)))
```

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.

19.5.4 High-Level Completion Functions

This section describes the higher-level convenient functions for reading certain sorts of names with completion.

In most cases, you should not call these functions in the middle of a Lisp function. When possible, do all minibuffer input as part of reading the arguments for a command, in the `interactive` specification. See Section 20.2 [Defining Commands], page 320.

read-buffer *prompt* &optional *default existing* Function

This function reads the name of a buffer and returns it as a string. The argument *default* is the default name to use, the value to return if the user exits with an empty minibuffer. If non-`nil`, it should be a string or a buffer. It is mentioned in the prompt, but is not inserted in the minibuffer as initial input.

If *existing* is non-`nil`, then the name specified must be that of an existing buffer. The usual commands to exit the minibuffer do not exit if the text

is not valid, and RET does completion to attempt to find a valid name. (However, *default* is not checked for validity; it is returned, whatever it is, if the user exits with the minibuffer empty.)

In the following example, the user enters '`minibuffer.t`', and then types RET. The argument *existing* is `t`, and the only buffer name starting with the given input is '`minibuffer.texi`', so that name is the value.

```
(read-buffer "Buffer name? " "foo" t)
;; After evaluation of the preceding expression,
;;    the following prompt appears,
;;    with an empty minibuffer:

---------- Buffer: Minibuffer ----------
Buffer name? (default foo) *
---------- Buffer: Minibuffer ----------

;; The user types minibuffer.t RET.
     ⇒ "minibuffer.texi"
```

read-buffer-function Variable

This variable specifies how to read buffer names. For example, if you set this variable to `iswitchb-read-buffer`, all Emacs commands that call `read-buffer` to read a buffer name will actually use the `iswitchb` package to read it.

read-command *prompt* &optional *default* Function

This function reads the name of a command and returns it as a Lisp symbol. The argument *prompt* is used as in `read-from-minibuffer`. Recall that a command is anything for which `commandp` returns `t`, and a command name is a symbol for which `commandp` returns `t`. See Section 20.3 [Interactive Call], page 325.

The argument *default* specifies what to return if the user enters null input. It can be a symbol or a string; if it is a string, `read-command` interns it before returning it. If *default* is `nil`, that means no default has been specified; then if the user enters null input, the return value is `nil`.

```
(read-command "Command name? ")

;; After evaluation of the preceding expression,
;;    the following prompt appears with an empty minibuffer:

---------- Buffer: Minibuffer ----------
Command name?
---------- Buffer: Minibuffer ----------
```

If the user types *forward-c* RET, then this function returns `forward-char`.

The `read-command` function is a simplified interface to `completing-read`. It uses the variable `obarray` so as to complete in the set of extant Lisp

symbols, and it uses the `commandp` predicate so as to accept only command names:

```
(read-command prompt)
≡
(intern (completing-read prompt obarray
                         'commandp t nil))
```

read-variable *prompt* &optional *default* Function

This function reads the name of a user variable and returns it as a symbol.

The argument *default* specifies what to return if the user enters null input. It can be a symbol or a string; if it is a string, `read-variable` interns it before returning it. If *default* is `nil`, that means no default has been specified; then if the user enters null input, the return value is `nil`.

```
(read-variable "Variable name? ")

;; After evaluation of the preceding expression,
;;    the following prompt appears,
;;    with an empty minibuffer:

---------- Buffer: Minibuffer ----------
Variable name? ⋆
---------- Buffer: Minibuffer ----------
```

If the user then types *fill-p* RET, `read-variable` returns `fill-prefix`.

This function is similar to `read-command`, but uses the predicate `user-variable-p` instead of `commandp`:

```
(read-variable prompt)
≡
(intern
 (completing-read prompt obarray
                  'user-variable-p t nil))
```

See also the functions `read-coding-system` and `read-non-nil-coding-system`, in Section 32.10.4 [User-Chosen Coding Systems], page 639.

19.5.5 Reading File Names

Here is another high-level completion function, designed for reading a file name. It provides special features including automatic insertion of the default directory.

read-file-name *prompt* &optional *directory default existing initial* Function

This function reads a file name in the minibuffer, prompting with *prompt* and providing completion. If *default* is non-`nil`, then the function returns

default if the user just types RET . *default* is not checked for validity; it is returned, whatever it is, if the user exits with the minibuffer empty.

If *existing* is non-`nil`, then the user must specify the name of an existing file; RET performs completion to make the name valid if possible, and then refuses to exit if it is not valid. If the value of *existing* is neither `nil` nor `t`, then RET also requires confirmation after completion. If *existing* is `nil`, then the name of a nonexistent file is acceptable.

The argument *directory* specifies the directory to use for completion of relative file names. If `insert-default-directory` is non-`nil`, *directory* is also inserted in the minibuffer as initial input. It defaults to the current buffer's value of `default-directory`.

If you specify *initial*, that is an initial file name to insert in the buffer (after *directory*, if that is inserted). In this case, point goes at the beginning of *initial*. The default for *initial* is `nil`—don't insert any file name. To see what *initial* does, try the command *C-x C-v*. **Note:** we recommend using *default* rather than *initial* in most cases.

Here is an example:

```
(read-file-name "The file is ")

;; After evaluation of the preceding expression,
;;    the following appears in the minibuffer:

---------- Buffer: Minibuffer ----------
The file is /gp/gnu/elisp/⋆
---------- Buffer: Minibuffer ----------
```

Typing *manual* TAB results in the following:

```
---------- Buffer: Minibuffer ----------
The file is /gp/gnu/elisp/manual.texi⋆
---------- Buffer: Minibuffer ----------
```

If the user types RET , `read-file-name` returns the file name as the string `"/gp/gnu/elisp/manual.texi"`.

insert-default-directory User Option

This variable is used by `read-file-name`. 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.

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 ----------
```

19.5.6 Programmed Completion

Sometimes it is not possible to create an alist or an obarray containing all the intended possible completions. In such a case, you can supply your own function to compute the completion of a given string. This is called *programmed completion*.

To use this feature, pass a symbol with a function definition as the *collection* argument to `completing-read`. The function `completing-read` arranges to pass your completion function along to `try-completion` and `all-completions`, which will then let your function do all the work.

The completion function should accept three arguments:

- The string to be completed.
- The predicate function to filter possible matches, or `nil` if none. Your function should call the predicate for each possible match, and ignore the possible match if the predicate returns `nil`.
- A flag specifying the type of operation.

There are three flag values for three operations:

- `nil` specifies `try-completion`. The completion function should return the completion of the specified string, or `t` if the string is a unique and exact match already, or `nil` if the string matches no possibility.

  If the string is an exact match for one possibility, but also matches other longer possibilities, the function should return the string, not `t`.
- `t` specifies `all-completions`. The completion function should return a list of all possible completions of the specified string.
- `lambda` specifies a test for an exact match. The completion function should return `t` if the specified string is an exact match for some possibility; `nil` otherwise.

It would be consistent and clean for completion functions to allow lambda expressions (lists that are functions) as well as function symbols as *collection*, but this is impossible. Lists as completion tables are already assigned another meaning—as alists. It would be unreliable to fail to handle an alist

normally because it is also a possible function. So you must arrange for any function you wish to use for completion to be encapsulated in a symbol.

Emacs uses programmed completion when completing file names. See Section 24.8.6 [File Name Completion], page 457.

19.6 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 20.4 [Command Loop Info], page 328) is either `nil` or a list—then it uses a dialog box or pop-up menu to ask the question. Otherwise, it uses keyboard input. You can force use of the mouse or use of keyboard input by binding `last-nonmenu-event` to a suitable value around the call.

Strictly speaking, `yes-or-no-p` uses the minibuffer and `y-or-n-p` does not; but it seems best to describe them together.

y-or-n-p *prompt* Function

This function asks the user a question, expecting input in the echo area. It returns `t` if the user types *y*, `nil` if the user types *n*. This function also accepts SPC to mean yes and DEL to mean no. It accepts *C-]* 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 38.3 [The Echo Area], page 740), 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 33.5 [Search and Replace], page 659.

In the following example, the user first types *q*, which is invalid. At the next prompt the user types *y*.

```
(y-or-n-p "Do you need a lift? ")

;; After evaluation of the preceding expression,
;;    the following prompt appears in the echo area:

---------- Echo area ----------
Do you need a lift? (y or n)
---------- Echo area ----------

;; If the user then types q, the following appears:

---------- Echo area ----------
Please answer y or n.  Do you need a lift? (y or n)
---------- Echo area ----------

;; When the user types a valid answer,
;;    it is displayed after the question:

---------- Echo area ----------
Do you need a lift? (y or n) y
---------- Echo area ----------
```

We show successive lines of echo area messages, but only one actually appears on the screen at a time.

y-or-n-p-with-timeout *prompt seconds default-value* Function

Like `y-or-n-p`, except that if the user fails to answer within *seconds* seconds, this function stops waiting and returns *default-value*. It works by setting up a timer; see Section 37.7 [Timers], page 727. 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 ----------
```

19.7 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* Function

This function asks the user a series of questions, reading a single-character answer in the echo area for each one.

The value of *list* specifies the objects to ask questions about. It should be either a list of objects or a generator function. If it is a function, it should expect no arguments, and should return either the next object to ask about, or `nil` meaning stop asking questions.

The argument *prompter* specifies how to ask each question. If *prompter* is a string, the question text is computed like this:

`(format` *prompter object*`)`

where *object* is the next object to ask about (as obtained from *list*).

If not a string, *prompter* should be a function of one argument (the next object to ask about) and should return the question text. If the value is a string, that is the question to ask the user. The function can also return `t` meaning do act on this object (and don't ask the user), or `nil` meaning ignore this object (and don't ask the user).

The argument *actor* says how to act on the answers that the user gives. It should be a function of one argument, and it is called with each object

that the user says yes for. Its argument is always an object obtained from *list*.

If the argument *help* is given, it should be a list of this form:

(*singular plural action*)

where *singular* is a string containing a singular noun that describes the objects conceptually being acted on, *plural* is the corresponding plural noun, and *action* is a transitive verb describing what *actor* does.

If you don't specify *help*, the default is `("object" "objects" "act on")`.

Each time a question is asked, the user may enter `y`, `Y`, or SPC to act on that object; `n`, `N`, or DEL to skip that object; `!` to act on all following objects; ESC or `q` to exit (skip all following objects); `.` (period) to act on the current object and then exit; or `C-h` to get help. These are the same answers that `query-replace` accepts. The keymap `query-replace-map` defines their meaning for `map-y-or-n-p` as well as for `query-replace`; see Section 33.5 [Search and Replace], page 659.

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.

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 20.4 [Command Loop Info], page 328) is either `nil` or a list—then it uses a dialog box or pop-up menu to ask the question. In this case, it does not use keyboard input or the echo area. You can force use of the mouse or use of keyboard input by binding `last-nonmenu-event` to a suitable value around the call.

The return value of `map-y-or-n-p` is the number of objects acted on.

19.8 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.

19.9 Minibuffer Miscellany

This section describes some basic functions and variables related to minibuffers.

exit-minibuffer Command

This command exits the active minibuffer. It is normally bound to keys in minibuffer local keymaps.

self-insert-and-exit Command

This command exits the active minibuffer after inserting the last character typed on the keyboard (found in `last-command-char`; see Section 20.4 [Command Loop Info], page 328).

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* Command

This command replaces the minibuffer contents with the value of the previous (older) history element that matches *pattern* (a regular expression).

next-matching-history-element *pattern* Command

This command replaces the minibuffer contents with the value of the next (newer) history element that matches *pattern* (a regular expression).

minibuffer-prompt Function

This function returns the prompt string of the currently active minibuffer. If no minibuffer is active, it returns `nil`.

minibuffer-prompt-width Function

This function returns the display width of the prompt string of the currently active minibuffer. If no minibuffer is active, it returns 0.

minibuffer-setup-hook Variable
This is a normal hook that is run whenever the minibuffer is entered. See Section 22.6 [Hooks], page 420.

minibuffer-exit-hook Variable
This is a normal hook that is run whenever the minibuffer is exited. See Section 22.6 [Hooks], page 420.

minibuffer-help-form Variable
The current value of this variable is used to rebind `help-form` locally inside the minibuffer (see Section 23.5 [Help Functions], page 429).

active-minibuffer-window Function
This function returns the currently active minibuffer window, or `nil` if none is currently active.

minibuffer-window &optional *frame* Function
This function returns the minibuffer window used for frame *frame*. If *frame* is `nil`, that stands for the current frame. Note that the minibuffer window used by a frame need not be part of that frame—a frame that has no minibuffer of its own necessarily uses some other frame's minibuffer window.

window-minibuffer-p *window* Function
This function returns non-`nil` if *window* is a minibuffer window.

It is not correct to determine whether a given window is a minibuffer by comparing it with the result of `(minibuffer-window)`, because there can be more than one minibuffer window if there is more than one frame.

minibuffer-window-active-p *window* Function
This function returns non-`nil` if *window*, assumed to be a minibuffer window, is currently active.

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.

Finally, some functions and variables deal with recursive minibuffers (see Section 20.11 [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. The minibuffer command `next-matching-history-element` (normally *M-s* in the minibuffer) uses this feature.

20 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.

20.1 Command Loop Overview

The first thing the command loop must do is read a key sequence, which is a sequence of events that translates into a command. It does this by calling the function `read-key-sequence`. Your Lisp code can also call this function (see Section 20.6.1 [Key Sequence Input], page 344). Lisp programs can also do input at a lower level with `read-event` (see Section 20.6.2 [Reading One Event], page 345) or discard pending input with `discard-input` (see Section 20.6.4 [Event Input Misc], page 348).

The key sequence is translated into a command through the currently active keymaps. See Section 21.7 [Key Lookup], page 370, 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 20.3 [Interactive Call], page 325).

To execute a command requires first reading the arguments for it. This is done by calling `command-execute` (see Section 20.3 [Interactive Call], page 325). For commands written in Lisp, the `interactive` specification says how to read the arguments. This may use the prefix argument (see Section 20.10 [Prefix Command Arguments], page 353) or may read with prompting in the minibuffer (see Chapter 19 [Minibuffers], page 295). For example, the command `find-file` has an `interactive` specification which says to read a file name using the minibuffer. The command's function body does not use the minibuffer; if you call this command from Lisp code as a function, you must supply the file name string as an ordinary Lisp function argument.

If the command is a string or vector (i.e., a keyboard macro) then `execute-kbd-macro` is used to execute it. You can call this function yourself (see Section 20.14 [Keyboard Macros], page 358).

To terminate the execution of a running command, type `C-g`. This character causes *quitting* (see Section 20.9 [Quitting], page 351).

pre-command-hook Variable

The editor command loop runs this normal hook before each command. At that time, `this-command` contains the command that is about to run, and `last-command` describes the previous command. See Section 22.6 [Hooks], page 420.

post-command-hook Variable

The editor command loop runs this normal hook after each command (including commands terminated prematurely by quitting or by errors), and also when the command loop is first entered. At that time, `this-command` describes the command that just ran, and `last-command` describes the command before that. See Section 22.6 [Hooks], page 420.

Quitting is suppressed while running `pre-command-hook` and `post-command-hook`. If an error happens while executing one of these hooks, it terminates execution of the hook, and clears the hook variable to `nil` so as to prevent an infinite loop of errors.

20.2 Defining Commands

A Lisp function becomes a command when its body contains, at top level, a form that calls the special form `interactive`. This form does nothing when actually executed, but its presence serves as a flag to indicate that interactive calling is permitted. Its argument controls the reading of arguments for an interactive call.

20.2.1 Using `interactive`

This section describes how to write the `interactive` form that makes a Lisp function an interactively-callable command.

interactive *arg-descriptor* Special Form

This special form declares that the function in which it appears is a command, and that it may therefore be called interactively (via `M-x` or by entering a key sequence bound to it). The argument *arg-descriptor* declares how to compute the arguments to the command when the command is called interactively.

A command may be called from Lisp programs like any other function, but then the caller supplies the arguments and *arg-descriptor* has no effect.

The `interactive` form has its effect because the command loop (actually, its subroutine `call-interactively`) scans through the function definition looking for it, before calling the function. Once the function is called, all its body forms including the `interactive` form are executed, but at this time `interactive` simply returns `nil` without even evaluating its argument.

There are three possibilities for the argument *arg-descriptor*:

- It may be omitted or `nil`; then the command is called with no arguments. This leads quickly to an error if the command requires one or more arguments.

- It may be a 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.

  If this expression reads keyboard input (this includes using the minibuffer), keep in mind that the integer value of point or the mark before reading input may be incorrect after reading input. This is because 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 only after reading the keyboard input:

  ```
  (interactive
   (let ((string (read-string "Foo: " nil 'my-history)))
     (list (region-beginning) (region-end) string)))
  ```

- It may be a string; then its contents should consist of a code character followed by a prompt (which some code characters use and some ignore). The prompt ends either with the end of the string or with a newline. Here is a simple example:

  ```
  (interactive "bFrobnicate buffer: ")
  ```

  The code letter '`b`' says to read the name of an existing buffer, with completion. The buffer name is the sole argument passed to the command. The rest of the string is a prompt.

  If there is a newline character in the string, it terminates the prompt. If the string does not end there, then the rest of the string should contain another code character and prompt, specifying another argument. You can specify any number of arguments in this way.

  The prompt string can use '`%`' to include previous argument values (starting with the first argument) in the prompt. This is done using `format` (see Section 4.7 [Formatting Strings], page 67). For example, here is how you could read the name of an existing buffer followed by a new name to give to that buffer:

  ```
  (interactive "bBuffer to rename: \nsRename buffer %s to: ")
  ```

  If the first character in the string is '`*`', then an error is signaled if the buffer is read-only.

  If the first character in the string is '`@`', and if the key sequence used to invoke the command includes any mouse events, then the window associated with the first of those events is selected before the command is run.

You can use '`*`' and '`@`' together; the order does not matter. Actual reading of arguments is controlled by the rest of the prompt string (starting with the first character that is not '`*`' or '`@`').

20.2.2 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 19.5 [Completion], page 301). `?` displays a list of possible completions.

Existing
: Require the name of an existing object. An invalid name is not accepted; the commands to exit the minibuffer do not exit if the current input is not valid.

Default
: A default value of some sort is used if the user enters no text in the minibuffer. The default depends on the code character.

No I/O
: This code letter computes an argument without reading any input. Therefore, it does not use a prompt string, and any prompt string you supply is ignored.

  Even though the code letter doesn't use a prompt string, you must follow it with a newline if it is not the last code character in the string.

Prompt
: A prompt immediately follows the code character. The prompt ends either with the end of the string or with a newline.

Special
: This code character is meaningful only at the beginning of the interactive string, and it does not look for a prompt or a newline. It is a single, isolated character.

Here are the code character descriptions for use with `interactive`:

'`*`'
: Signal an error if the current buffer is read-only. Special.

'`@`'
: Select the window mentioned in the first mouse event in the key sequence that invoked this command. Special.

'`a`'
: A function name (i.e., a symbol satisfying `fboundp`). Existing, Completion, Prompt.

'`b`'
: The name of an existing buffer. By default, uses the name of the current buffer (see Chapter 26 [Buffers], page 479). 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 29.1 [Point], page 551). No I/O.

'`D`' A directory name. The default is the current default directory of the current buffer, `default-directory` (see Section 37.3 [System Environment], page 718). 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 20.5 [Input Events], page 330. 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 24.8 [File Names], page 451). The default directory is `default-directory`. Existing, Completion, Default, Prompt.

'`F`' A file name. The file need not exist. 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 21.1 [Keymap Terminology], page 361). 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.

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 is asked to try again. The prefix argument, if any, is not used. Prompt.

'`N`' The numeric prefix argument; but if there is no prefix argument, read a number as with *n*. Requires a number. See Section 20.10 [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 19.2 [Text from Minibuffer], page 296). 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.

'`v`' A variable declared to be a user option (i.e., satisfying the predicate `user-variable-p`). See Section 19.5.4 [High-Level Completion], page 307. 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 19.3 [Object from Minibuffer], page 298. Prompt.

'`X`' A Lisp form is read as with *x*, but then evaluated 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 32.10 [Coding Systems], page 636. 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.

20.2.3 Examples of Using `interactive`

Here are some examples of `interactive`:

```
(defun foo1 ()              ; foo1 takes no arguments,
    (interactive)           ;    just moves forward two words.
    (forward-word 2))
     ⇒ foo1

(defun foo2 (n)             ; foo2 takes one argument,
    (interactive "p")       ;    which is the numeric prefix.
    (forward-word (* 2 n)))
     ⇒ foo2

(defun foo3 (n)             ; foo3 takes one argument,
    (interactive "nCount:") ;    which is read with the Minibuffer.
    (forward-word (* 2 n)))
     ⇒ foo3

(defun three-b (b1 b2 b3)
  "Select three existing buffers.
Put them into three windows, selecting the last one."
    (interactive "bBuffer1:\nbBuffer2:\nbBuffer3:")
    (delete-other-windows)
    (split-window (selected-window) 8)
    (switch-to-buffer b1)
    (other-window 1)
    (split-window (selected-window) 8)
    (switch-to-buffer b2)
    (other-window 1)
    (switch-to-buffer b3))
     ⇒ three-b
(three-b "*scratch*" "declarations.texi" "*mail*")
     ⇒ nil
```

20.3 Interactive Call

After the command loop has translated a key sequence into a command it invokes that command using the function `command-execute`. If the command is a function, `command-execute` calls `call-interactively`, which reads the arguments and calls the command. You can also call these functions yourself.

commandp *object* Function

Returns `t` if *object* is suitable for calling interactively; that is, if *object* is a command. Otherwise, returns `nil`.

The interactively callable objects include strings and vectors (treated as keyboard macros), lambda expressions that contain a top-level call to `interactive`, byte-code function objects made from such lambda expressions, autoload objects that are declared as interactive (non-`nil` fourth argument to `autoload`), and some of the primitive functions.

A symbol satisfies `commandp` if its function definition satisfies `commandp`.

Keys and keymaps are not commands. Rather, they are used to look up commands (see Chapter 21 [Keymaps], page 361).

See `documentation` in Section 23.2 [Accessing Documentation], page 424, for a realistic example of using `commandp`.

call-interactively *command* &optional *record-flag keys* Function

This function calls the interactively callable function *command*, reading arguments according to its interactive calling specifications. 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 *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 20.13 [Command History], page 358.

The argument *keys*, if given, specifies the sequence of events to supply if the command inquires which events were used to invoke it.

command-execute *command* &optional *record-flag keys* Function

This function executes *command*. The argument *command* must satisfy the `commandp` predicate; i.e., it must be an interactively callable function or a keyboard macro.

A string or vector as *command* is executed with `execute-kbd-macro`. A function is passed to `call-interactively`, along with the optional *record-flag*.

A symbol is handled by using its function definition in its place. A symbol with an `autoload` definition counts as a command if it was declared to stand for an interactively callable function. Such a definition is handled by loading the specified library and then rechecking the definition of the symbol.

The argument *keys*, if given, specifies the sequence of events to supply if the command inquires which events were used to invoke it.

execute-extended-command *prefix-argument* Command

This function reads a command name from the minibuffer using `completing-read` (see Section 19.5 [Completion], page 301). 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 1)
---------- Buffer: Minibuffer ----------
1 M-x forward-word RET
---------- Buffer: Minibuffer ----------
     ⇒ t
```

interactive-p Function

This function returns `t` if the containing function (the one whose code includes the call to `interactive-p`) was called interactively, with the function `call-interactively`. (It makes no difference whether `call-interactively` was called from Lisp or directly from the editor command loop.) If the containing function was called by Lisp evaluation (or with `apply` or `funcall`), then it was not called interactively.

The most common use of `interactive-p` is for deciding whether to print an informative message. As a special exception, `interactive-p` returns `nil` whenever a keyboard macro is being run. This is to suppress the informative messages and speed execution of the macro.

For example:

```
(defun foo ()
  (interactive)
  (when (interactive-p)
    (message "foo")))
     ⇒ foo

(defun bar ()
  (interactive)
  (setq foobar (list (foo) (interactive-p))))
     ⇒ bar

;; Type M-x foo.
     ⊣ foo

;; Type M-x bar.
;; This does not print anything.

foobar
     ⇒ (nil t)
```

The other way to do this sort of job is to make the command take an argument `print-message` which should be non-`nil` in an interactive call, and use the `interactive` spec to make sure it is non-`nil`. Here's how:

```
(defun foo (&optional print-message)
  (interactive "p")
  (when print-message
    (message "foo")))
```

The numeric prefix argument, provided by '`p`', is never `nil`.

20.4 Information from the Command Loop

The editor command loop sets several Lisp variables to keep status records for itself and for commands that are run.

last-command Variable

This variable records the name of the previous command executed by the command loop (the one before the current command). Normally the value is a symbol with a function definition, but this is not guaranteed.

The value is copied from `this-command` when a command returns to the command loop, except when the command has specified a prefix argument for the following command.

This variable is always local to the current terminal and cannot be buffer-local. See Section 28.2 [Multiple Displays], page 526.

real-last-command Variable

This variable is set up by Emacs just like `last-command`, but never altered by Lisp programs.

this-command Variable

This variable records the name of the command now being executed by the editor command loop. Like `last-command`, it is normally a symbol with a function definition.

The command loop sets this variable just before running a command, and copies its value into `last-command` when the command finishes (unless the command specified a prefix argument for the following command).

Some commands set this variable during their execution, as a flag for whatever command runs next. In particular, the functions for killing text set `this-command` to `kill-region` so that any kill commands immediately following will know to append the killed text to the previous kill.

If you do not want a particular command to be recognized as the previous command in the case where it got an error, you must code that command to

prevent this. One way is to set `this-command` to `t` at the beginning of the command, and set `this-command` back to its proper value at the end, like this:

```
(defun foo (args...)
  (interactive ...)
  (let ((old-this-command this-command))
    (setq this-command t)
    ...do the work...
    (setq this-command old-this-command)))
```

We do not bind `this-command` with `let` because that would restore the old value in case of error—a feature of `let` which in this case does precisely what we want to avoid.

this-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. The value is a string if all those events were characters. See Section 20.5 [Input Events], page 330.

```
(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 do never need to deal with the complexities of storing input events in a string (see Section 20.5.14 [Strings of Events], page 342).

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 19.6 [Yes-or-No Queries], page 312).

last-command-event Variable
last-command-char Variable

This variable is set to the last input event that was read by the command loop as part of a command. The principal use of this variable is in `self-insert-command`, which uses it to decide which character to insert.

```
last-command-event
;; Now use C-u C-x C-e to evaluate that.
     ⇒ 5
```

The value is 5 because that is the ASCII code for `C-e`.

The alias `last-command-char` exists for compatibility with Emacs version 18.

last-event-frame Variable

This variable records which frame the last input event was directed to. Usually this is the frame that was selected when the event was generated, but if that frame has redirected input focus to another frame, the value is the frame to which the event was redirected. See Section 28.9 [Input Focus], page 538.

20.5 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`.

20.5.1 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 20.5.12 [Classifying Events], page 339.

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. (On some terminals, the key labeled ALT is actually the meta key.)

It is best to avoid mentioning specific bit numbers in your program. To test the modifier bits of a character, use the function `event-modifiers` (see Section 20.5.12 [Classifying Events], page 339). 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 21.9 [Changing Key Bindings], page 374). The function `event-convert-list` converts such a list into an event type (see Section 20.5.12 [Classifying Events], page 339).

20.5.2 Function Keys

Most keyboards also have *function keys*—keys that have names or symbols that are not characters. Function keys are represented in Emacs Lisp as symbols; the symbol's name is the function key's label, in lower case. For example, pressing a key labeled F1 places the symbol `f1` in the input stream.

The event type of a function key event is the event symbol itself. See Section 20.5.12 [Classifying Events], page 339.

Here are a few special cases in the symbol-naming convention for function keys:

`backspace`, `tab`, `newline`, `return`, `delete`

These keys correspond to common ASCII control characters that have special keys on most keyboards.

In ASCII, `C-i` and TAB are the same character. If the terminal can distinguish between them, Emacs conveys the distinction to Lisp programs by representing the former as the integer 9, and the latter as the symbol `tab`.

Most of the time, it's not useful to distinguish the two. So normally `function-key-map` (see Section 37.8.2 [Translating In-

put], page 730) 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.

20.5.3 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 20.5.7 [Repeat Events], page 335). 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 20.2.2 [Interactive Codes], page 322.

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.

20.5.4 Click Events

When the user presses a mouse button and releases it at the same location, that generates a *click* event. Mouse click events have this form:

```
(event-type
 (window buffer-pos (x . y) timestamp)
 click-count)
```

Here is what the elements normally mean:

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`.

window This is the window in which the click occurred.

x, *y* These are the pixel-denominated coordinates of the click, relative to the top left corner of *window*, which is `(0 . 0)`.

buffer-pos This is the buffer position of the character clicked on.

timestamp
: This is the time at which the event occurred, in milliseconds. (Since this value wraps around the entire range of Emacs Lisp integers in about five hours, it is useful only for relating the times of nearby events.)

click-count
: This is the number of rapid repeated presses so far of the same mouse button. See Section 20.5.7 [Repeat Events], page 335.

The meanings of *buffer-pos*, *x* and *y* are somewhat different when the event location is in a special part of the screen, such as the mode line or a scroll bar.

If the location is in a scroll bar, then *buffer-pos* is the symbol `vertical-scroll-bar` or `horizontal-scroll-bar`, and the pair `(`*x* `.` *y*`)` is replaced with a pair `(`*portion* `.` *whole*`)`, where *portion* is the distance of the click from the top or left end of the scroll bar, and *whole* is the length of the entire scroll bar.

If the position is on a mode line or the vertical line separating *window* from its neighbor to the right, then *buffer-pos* is the symbol `mode-line` or `vertical-line`. For the mode line, *y* does not have meaningful data. For the vertical line, *x* does not have meaningful data.

In one special case, *buffer-pos* is a list containing a symbol (one of the symbols listed above) instead of just the symbol. This happens after the imaginary prefix keys for the event are inserted into the input stream. See Section 20.6.1 [Key Sequence Input], page 344.

20.5.5 Drag Events

With Emacs, you can have a drag event without even changing your clothes. A *drag event* happens every time the user presses a mouse button and then moves the mouse to a different character position before releasing the button. Like all mouse events, drag events are represented in Lisp as lists. The lists record both the starting mouse position and the final position, like this:

```
(event-type
 (window1 buffer-pos1 (x1 . y1) timestamp1)
 (window2 buffer-pos2 (x2 . y2) timestamp2)
 click-count)
```

For a drag event, the name of the symbol *event-type* contains the prefix '`drag-`'. For example, dragging the mouse with button 2 held down generates a `drag-mouse-2` event. The second and third elements of the event give the starting and ending position of the drag. Aside from that, the data have the same meanings as in a click event (see Section 20.5.4 [Click Events], page 333). 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.

20.5.6 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.[1] These occur as soon as a button is pressed. They are represented by lists that look exactly like click events (see Section 20.5.4 [Click Events], page 333), 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 20.5.8 [Motion Events], page 337.

20.5.7 Repeat Events

If you press the same mouse button more than once in quick succession without moving the mouse, Emacs generates special *repeat* mouse events for the second and subsequent presses.

The most common repeat events are *double-click* events. Emacs generates a double-click event when you click a button twice; the event happens when you release the button (as is normal for all click events).

The event type of a double-click event contains the prefix '`double-`'. Thus, a double click on the second mouse button with meta held down comes to the Lisp program as `M-double-mouse-2`. If a double-click event has no binding, the binding of the corresponding ordinary click event is used to execute it. Thus, you need not pay attention to the double click feature unless you really want to.

When the user performs a double click, Emacs generates first an ordinary click event, and then a double-click event. Therefore, you must design the command binding of the double click event to assume that the single-click

[1] Button-down is the conservative antithesis of drag.

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-time Variable

To generate repeat events, successive mouse button presses must be at the same screen position, and 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.

20.5.8 Motion Events

Emacs sometimes generates *mouse motion* events to describe motion of the mouse without any button activity. Mouse motion events are represented by lists that look like this:

```
(mouse-movement (window buffer-pos (x . y) timestamp))
```

The second element of the list describes the current position of the mouse, just as in a click event (see Section 20.5.4 [Click Events], page 333).

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 28.13 [Mouse Tracking], page 541.

20.5.9 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 28.9 [Input Focus], page 538.

Focus events are represented in Lisp as lists that look like this:

```
(switch-frame new-frame)
```

where *new-frame* is the frame switched to.

Most X window managers are set up so that just moving the mouse into a window is enough to set the focus there. Emacs appears to do this, because it changes the cursor to solid in the new frame. However, there is no need for the Lisp program to know about the focus change until some other kind of input arrives. So Emacs generates a focus event only when the user actually types a keyboard key or presses a mouse button in the new frame; just moving the mouse between frames does not generate a focus event.

A focus event in the middle of a key sequence would garble the sequence. So Emacs never generates a focus event in the middle of a key sequence. If the user changes focus in the middle of a key sequence—that is, after a prefix key—then Emacs reorders the events so that the focus event comes either before or after the multi-event key sequence, and not within it.

20.5.10 Miscellaneous Window System Events

A few other event types represent occurrences within the window 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.

`(mouse-wheel` *position delta*`)`
: This kind of event is generated by moving a wheel on a mouse (such as the MS Intellimouse). Its effect is typically a kind of scroll or zoom.

  The element *delta* describes the amount and direction of the wheel rotation. Its absolute value is the number of increments by which the wheel was rotated. A negative *delta* indicates that the wheel was rotated backwards, towards the user, and a positive *delta* indicates that the wheel was rotated forward, away from the user.

  The element *position* is a list describing the position of the event, in the same format as used in a mouse-click event.

  This kind of event is generated only on some kinds of systems.

`(drag-n-drop` *position files*`)`
: This kind of event is generated when a group of files is selected in an application outside of Emacs, and then dragged and dropped onto an Emacs frame.

  The element *position* is a list describing the position of the event, in the same format as used in a mouse-click event, and *files* is the list of file names that were dragged and dropped. The usual way to handle this event is by visiting these files.

  This kind of event is generated, at present, only on some kinds of systems.

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.

20.5.11 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))
```

20.5.12 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`.

The argument *event* may be an entire event object, or just an event type.

Here are some examples:

```
(event-modifiers ?a)
     ⇒ nil
```

```
(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. 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. For example,

```
(event-convert-list '(control ?a))
     ⇒ 1
(event-convert-list '(control meta ?a))
     ⇒ -134217727
(event-convert-list '(control super f1))
     ⇒ C-s-f1
```

20.5.13 Accessing Events

This section describes convenient functions for accessing the data in a mouse button or motion event.

These two functions return the starting or ending position of a mouse-button event, as a list of this form:

(*window buffer-position* (*x* . *y*) *timestamp*)

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 five 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-point *position* Function

Return the buffer position in *position*. This is an integer.

posn-x-y *position* Function

Return the pixel-based x and y coordinates in *position*, as a cons cell (*x* . *y*).

posn-col-row *position* Function

Return the row and column (in units of characters) of *position*, as a cons cell (*col* . *row*). These are computed from the *x* and *y* values actually found in *position*.

posn-timestamp *position* Function
Return the timestamp in *position*.

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.

20.5.14 Putting Keyboard Events in Strings

In most of the places where strings are used, we conceptualize the string as containing text characters—the same kind of characters found in buffers or files. Occasionally Lisp programs use strings that conceptually contain keyboard characters; for example, they may be key sequences or keyboard macro definitions. However, storing keyboard characters in a string is a complex matter, for reasons of historical compatibility, and it is not always possible.

We recommend that new programs avoid dealing with these complexities by not storing keyboard events in strings. Here is how to do that:

- Use vectors instead of strings for key sequences, when you plan to use them for anything other than as arguments to `lookup-key` and `define-key`. For example, you can use `read-key-sequence-vector` instead of `read-key-sequence`, and `this-command-keys-vector` instead of `this-command-keys`.
- Use vectors to write key sequence constants containing meta characters, even when passing them directly to `define-key`.
- When you have to look at the contents of a key sequence that might be a string, use `listify-key-sequence` (see Section 20.6.4 [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.

20.6 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 38.7 [Temporary Displays], page 746, and `sit-`

`for` in Section 20.8 [Waiting], page 350. See Section 37.8 [Terminal Input], page 729, for functions and variables for controlling terminal input modes and debugging terminal input. See Section 37.8.2 [Translating Input], page 730, for features you can use for translating or modifying input events while reading them.

For higher-level input facilities, see Chapter 19 [Minibuffers], page 295.

20.6.1 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* 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.

If the events are all characters and all can fit in a string, then `read-key-sequence` returns a string (see Section 20.5.14 [Strings of Events], page 342). 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.

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.

In the example below, the prompt '`?`' is displayed in the echo area, and 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 20.9 [Quitting], page 351.

read-key-sequence-vector *prompt* Function

This is like `read-key-sequence` except that it always returns the key sequence as a vector, never as a string. See Section 20.5.14 [Strings of Events], page 342.

If an input character is an upper-case letter and has no key binding, but its lower-case equivalent has one, then `read-key-sequence` converts

the character to lower case. Note that `lookup-key` does not perform case conversion in this way.

The function `read-key-sequence` also transforms some mouse events. It converts unbound drag events into click events, and discards unbound button-down events entirely. It also reshuffles focus events and miscellaneous window events so that they never appear in a key sequence with any other events.

When mouse events occur in special parts of a window, such as a mode line or a scroll bar, the event type shows nothing special—it is the same symbol that would normally represent that combination of mouse button and modifier keys. The information about the window part is kept elsewhere in the event—in the coordinates. But `read-key-sequence` translates this information into imaginary "prefix keys", all of which are symbols: `mode-line`, `vertical-line`, `horizontal-scroll-bar` 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.

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.

20.6.2 Reading One Event

The lowest level functions for command input are those that read a single event.

read-event &optional *prompt suppress-input-method* 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 *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 38.3 [The Echo Area], page 740.

If *suppress-input-method* is non-`nil`, then the current input method is disabled for reading this event. If you want to read an event without input-method processing, always do it this way; don't try binding `input-method-function` (see below).

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 `read-event` gets an event that is defined as a help character, in some cases `read-event` processes the event directly without returning. See Section 23.5 [Help Functions], page 429. Certain other events, called *special events*, are also processed directly within `read-event` (see Section 20.7 [Special Events], page 349).

Here is what happens if you call `read-event` and then press the right-arrow function key:

```
(read-event)
     ⇒ right
```

read-char Function

This function reads and returns a character of command input. It discards any events that are not characters, until it gets a character.

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-event` also invokes the current input method, if any. 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 event as an argument.

input-method-function Variable

If this is non-`nil`, its value specifies the current input method function.

Note: 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`. 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 event of a key sequence. Thus, these characters are not subject to input method processing. It is usually a good idea for the input method processing to 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.

20.6.3 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.

Quitting is suppressed when the first character is read, so that the user can enter a `C-g`. See Section 20.9 [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-177
---------- Echo Area ----------

     ⇒ 127
```

20.6.4 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 19.8 [Reading a Password], page 315).

unread-command-events Variable

This variable holds a list of events waiting to be read as command input. The events are used in the order they appear in the list, and removed one by one as they are used.

The variable is needed because in some cases a function reads an event and then decides not to use it. Storing the event in this variable causes it to be processed normally, by the command loop or by the functions to read command input.

For example, the function that implements numeric prefix arguments reads any number of digits. When it finds a non-digit event, it must unread the event so that it can be read normally by the command loop. Likewise, incremental search uses this feature to unread events with no special meaning in a search, because these events should exit the search and then execute normally.

The reliable and easy way to extract events from a key sequence so as to put them in `unread-command-events` is to use `listify-key-sequence` (see Section 20.5.14 [Strings of Events], page 342).

Normally you add events to the front of this list, so that the events most recently unread will be reread first.

listify-key-sequence *key* Function

This function converts the string or vector *key* to a list of individual events, which you can put in `unread-command-events`.

unread-command-char Variable

This variable holds a character to be read as command input. A value of -1 means "empty".

This variable is mostly obsolete now that you can use `unread-command-events` instead; it exists only to support programs written for Emacs versions 18 and earlier.

input-pending-p Function

This function determines whether any command input is currently available to be read. It returns immediately, with value `t` if there is available input, `nil` otherwise. On rare occasions it may return `t` when no input is available.

last-input-event Variable
last-input-char Variable

This variable records the last terminal input event read, whether as part of a command or explicitly by a Lisp program.

In the example below, the Lisp program reads the character *1*, ASCII code 49. It becomes the value of `last-input-event`, while *C-e* (we assume *C-x C-e* command is used to evaluate this expression) remains the value of `last-command-event`.

```
(progn (print (read-char))
       (print last-command-event)
       last-input-event)
     ⊣ 49
     ⊣ 5
     ⇒ 49
```

The alias `last-input-char` exists for compatibility with Emacs version 18.

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
```

20.7 Special Events

Special events are handled at a very low level—as soon as they are read. The `read-event` function processes these events itself, and never returns them.

Events that are handled in this way do not echo, they are never grouped into key sequences, and they never appear in the value of `last-command-event` or `(this-command-keys)`. They do not discard a numeric argument, they cannot be unread with `unread-command-events`, they may not appear

in a keyboard macro, and they are not recorded in a keyboard macro while you are defining one.

These events do, however, appear in `last-input-event` immediately after they are read, and this is the way for the event's definition to find the actual event.

The events types `iconify-frame`, `make-frame-visible` and `delete-frame` 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 21.6 [Active Keymaps], page 367).

20.8 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 *millisec 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 value is `t` if `sit-for` waited the full time with no input arriving (see `input-pending-p` in Section 20.6.4 [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 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*.

Redisplay is always preempted if input arrives, and does not happen at all if input is available before it starts. Thus, there is no way to force screen updating if there is pending input; however, if there is no input pending, you can force an update with no delay by using `(sit-for 0)`.

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).

Iconifying or deiconifying a frame makes `sit-for` return, because that generates an event. See Section 20.5.10 [Misc Events], page 337.

The usual purpose of `sit-for` is to give the user time to read text that you display.

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 37.5 [Time of Day], page 723, for functions to get the current time.

20.9 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.

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 9.5.3 [Errors], page 138.)

You can specify a character other than `C-g` to use for quitting. See the function `set-input-mode` in Section 37.8 [Terminal Input], page 729.

20.10 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` declaration. (See Section 20.2.1 [Using Interactive], page 320.) 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.

20.11 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 9.5.1 [Catch and Throw], page 135). 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 Function

This function invokes the editor command loop. It is called automatically by the initialization of Emacs, to let the user begin editing. When called from a Lisp program, it enters a recursive editing level.

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 20.9 [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.

20.12 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 '`.emacs`' file with Lisp expressions such as this:

```
(put 'upcase-region 'disabled t)
```

For a few commands, these properties are present by default and may be removed by the '`.emacs`' file.

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* to be executed without special confirmation from now on, and (if the user confirms) alter the user's '`.emacs`' file so that this will apply to future sessions.

disable-command *command* Command
Require special confirmation to execute *command* from now on, and (if the user confirms) alter the user's '`.emacs`' file so that this will apply to future sessions.

disabled-command-hook Variable
When the user invokes a disabled command interactively, this normal hook is run instead of the disabled command. The hook functions can use `this-command-keys` to determine what the user typed to run the

command, and thus find the command itself. See Section 22.6 [Hooks], page 420.

By default, `disabled-command-hook` contains a function that asks the user whether to proceed.

20.13 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 (specified by the variable `history-length`), 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 19.4 [Minibuffer History], page 300), 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.

20.14 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 12 [Macros], page 189).

execute-kbd-macro *kbdmacro* &optional *count* 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.

See Section 20.6.2 [Reading One Event], page 345, for an example of using `execute-kbd-macro`.

executing-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 indicates whether a keyboard macro is being defined. A command can test this variable so as to behave differently while a macro is being defined. The commands `start-kbd-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 28.2 [Multiple Displays], page 526.

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 28.2 [Multiple Displays], page 526.

21 Keymaps

The bindings between input events and commands are recorded in data structures called *keymaps*. Each binding 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*.

21.1 Keymap Terminology

A *keymap* is a table mapping event types to definitions (which can be any Lisp objects, though only certain types are meaningful for execution by the command loop). Given an event (or an event type) and a keymap, Emacs can get the event's definition. Events include characters, function keys, and mouse actions (see Section 20.5 [Input Events], page 330).

A sequence of input events that form a unit is called a *key sequence*, or *key* for short. A sequence of one event is always a key sequence, and so are some multi-event sequences.

A keymap determines a binding or definition for any key sequence. If the key sequence is a single event, its binding is the definition of the event in the keymap. The binding of a key sequence of more than one event is found by an iterative process: 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.

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 21.5 [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 21.6 [Active Keymaps], page 367, for details.

21.2 Format of Keymaps

A keymap is a list whose CAR is the symbol `keymap`. The remaining elements of the list define the key bindings of the 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 20.5.12 [Classifying Events], page 339.

(`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.

vector
: If an element of a keymap is a vector, the vector counts as bindings for all the ASCII characters, codes 0 through 127; vector 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 vector is called a *full keymap*. Other keymaps are called *sparse keymaps*.

  When a keymap contains a vector, it always defines a binding for each ASCII character, even if the vector contains `nil` for that character. Such a binding of `nil` overrides any default key binding in the keymap, for ASCII characters. However, default bindings are still meaningful for events other than ASCII characters. 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.

string
: Aside from bindings, 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 21.12 [Menu Keymaps], page 381.

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 really represented as ESC `a`, and its global binding is found at the slot for `a` in `esc-map` (see Section 21.5 [Prefix Keys], page 365).

Here as an example is the local keymap for Lisp mode, a sparse keymap. It defines bindings for DEL and TAB, plus *C-c C-l*, *M-C-q*, and *M-C-x*.

```
lisp-mode-map
⇒
(keymap
 ;; TAB
 (9 . lisp-indent-line)
 ;; DEL
 (127 . backward-delete-char-untabify)
 (3 keymap
    ;; C-c C-l
    (12 . run-lisp))
 (27 keymap
     ;; M-C-q, treated as ESC C-q
     (17 . indent-sexp)
     ;; M-C-x, treated as ESC C-x
     (24 . lisp-send-defun)))
```

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`.

```
(keymapp '(keymap))
    ⇒ t
(keymapp (current-global-map))
    ⇒ t
```

21.3 Creating Keymaps

Here we describe the functions for creating keymaps.

make-keymap &optional *prompt* Function

This function creates and returns a new full keymap (i.e., one containing a vector of length 128 for defining all the ASCII characters). The new keymap initially binds all ASCII characters to `nil`, and does not bind any other kind of event.

```
(make-keymap)
    ⇒ (keymap [nil nil nil ... nil nil])
```

If you specify *prompt*, that becomes the overall prompt string for the keymap. The prompt string is useful for menu keymaps (see Section 21.12 [Menu Keymaps], page 381).

make-sparse-keymap &optional *prompt* Function

This function creates and returns a new sparse keymap with no entries. The new keymap does not bind any events. The argument *prompt* specifies a prompt string, as in `make-keymap`.

```
(make-sparse-keymap)
    ⇒ (keymap)
```

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
```

21.4 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 bindings... . 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 *bindings*.

If you change the bindings in *parent-keymap* using `define-key` or other key-binding functions, these changes are visible in the inheriting keymap unless shadowed by *bindings*. The converse is not true: if you use `define-key` to change the inheriting keymap, that affects *bindings*, but has 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)
```

21.5 Prefix Keys

A *prefix key* is a key sequence whose binding is a keymap. The keymap defines what to do with key sequences that extend the prefix key. For example, `C-x` is a prefix key, and it uses a keymap that is also stored in the variable `ctl-x-map`. This keymap defines bindings for key sequences starting with `C-x`.

Some of the standard Emacs prefix keys use keymaps that are also found in Lisp variables:

- `esc-map` is the global keymap for the ESC prefix key. Thus, the global definitions of all meta characters are actually found here. This map is also the function definition of `ESC-prefix`.
- `help-map` is the global keymap for the `C-h` prefix key.
- `mode-specific-map` is the global keymap for the prefix key `C-c`. This map is actually global, not mode-specific, but its name provides useful information about `C-c` in the output of `C-h b` (`display-bindings`), since the main use of this prefix key is for mode-specific bindings.
- `ctl-x-map` is the global keymap used for the `C-x` prefix key. This map is found via the function cell of the symbol `Control-X-prefix`.
- `mule-keymap` is the global keymap used for the `C-x` RET prefix key.
- `ctl-x-4-map` is the global keymap used for the `C-x 4` prefix key.
- `ctl-x-5-map` is the global keymap used for the `C-x 5` prefix key.
- `2C-mode-map` is the global keymap used for the `C-x 6` prefix key.

- `vc-prefix-map` is the global keymap used for the *C-x v* prefix key.
- `facemenu-keymap` is the global keymap used for the *M-g* 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 21.6 [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* Function

This function prepares *symbol* for use as a prefix key's binding: it creates a full 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.

This function also sets *symbol* as a variable, with the keymap as its value. It returns *symbol*.

21.6 Active Keymaps

Emacs normally contains many keymaps; at any given time, just a few of them are *active* in that they participate in the interpretation of user input. These are the global keymap, the current buffer's local keymap, and the keymaps of any enabled minor modes.

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. Text properties can specify an alternative local map for certain parts of the buffer; see Section 31.19.4 [Special Properties], page 615.

Each minor mode can have a keymap; if it does, the keymap is active when the minor mode is enabled.

The variable `overriding-local-map`, if non-`nil`, specifies another local keymap that overrides the buffer's local map and all the minor mode keymaps.

All the active keymaps are used together to determine what command to execute when a key is entered. Emacs searches these maps one by one, in order of decreasing precedence, until it finds a binding in one of the maps. The procedure for searching a single keymap is called *key lookup*; see Section 21.7 [Key Lookup], page 370.

Normally, Emacs first searches for the key in the minor mode maps, in the order specified by `minor-mode-map-alist`; if they do not supply a binding for the key, Emacs searches the local map; if that too has no binding, Emacs then searches the global map. However, if `overriding-local-map` is non-`nil`, Emacs searches that map first, before the global map.

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 maps 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 19.1 [Intro to Minibuffers], page 295.

Emacs has other keymaps that are used in a different way—translating events within `read-key-sequence`. See Section 37.8.2 [Translating Input], page 730.

See Appendix E [Standard Keymaps], page 817, for a list of standard 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 map, but you should not assign this variable any value other than the keymap it starts out with.

current-global-map Function

This function returns the current global keymap. This is the same as the value of `global-map` unless you change one or the other.

```
(current-global-map)
⇒ (keymap [set-mark-command beginning-of-line ...
           delete-backward-char])
```

current-local-map Function

This function returns the current buffer's local keymap, or `nil` if it has none. In the following example, the keymap for the '`*scratch*`' buffer (using Lisp Interaction mode) is a sparse keymap in which the entry for ESC, ASCII code 27, is another sparse keymap.

```
(current-local-map)
⇒ (keymap
    (10 . eval-print-last-sexp)
    (9 . lisp-indent-line)
    (127 . backward-delete-char-untabify)
    (27 keymap
        (24 . eval-defun)
        (17 . indent-sexp)))
```

current-minor-mode-maps Function

This function returns a list of the keymaps of currently enabled minor modes.

use-global-map *keymap* Function

This function makes *keymap* the new current global keymap. It returns `nil`.

It is very unusual to change the global keymap.

use-local-map *keymap* Function

This function makes *keymap* the new local keymap of the current buffer. If *keymap* is `nil`, then the buffer has no local keymap. `use-local-map` returns `nil`. Most major mode commands use this function.

minor-mode-map-alist Variable

This variable is an alist describing keymaps that may or may not be active according to the values of certain variables. Its elements look like this:

(*variable* . *keymap*)

The keymap *keymap* is active whenever *variable* has a non-`nil` value. Typically *variable* is the variable that enables or disables a minor mode. See Section 22.2.2 [Keymaps and Minor Modes], page 404.

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 CADR will not do. The CADR 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, their order of priority is the order of `minor-mode-map-alist`. 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 22.2.2 [Keymaps and Minor Modes], page 404, for more information about minor modes. See also `minor-mode-key-binding` (see Section 21.8 [Functions for Key Lookup], page 372).

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 and instead of all the minor mode keymaps. This keymap, if any, 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 and all the minor mode keymaps.

This variable is always local to the current terminal and cannot be buffer-local. See Section 28.2 [Multiple Displays], page 526. 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 20.7 [Special Events], page 349.

21.7 Key Lookup

Key lookup is the process of finding the binding of a key sequence from a given keymap. Actual execution 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 mouse events with just their types (symbols) instead of with entire mouse events (lists). See Section 20.5 [Input Events], page 330. 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 menu key bindings, 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 kinds 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 11.1 [What Is a Function], page 171.

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 20.14 [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 the types of the elements of the list.

- 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 command, and is treated as such (see above).
- 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 20.3 [Interactive Call], page 325).

The symbol `undefined` is worth special mention: it means to treat the key as undefined. Strictly speaking, the key is defined, and its binding is the command `undefined`; but that command does the same thing that is done automatically for an undefined key: it rings the bell (by calling `ding`) but does not signal an error.

`undefined` is used in local keymaps to override a global key binding and make the key "undefined" locally. A local binding of `nil` would fail to do this because it would not override the global binding.

anything else
: If any other type of object is found, the events used so far in the lookup form a complete key, and the object is its binding, but the binding is not executable as a command.

In short, a keymap entry may be a keymap, a command, a keyboard macro, a symbol that leads to one of them, or an indirection or `nil`. Here is an example of a sparse keymap with two characters bound to commands and one bound to another keymap. This map is the normal value of `emacs-lisp-mode-map`. Note that 9 is the code for TAB, 127 for DEL, 27 for ESC, 17 for `C-q` and 24 for `C-x`.

```
(keymap (9 . lisp-indent-line)
        (127 . backward-delete-char-untabify)
        (27 keymap (17 . indent-sexp) (24 . eval-defun)))
```

21.8 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) "\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 21.2 [Format of Keymaps], page 362.)

If *key* contains a meta character, 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 20.6.1 [Key Sequence Input], page 344). 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.

key-binding *key* &optional *accept-defaults* Function

This function returns the binding for *key* in the current keymaps, trying all the 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` (above).

An error is signaled if *key* is not a string or a vector.

```
(key-binding "\C-x\C-f")
    ⇒ find-file
```

local-key-binding *key* &optional *accept-defaults* Function

This function returns the binding for *key* in the current local keymap, or `nil` if it is undefined there.

The argument *accept-defaults* controls checking for default bindings, as in `lookup-key` (above).

global-key-binding *key* &optional *accept-defaults* Function

This function returns the binding for command *key* in the current global keymap, or `nil` if it is undefined there.

The argument *accept-defaults* controls checking for default bindings, as in `lookup-key` (above).

minor-mode-key-binding *key* &optional *accept-defaults* Function

This function returns a list of all the active minor mode bindings of *key*. More precisely, it returns an alist of pairs (*modename* . *binding*), where

modename is the variable that enables the minor mode, and *binding* is *key*'s binding in that mode. If *key* has no minor-mode bindings, the value is `nil`.

If the first binding found is not a prefix definition (a keymap or a symbol defined as a keymap), all subsequent bindings from other minor modes are omitted, since they would be completely shadowed. Similarly, the list omits non-prefix bindings that follow prefix bindings.

The argument *accept-defaults* controls checking for default bindings, as in `lookup-key` (above).

meta-prefix-char Variable

This variable is the meta-prefix character code. It is used when 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 21.5 [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 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. Here is an illustration:

```
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!
```

21.9 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 21.10

[Key Binding Commands], page 378). You can also use `define-key`, a more general function; then you must specify explicitly the map to change.

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 27). 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 19.

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.

For the functions below, an error is signaled 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.

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 21.7 [Key Lookup], page 370.) The value returned by `define-key` is *binding*.

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 in most cases, but it does matter for menu keymaps (see Section 21.12 [Menu Keymaps], page 381).

Here is an example that 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 "\C-xf" '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 "\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 "\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.

substitute-key-definition *olddef newdef keymap* Function
&optional *oldmap*

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`, then its bindings 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 making all the printing characters undefined. More precisely, it binds them to the command `undefined`. This makes 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 26.7 [Read Only Buffers], page 487).

Since this function modifies *keymap*, you would normally use it on a newly created keymap. Operating on an existing keymap that is used for some other purpose is likely to cause trouble; for example, suppressing `global-map` would make it impossible to use most of Emacs.

Most often, `suppress-keymap` is used to initialize local keymaps of modes such as Rmail and Dired where insertion of text is not desirable and the buffer is read-only. Here is an example taken from the file '`emacs/lisp/dired.el`', showing how the local keymap for Dired mode is set up:

```
(setq dired-mode-map (make-keymap))
(suppress-keymap dired-mode-map)
(define-key dired-mode-map "r" 'dired-rename-file)
(define-key dired-mode-map "\C-d" 'dired-flag-file-deleted)
(define-key dired-mode-map "d" 'dired-flag-file-deleted)
(define-key dired-mode-map "v" 'dired-view-file)
(define-key dired-mode-map "e" 'dired-find-file)
(define-key dired-mode-map "f" 'dired-find-file)
...
```

21.10 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 '`.emacs`' file for simple customization. For example,

```
(global-set-key "\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, typed with the Meta key, to set point where you click.

global-set-key *key definition* Command

This function sets the binding of *key* in the current global map to *definition*.

```
(global-set-key key definition)

(define-key (current-global-map) key definition)
```

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 definition* Command

This function sets the binding of *key* in the current local keymap to *definition*.

```
(local-set-key key definition)

(define-key (current-local-map) key definition)
```

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)
```

21.11 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 $\overline{\text{ESC}}$, 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.

where-is-internal *command* &optional *keymap firstonly noindirect* 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 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 non-`nil`, then the maps searched are *keymap* and the global keymap.

Usually it's best to use `overriding-local-map` as the expression for *keymap*. Then `where-is-internal` searches precisely the keymaps that are active. To search only the global map, pass `(keymap)` (an empty keymap) as *keymap*.

If *firstonly* is `non-ascii`, then the value is a single string 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.

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.

```
(where-is-internal 'describe-function)
      ("\^hf" "\^hd")
```

describe-bindings &optional *prefix* 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`.

21.12 Menu Keymaps

A keymap can define a menu as well as bindings for keyboard keys and mouse button. Menus are usually actuated with the mouse, but they can work with the keyboard also.

21.12.1 Defining Menus

A keymap is suitable for menu use if it has an *overall prompt string*, which is a string that appears as an element of the keymap. (See Section 21.2 [Format of Keymaps], page 362.) The string should describe the purpose of the menu. The easiest way to construct a keymap with a prompt string is to specify the string as an argument when you call `make-keymap` or `make-sparse-keymap` (see Section 21.3 [Creating Keymaps], page 363).

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 21.12.6 [Modifying Menus], page 388).

21.12.1.1 Simple Menu Items

The simpler and older way to define a menu keymap binding looks like this:

(*item-string* . *real-binding*)

The CAR, *item-string*, is the string to be displayed in the menu. It should be short—preferably one to three words. It should describe the action of the command it corresponds to.

You can also supply a second string, called the help string, as follows:

(*item-string help-string* . *real-binding*)

Currently Emacs does not actually use *help-string*; it knows only how to ignore *help-string* in order to extract *real-binding*. In the future we may

use *help-string* as extended documentation for the menu item, available on request.

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 22.3 [Mode Line Format], page 405).

You've probably noticed that menu items show the equivalent keyboard key sequence (if any) to invoke the same command. To save time on recalculation, menu display caches this information in a sublist in the binding, like this:

(*item-string* [*help-string*] (*key-binding-data*) . *real-binding*)

Don't put these sublists in the menu item yourself; menu display calculates them automatically. Don't mention keyboard equivalents in the item strings themselves, since that is redundant.

21.12.1.2 Extended Menu Items

An extended-format menu item is a more flexible and also cleaner alternative to the simple format. It consists of a list that starts with the symbol `menu-item`. To define a non-selectable string, the item looks like this:

(`menu-item` *item-name*)

where a string consisting of two or more dashes specifies a separator line.

To define a real menu item which can be selected, the extended format item 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).

`: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*, is the extra help string (not currently used by Emacs).

`:button` (*type* `.` *selected*)
: This property provides a way to define radio buttons and toggle buttons. The CAR, *type*, says which: is 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.

21.12.1.3 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`.

21.12.2 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 28.15 [Pop-Up Menus], page 542.)

If the prefix key ends with a mouse event, Emacs handles the menu keymap by popping up a visible menu, so that the user can select a choice with the mouse. When the user clicks on a menu item, the event generated is whatever character or symbol has the binding that brought about that menu item. (A menu item may generate a series of events if the menu has multiple levels or comes from the menu bar.)

It's often best to use a button-down event to trigger the menu. Then the user can select a menu item by releasing the button.

A single keymap can appear as multiple menu panes, if you explicitly arrange for this. The way to do this is to make a keymap for each pane, then create a binding for each of those maps in the main keymap of the menu. Give each of these bindings an item string that starts with '`@`'. The rest of the item string becomes the name of the pane. See the file '`lisp/mouse.el`' for an example of this. Any ordinary bindings with '`@`'-less item strings are grouped into one pane, which appears along with the other panes explicitly created for the submaps.

X toolkit menus don't have panes; instead, they can have submenus. Every nested keymap becomes a submenu, whether the item string starts with '`@`' or not. In a toolkit version of Emacs, the only thing special about '`@`' at the beginning of an item string is that the '`@`' doesn't appear in the menu item.

You can also produce multiple panes or submenus from separate keymaps. The full definition of a prefix key always comes from merging the definitions supplied by the various active keymaps (minor mode, local, and global). When more than one of these keymaps is a menu, each of them makes a separate pane or panes (when Emacs does not use an X-toolkit) or a separate submenu (when using an X-toolkit). See Section 21.6 [Active Keymaps], page 367.

21.12.3 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 user can use the keyboard to choose a menu item.

Emacs displays the menu alternatives (the item strings of the bindings) in the echo area. If they don't all fit at once, the user can type SPC to see the next line of alternatives. Successive uses of SPC eventually get to the end of the menu and then cycle around to the beginning. (The variable `menu-prompt-more-char` specifies which character is used for this; SPC is the default.)

When the user has found the desired alternative from the menu, he or she should type the corresponding character—the one whose binding is that alternative.

This way of using menus in an Emacs-like editor was inspired by the Hierarkey system.

menu-prompt-more-char Variable

This variable specifies the character to use to ask to see the next line of a menu. Its initial value is 32, the code for SPC.

21.12.4 Menu Example

Here is a complete example of defining a menu keymap. It is the definition of the 'Print' submenu in the 'Tools' menu in the menu bar, and it uses the simple menu item format (see Section 21.12.1.1 [Simple Menu Items], page 381). First we create the keymap, and give it a name:

```
(defvar menu-bar-print-menu (make-sparse-keymap "Print"))
```

Next we define the menu items:

```
(define-key menu-bar-print-menu [ps-print-region]
  '("Postscript Print Region" . ps-print-region-with-faces))
(define-key menu-bar-print-menu [ps-print-buffer]
  '("Postscript Print Buffer" . ps-print-buffer-with-faces))
(define-key menu-bar-print-menu [separator-ps-print]
  '("--"))
(define-key menu-bar-print-menu [print-region]
  '("Print Region" . print-region))
(define-key menu-bar-print-menu [print-buffer]
  '("Print Buffer" . print-buffer))
```

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 binding whose definition is `("--")` is a separator line. Like a real menu item, the separator has a key symbol, in this case `separator-ps-print`. If one menu has two separators, they must have two different key symbols.

Here is code to define enable conditions for two of the commands in the menu:

```
(put 'print-region 'menu-enable 'mark-active)
(put 'ps-print-region-with-faces 'menu-enable 'mark-active)
```

Here is how we make this menu appear as an item in the parent menu:

```
(define-key menu-bar-tools-menu [print]
  (cons "Print" menu-bar-print-menu))
```

Note that this incorporates the submenu keymap, which is the value of the variable `menu-bar-print-menu`, rather than the symbol `menu-bar-print-menu` itself. Using that symbol in the parent menu item would be meaningless because `menu-bar-print-menu` is not a command.

If you wanted to attach the same print menu to a mouse click, you can do it this way:

```
(define-key global-map [C-S-down-mouse-1]
   menu-bar-print-menu)
```

We could equally well use an extended menu item (see Section 21.12.1.2 [Extended Menu Items], page 382) for `print-region`, like this:

```
(define-key menu-bar-print-menu [print-region]
  '(menu-item "Print Region" print-region
              :enable (mark-active)))
```

With the extended menu item, the enable condition is specified inside the menu item itself. If we wanted to make this item disappear from the menu entirely when the mark is inactive, we could do it this way:

```
(define-key menu-bar-print-menu [print-region]
  '(menu-item "Print Region" print-region
              :visible (mark-active)))
```

21.12.5 The Menu Bar

Most window systems allow each frame to have a *menu bar*—a permanently displayed menu stretching horizontally across the top of the frame. The items of the menu bar are the subcommands of the fake "function key" `menu-bar`, as defined by all the active keymaps.

To add an item to the menu bar, invent a fake "function key" of your own (let's call it *key*), and make a binding for the key sequence `[menu-bar` *key*`]`. Most often, the binding is a menu keymap, so that pressing a button on the menu bar item leads to another menu.

When more than one active keymap defines the same fake function key for the menu bar, the item appears just once. If the user clicks on that menu bar item, it brings up a single, combined menu containing all the subcommands of that item—the global subcommands, the local subcommands, and the minor mode subcommands.

The variable `overriding-local-map` is normally ignored when determining the menu bar contents. That is, the menu bar is computed from the keymaps that would be active if `overriding-local-map` were `nil`. See Section 21.6 [Active Keymaps], page 367.

In order for a frame to display a menu bar, its `menu-bar-lines` parameter must be greater than zero. Emacs uses just one line for the menu bar itself; if you specify more than one line, the other lines serve to separate the menu

bar from the windows in the frame. We recommend 1 or 2 as the value of `menu-bar-lines`. See Section 28.3.3 [Window Frame Parameters], page 528.

Here's an example of setting up a menu bar item:

```
(modify-frame-parameters (selected-frame)
                         '((menu-bar-lines . 2)))

;; Make a menu keymap (with a prompt string)
;; and make it the menu bar item's definition.
(define-key global-map [menu-bar words]
  (cons "Words" (make-sparse-keymap "Words")))

;; Define specific subcommands in this menu.
(define-key global-map
  [menu-bar words forward]
  '("Forward word" . forward-word))
(define-key global-map
  [menu-bar words backward]
  '("Backward word" . backward-word))
```

A local keymap can cancel a menu bar item made by the global keymap by rebinding the same fake function key with `undefined` as the binding. For example, this is how Dired suppresses the '`Edit`' menu bar item:

```
(define-key dired-mode-map [menu-bar edit] 'undefined)
```

`edit` is the fake function key used by the global map for the '`Edit`' menu bar item. The main reason to suppress a global menu bar item is to regain space for mode-specific items.

menu-bar-final-items Variable

Normally the menu bar shows global items followed by items defined by the local maps.

This variable holds a list of fake function keys for items to display at the end of the menu bar rather than in normal sequence. The default value is `(help-menu)`; thus, the '`Help`' menu item normally appears at the end of the menu bar, following local menu items.

menu-bar-update-hook Variable

This normal hook is run whenever the user clicks on the menu bar, before displaying a submenu. You can use it to update submenus whose contents should vary.

21.12.6 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 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`, then the new binding goes last, at the end of the 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)
```

22 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 21 [Keymaps], page 361, and Chapter 34 [Syntax Tables], page 669.

22.1 Major Modes

Major modes specialize Emacs for editing particular kinds of text. Each buffer has only one major mode at a time.

The least specialized major mode is called *Fundamental mode*. This mode has no mode-specific definitions or variable settings, so each Emacs command behaves in its default manner, and each option is in its default state. All other major modes redefine various keys and options. For example, Lisp Interaction mode provides special key bindings for `C-j` (`eval-print-last-sexp`), TAB (`lisp-indent-line`), and other keys.

When you need to write several editing commands to help you perform a specialized editing task, creating a new major mode is usually a good idea. In practice, writing a major mode is easy (in contrast to writing a minor mode, which is often difficult).

If the new mode is similar to an old one, it is often unwise to modify the old one to serve two purposes, since it may become harder to use and maintain. Instead, copy and rename an existing major mode definition and alter the copy—or define a *derived mode* (see Section 22.1.5 [Derived Modes], page 401). For example, Rmail Edit mode, which is in '`emacs/lisp/rmailedit.el`', is a major mode that is very similar to Text mode except that it provides three additional commands. Its definition is distinct from that of Text mode, but was derived from it.

Rmail Edit mode offers an example of changing the major mode temporarily for a buffer, so it can be edited in a different way (with ordinary Emacs commands rather than Rmail commands). In such cases, the temporary major mode usually has a command to switch back to the buffer's usual mode (Rmail mode, in this case). You might be tempted to present the temporary redefinitions inside a recursive edit and restore the usual ones when the user exits; but this is a bad idea because it constrains the user's options when it is done in more than one buffer: recursive edits must be exited most-recently-entered first. Using an alternative major mode avoids this limitation. See Section 20.11 [Recursive Editing], page 355.

The standard GNU Emacs Lisp library directory contains the code for several major modes, in files such as '`text-mode.el`', '`texinfo.el`',

'`lisp-mode.el`', '`c-mode.el`', and '`rmail.el`'. You can study these libraries to see how modes are written. Text mode is perhaps the simplest major mode aside from Fundamental mode. Rmail mode is a complicated and specialized mode.

22.1.1 Major Mode Conventions

The code for existing major modes follows various coding conventions, including conventions for local keymap and syntax table initialization, global names, and hooks. Please follow these conventions when you define a new major mode:

- Define a command whose name ends in '`-mode`', with no arguments, that switches to the new mode in the current buffer. This command should set up the keymap, syntax table, and buffer-local variables in an existing buffer, without changing the buffer's contents.
- Write a documentation string for this command that describes the special commands available in this mode. *C-h m* (`describe-mode`) in your mode will display this string.

  The documentation string may include the special documentation substrings, '`\[`*command*`]`', '`\{`*keymap*`}`', and '`\<`*keymap*`>`', that enable the documentation to adapt automatically to the user's own key bindings. See Section 23.3 [Keys in Documentation], page 427.
- The major mode command should start by calling `kill-all-local-variables`. This is what gets rid of the buffer-local variables of the major mode previously in effect.
- The major mode command should set the variable `major-mode` to the major mode command symbol. This is how `describe-mode` discovers which documentation to print.
- The major mode command should set the variable `mode-name` to the "pretty" name of the mode, as a string. This string appears in the mode line.
- Since all global names are in the same name space, all the global variables, constants, and functions that are part of the mode should have names that start with the major mode name (or with an abbreviation of it if the name is long). See Section A.1 [Coding Conventions], page 781.
- 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 21.6 [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 10.6 [Tips for Defining], page 154, 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.

  It is reasonable for a major mode to rebind a key sequence with a standard meaning, if it implements a command that does "the same job" in a way that fits the major mode better. 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.

  Major modes such as Dired or Rmail that do not allow self-insertion of text can reasonably redefine letters and other printing characters as editing commands. Dired and Rmail both do this.
- 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 34 [Syntax Tables], page 669.
- 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`. See Section 35.2 [Abbrev Tables], page 683.
- 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 22.5 [Font Lock Mode], page 414).
- 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` or `imenu-create-index-function` (see Section 22.4 [Imenu], page 412).
- Use `defvar` or `defcustom` to set mode-related variables, so that they are not reinitialized if they already have a value. (Such reinitialization could discard customizations made by the user.)
- To make a buffer-local binding for an Emacs customization variable, use `make-local-variable` in the major mode command, not `make-variable-buffer-local`. The latter function would make the variable local to every buffer in which it is subsequently set, which would affect buffers that do not use this mode. It is undesirable for a mode to have such global effects. See Section 10.10 [Buffer-Local Variables], page 161.

  It's OK to use `make-variable-buffer-local`, if you wish, for a variable used only within a single Lisp package.
- Each major mode should have a *mode hook* named *modename*`-mode-hook`. The major mode command should run that hook, with `run-hooks`, as the very last thing it does. See Section 22.6 [Hooks], page 420.

- The major mode command may also run the hooks of some more basic modes. For example, `indented-text-mode` runs `text-mode-hook` as well as `indented-text-mode-hook`. It may run these other hooks immediately before the mode's own hook (that is, after everything else), or it may run them earlier.
- 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 10.10.2 [Creating Buffer-Local], page 163).
- If this mode is appropriate only for specially-prepared text, then the major mode command symbol should have a property named `mode-class` with value `special`, put on as follows:

  ```
  (put 'funny-mode 'mode-class 'special)
  ```

  This tells Emacs that new buffers created while the current buffer has Funny mode should not inherit Funny mode. Modes such as Dired, Rmail, and Buffer List use this feature.
- If you want to make the new mode the default for files with certain recognizable names, add an element to `auto-mode-alist` to select the mode for those file names. If you define the mode command to autoload, you should add this element in the same file that calls `autoload`. Otherwise, it is sufficient to add the element in the file that contains the mode definition. See Section 22.1.3 [Auto Major Mode], page 398.
- In the documentation, you should provide a sample `autoload` form and an example of how to add to `auto-mode-alist`, that users can include in their '`.emacs`' files.
- The top-level forms in the file defining the mode should be written so that they may be evaluated more than once without adverse consequences. Even if you never load the file more than once, someone else will.

22.1.2 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 mode-specific tables.
(defvar text-mode-syntax-table nil
  "Syntax table used while in text mode.")

(if text-mode-syntax-table
    ()              ; Do not change the table if it is already set up.
  (setq text-mode-syntax-table (make-syntax-table))
  (modify-syntax-entry ?\" ".   " text-mode-syntax-table)
  (modify-syntax-entry ?\\ ".   " text-mode-syntax-table)
  (modify-syntax-entry ?' "w   " text-mode-syntax-table))
```

```
(defvar text-mode-abbrev-table nil
  "Abbrev table used while in text mode.")
(define-abbrev-table 'text-mode-abbrev-table ())

(defvar text-mode-map nil)    ; Create a mode-specific keymap.


(if text-mode-map
    ()              ; Do not change the keymap if it is already set up.
  (setq text-mode-map (make-sparse-keymap))
  (define-key text-mode-map "\t" 'indent-relative)
  (define-key text-mode-map "\es" 'center-line)
  (define-key text-mode-map "\eS" 'center-paragraph))
```

Here is the complete major mode function definition for Text mode:

```
(defun text-mode ()
  "Major mode for editing text intended for humans to read....
 Special commands: \\{text-mode-map}
Turning on text-mode runs the hook `text-mode-hook'."
  (interactive)
  (kill-all-local-variables)
  (use-local-map text-mode-map)
  (setq local-abbrev-table text-mode-abbrev-table)
  (set-syntax-table text-mode-syntax-table)
  (make-local-variable 'paragraph-start)
  (setq paragraph-start (concat "[ \t]*$\\|" page-delimiter))
  (make-local-variable 'paragraph-separate)
  (setq paragraph-separate paragraph-start)
  (setq mode-name "Text")
  (setq major-mode 'text-mode)
  (run-hooks 'text-mode-hook))      ; Finally, this permits the user to
                                    ;    customize the mode with a hook.
```

The three Lisp modes (Lisp mode, Emacs Lisp mode, and Lisp Interaction mode) have more features than Text mode and the code is correspondingly more complicated. Here are excerpts from '`lisp-mode.el`' that illustrate how these modes are written.

```
;; Create mode-specific table variables.
(defvar lisp-mode-syntax-table nil "")
(defvar emacs-lisp-mode-syntax-table nil "")
(defvar lisp-mode-abbrev-table nil "")

(if (not emacs-lisp-mode-syntax-table) ; Do not change the table
                                       ;    if it is already set.
    (let ((i 0))
      (setq emacs-lisp-mode-syntax-table (make-syntax-table))
```

```
      ;; Set syntax of chars up to 0 to class of chars that are
      ;;    part of symbol names but not words.
      ;;    (The number 0 is 48 in the ASCII character set.)
      (while (< i ?0)
        (modify-syntax-entry i "_   " emacs-lisp-mode-syntax-table)
        (setq i (1+ i)))
      ...
      ;; Set the syntax for other characters.
      (modify-syntax-entry ?  "    " emacs-lisp-mode-syntax-table)
      (modify-syntax-entry ?\t "    " emacs-lisp-mode-syntax-table)
      ...
      (modify-syntax-entry ?\( "()  " emacs-lisp-mode-syntax-table)
      (modify-syntax-entry ?\) ")(  " emacs-lisp-mode-syntax-table)
      ...))
;; Create an abbrev table for lisp-mode.
(define-abbrev-table 'lisp-mode-abbrev-table ())
```

Much code is shared among the three Lisp modes. The following function sets various variables; it is called by each of the major Lisp mode functions:

```
(defun lisp-mode-variables (lisp-syntax)
  (cond (lisp-syntax
  (set-syntax-table lisp-mode-syntax-table)))
  (setq local-abbrev-table lisp-mode-abbrev-table)
  ...
```

Functions such as `forward-paragraph` use the value of the `paragraph-start` variable. Since Lisp code is different from ordinary text, the `paragraph-start` variable needs to be set specially to handle Lisp. Also, comments are indented in a special fashion in Lisp and the Lisp modes need their own mode-specific `comment-indent-function`. The code to set these variables is the rest of `lisp-mode-variables`.

```
  (make-local-variable 'paragraph-start)
  (setq paragraph-start (concat page-delimiter "\\|$" ))
  (make-local-variable 'paragraph-separate)
  (setq paragraph-separate paragraph-start)
  ...
  (make-local-variable 'comment-indent-function)
  (setq comment-indent-function 'lisp-comment-indent))
```

Each of the different Lisp modes has a slightly different keymap. For example, Lisp mode binds `C-c C-z` to `run-lisp`, but the other Lisp modes do not. However, all Lisp modes have some commands in common. The following code sets up the common commands:

```
(defvar shared-lisp-mode-map ()
  "Keymap for commands shared by all sorts of Lisp modes.")

(if shared-lisp-mode-map
    ()
   (setq shared-lisp-mode-map (make-sparse-keymap))
   (define-key shared-lisp-mode-map "\e\C-q" 'indent-sexp)
   (define-key shared-lisp-mode-map "\177"
               'backward-delete-char-untabify))
```

And here is the code to set up the keymap for Lisp mode:

```
(defvar lisp-mode-map ()
  "Keymap for ordinary Lisp mode....")

(if lisp-mode-map
    ()
  (setq lisp-mode-map (make-sparse-keymap))
  (set-keymap-parent lisp-mode-map shared-lisp-mode-map)
  (define-key lisp-mode-map "\e\C-x" 'lisp-eval-defun)
  (define-key lisp-mode-map "\C-c\C-z" 'run-lisp))
```

Finally, here is the complete major mode function definition for Emacs Lisp mode.

```
(defun lisp-mode ()
  "Major mode for editing Lisp code for Lisps other than GNU Emacs Lisp.
Commands:
Delete converts tabs to spaces as it moves back.
Blank lines separate paragraphs.  Semicolons start comments.
\\{lisp-mode-map}
Note that `run-lisp' may be used either to start an inferior Lisp job
or to switch back to an existing one.

Entry to this mode calls the value of `lisp-mode-hook'
if that value is non-nil."
  (interactive)
  (kill-all-local-variables)
  (use-local-map lisp-mode-map)          ; Select the mode's keymap.
  (setq major-mode 'lisp-mode)           ; This is how describe-mode
                                         ;    finds out what to describe.
  (setq mode-name "Lisp")                ; This goes into the mode line.
  (lisp-mode-variables t)                ; This defines various variables.
  (setq imenu-case-fold-search t)
  (set-syntax-table lisp-mode-syntax-table)
  (run-hooks 'lisp-mode-hook))           ; This permits the user to use a
                                         ;    hook to customize the mode.
```

22.1.3 How Emacs Chooses a Major Mode

Based on information in the file name or in the file itself, Emacs automatically selects a major mode for the new buffer when a file is visited. It also processes local variables specified in the file text.

fundamental-mode Command

Fundamental mode is a major mode that is not specialized for anything in particular. Other major modes are defined in effect by comparison with this one—their definitions say what to change, starting from Fundamental mode. The `fundamental-mode` function does *not* run any hooks; you're not supposed to customize it. (If you want Emacs to behave differently in Fundamental mode, change the *global* state of Emacs.)

normal-mode &optional *find-file* Command

This function establishes the proper major mode and buffer-local variable bindings for the current buffer. First it calls `set-auto-mode`, then it runs `hack-local-variables` to parse, and bind or evaluate as appropriate, the file's local variables.

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 a local variables list at the end of the file and in the '`-*-`' line. 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 local variables list.

`normal-mode` uses `condition-case` around the call to the major mode function, so errors are caught and reported as a '`File mode specification error`', followed by the original error message.

enable-local-variables User Option

This variable controls processing of local variables lists in files being visited. A value of `t` means process the local variables lists unconditionally; `nil` means ignore them; anything else means ask the user what to do for each file. The default value is `t`.

ignored-local-variables Variable

This variable holds a list of variables that should not be set by a file's local variables list. Any value specified for one of these variables is ignored.

In addition to this list, any variable whose name has a non-`nil` `risky-local-variable` property is also ignored.

enable-local-eval User Option

This variable controls processing of '`Eval:`' in 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`.

set-auto-mode Function

This function selects the major mode that is appropriate for the current buffer. It may base its decision on the value of the '`-*-`' line, on the visited file name (using `auto-mode-alist`), on the '`#!`' line (using `interpreter-mode-alist`), or on the file's local variables list. However, this function does not look for the '`mode:`' local variable near the end of a file; the `hack-local-variables` function does that. See section "How Major Modes are Chosen" in *The GNU Emacs Manual*.

default-major-mode User Option

This variable holds the default major mode for new buffers. The standard value is `fundamental-mode`.

If the value of `default-major-mode` is `nil`, Emacs uses the (previously) current buffer's major mode for the major mode of a new buffer. However, if that major mode symbol has a `mode-class` property with value `special`, then it is not used for new buffers; Fundamental mode is used instead. The modes that have this property are those such as Dired and Rmail that are useful only with text that has been specially prepared.

set-buffer-major-mode *buffer* Function

This function sets the major mode of *buffer* to the value of `default-major-mode`. If that variable is `nil`, it uses the current buffer's major mode (if that is suitable).

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 Variable

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`.

auto-mode-alist Variable

This variable contains an association list of file name patterns (regular expressions; see Section 33.2 [Regular Expressions], page 649) 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 24.8.4 [File Name Expansion], page 454) 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 '`.emacs`' 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))
```

interpreter-mode-alist Variable

This variable specifies major modes to use for scripts that specify a command interpreter in an '`#!`' line. Its value is a list of 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*. The value of *interpreter* is actually a regular expression.

This variable is applicable only when the `auto-mode-alist` does not indicate which major mode to use.

hack-local-variables &optional *force* Function

This function parses, and binds or evaluates as appropriate, any local variables specified by the contents of the current buffer.

The handling of `enable-local-variables` documented for `normal-mode` actually takes place here. The argument *force* usually comes from the argument *find-file* given to `normal-mode`.

22.1.4 Getting Help about a Major Mode

The `describe-mode` function is used to provide information about major modes. It is normally called with `C-h m`. The `describe-mode` function uses the value of `major-mode`, which is why every major mode function needs to set the `major-mode` variable.

describe-mode Command

This function displays the documentation of the current major mode.

The `describe-mode` function calls the `documentation` function using the value of `major-mode` as an argument. Thus, it displays the documentation string of the major mode function. (See Section 23.2 [Accessing Documentation], page 424.)

major-mode Variable

This variable holds the symbol for the current buffer's major mode. This symbol should have a function definition that is the command to switch to that major mode. The `describe-mode` function uses the documentation string of the function as the documentation of the major mode.

22.1.5 Defining Derived Modes

It's often useful to define a new major mode in terms of an existing one. An easy way to do this is to use `define-derived-mode`.

define-derived-mode *variant parent name docstring body...* Macro

This construct defines *variant* as a major mode command, using *name* as the string form of the mode name.

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 keymap, named *variant*`-map`. `define-derived-mode` initializes this map to inherit from *parent*`-map`, if it is not already set.
- The new mode has its own syntax table, kept in the variable *variant*`-syntax-table`. `define-derived-mode` initializes this variable by copying *parent*`-syntax-table`, if it is not already set.
- The new mode has its own abbrev table, kept in the variable *variant*`-abbrev-table`. `define-derived-mode` initializes this variable by copying *parent*`-abbrev-table`, if it is not already set.

- The new mode has its own mode hook, *variant*`-hook`, which it runs in standard fashion as the very last thing that it does. (The new mode also runs the mode hook of *parent* as part of calling *parent*.)

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 *variant*`-hook`.

The argument *docstring* specifies the documentation string for the new mode. If you omit *docstring*, `define-derived-mode` generates a documentation string.

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)
```

22.2 Minor Modes

A *minor mode* provides features that users may enable or disable independently of the choice of major mode. Minor modes can be enabled individually or in combination. Minor modes would be better named "generally available, optional feature modes," except that such a name would be unwieldy.

A minor mode is not usually a modification of single major mode. For example, Auto Fill mode works with any major mode that permits text insertion. To be general, a minor mode must be effectively independent of the things major modes do.

A minor mode is often much more difficult to implement than a major mode. One reason is that you should be able to activate and deactivate minor modes in any order. A minor mode should be able to have its desired effect regardless of the major mode and regardless of the other minor modes in effect.

Often the biggest problem in implementing a minor mode is finding a way to insert the necessary hook into the rest of Emacs. Minor mode keymaps make this easier than it used to be.

22.2.1 Conventions for Writing Minor Modes

There are conventions for writing minor modes just as there are for major modes. Several of the major mode conventions apply to minor modes as well:

those regarding the name of the mode initialization function, the names of global symbols, and the use of keymaps and other tables.

In addition, there are several conventions that are specific to minor modes.

- Make a variable whose name ends in '`-mode`' to control the minor mode. We call this the *mode variable*. The minor mode command should set this variable (`nil` to disable; anything else to enable).

  If it is possible, implement the mode so that setting the variable automatically enables or disables the mode. Then the minor mode command does not need to do anything except set the variable.

  This variable is used in conjunction with the `minor-mode-alist` to display the minor mode name in the mode line. It can also enable or disable a minor mode keymap. Individual commands or hooks can also check the variable's value.

  If you want the minor mode to be enabled separately in each buffer, make the variable buffer-local.

- Define a command whose name is the same as the mode variable. Its job is to enable and disable the mode by setting the variable.

  The command should accept one optional argument. If the argument is `nil`, it should toggle the mode (turn it on if it is off, and off if it is on). Otherwise, it should turn the mode on if the argument is a positive integer, a symbol other than `nil` or `-`, or a list whose CAR is such an integer or symbol; it should turn the mode off otherwise.

  Here is an example taken from the definition of `transient-mark-mode`. It shows the use of `transient-mark-mode` as a variable that enables or disables the mode's behavior, and also shows the proper way to toggle, enable or disable the minor mode based on the raw prefix argument value.

```
(setq transient-mark-mode
      (if (null arg) (not transient-mark-mode)
        (> (prefix-numeric-value arg) 0)))
```

- Add an element to `minor-mode-alist` for each minor mode (see Section 22.3.2 [Mode Line Variables], page 408), 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:

```
(or (assq 'leif-mode minor-mode-alist)
    (setq minor-mode-alist
          (cons '(leif-mode " Leif") minor-mode-alist)))
```

You can also use `add-to-list` to add an element to this list just once (see Section 10.8 [Setting Variables], page 156).

22.2.2 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 Section 21.6 [Active Keymaps], page 367.

One use of minor mode keymaps is to modify the behavior of certain self-inserting characters so that they do something else as well as self-insert. In general, this is the only way to do that, since the facilities for customizing `self-insert-command` are limited to special cases (designed for abbrevs and Auto Fill mode). (Do not try substituting your own definition of `self-insert-command` for the standard one. The editor command loop handles this function specially.)

The key sequences bound in a minor mode should consist of `C-c` followed by a punctuation character *other than* `{`, `}`, `<`, `>`, `:` or `;`. (Those few punctuation characters are reserved for major modes.)

22.2.3 Easy-Mmode

The easy-mmode package provides a convenient way of implementing a minor mode; with it, you can specify all about a simple minor mode in one self-contained definition.

easy-mmode-define-minor-mode *mode doc* &optional *init-value mode-indicator keymap* Macro

This macro defines a new minor mode whose name is *mode* (a symbol).

This macro defines a command named *mode* which toggles the minor mode, and has *doc* as its documentation string.

It also defines a variable named *mode*, which is set to `t` or `nil` by enabling or disabling the mode. The variable is initialized to *init-value*.

The string *mode-indicator* says what to display in the mode line when the mode is enabled; if it is `nil`, the mode is not displayed in the mode line.

The optional argument *keymap* specifies the keymap for the minor mode. It can be a variable name, whose value is the keymap, or it can be an alist specifying bindings in this form:

(*key-sequence* . *definition*)

Here is an example of using `easy-mmode-define-minor-mode`:

```
(easy-mmode-define-minor-mode hungry-mode
  "Toggle Hungry mode.
With no argument, this command toggles the mode.
Non-null prefix argument turns on the mode.
Null prefix argument turns off the mode.

When Hungry mode is enabled, the control delete key
gobbles all preceding whitespace except the last.
See the command \\[hungry-electric-delete]."
 ;; The initial value.
 nil
 ;; The indicator for the mode line.
 " Hungry"
 ;; The minor mode bindings.
 '(("\C-\^?" . hungry-electric-delete)
   ("\C-\M-\^?"
    . (lambda ()
        (interactive)
        (hungry-electric-delete t)))))
```

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 key bindings for *C-*DEL and *C-M-*DEL.

22.3 Mode Line Format

Each Emacs window (aside from minibuffer windows) includes a mode line, 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 the major and minor modes.

This section describes how the contents of the mode line are controlled. We include it in this chapter because much of the information displayed in the mode line relates to the enabled major and minor modes.

`mode-line-format` is a buffer-local variable that holds a template used to display the mode line of the current buffer. All windows for the same buffer use the same `mode-line-format` and their mode lines appear the same (except for scrolling percentages, and line and column numbers).

The mode line of a window is normally updated whenever a different buffer is shown in the window, or when the buffer's modified-status changes from `nil` to `t` or vice-versa. If you modify any of the variables referenced by `mode-line-format` (see Section 22.3.2 [Mode Line Variables], page 408),

or any other variables and data structures that affect how text is displayed (see Chapter 38 [Display], page 739), you may want to force an update of the mode line so as to display the new information or display it in the new way.

force-mode-line-update Function

Force redisplay of the current buffer's mode line.

The mode line is usually displayed in inverse video; see `mode-line-inverse-video` in Section 38.12 [Inverse Video], page 760.

22.3.1 The Data Structure of the Mode Line

The mode line contents are controlled by a data structure of lists, strings, symbols, and numbers kept in the buffer-local variable `mode-line-format`. The data structure is called a *mode line construct*, and it is built in recursive fashion out of simpler mode line constructs. The same data structure is used for constructing frame titles (see Section 28.4 [Frame Titles], page 534).

mode-line-format Variable

The value of this variable is a mode line construct with overall responsibility for the mode line format. The value of this variable controls which other variables are used to form the mode line text, and where they appear.

A mode line construct may be as simple as a fixed string of text, but it usually specifies how to use other variables to construct the text. Many of these variables are themselves defined to have mode line constructs as their values.

The default value of `mode-line-format` incorporates the values of variables such as `mode-name` and `minor-mode-alist`. Because of this, 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` refers to.

A mode line construct may be a list, a symbol, or a string. If the value is a list, each element may be a list, a symbol, or a string.

string A string as a mode line construct is displayed verbatim in the mode line except for **%**-*constructs*. Decimal digits after the '**%**' specify the field width for space filling on the right (i.e., the data is left justified). See Section 22.3.3 [%-Constructs], page 410.

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; so 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.

(*string rest...*) or (*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.

(*symbol then else*)
: A list whose first element is a symbol is a conditional. Its meaning depends on the value of *symbol*. If the value is non-`nil`, the second element, *then*, is processed recursively as a mode line element. But if the value of *symbol* is `nil`, 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`.

(*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. Then the result is space filled (if *width* is positive) or truncated (to $-$*width* columns, if *width* is negative) on the right.

  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")`.

If you do alter `mode-line-format` itself, the new value should use the same variables that appear in the default value (see Section 22.3.2 [Mode Line Variables], page 408), rather than duplicating their contents or displaying the information in another fashion. This way, customizations made by the user or by Lisp programs (such as `display-time` and major modes) via changes to those variables remain effective.

Here is an example of a `mode-line-format` that might be useful for `shell-mode`, since it contains the host name and default directory.

```
(setq mode-line-format
  (list "-"
   'mode-line-mule-info
   'mode-line-modified
   'mode-line-frame-identification
   "%b--"
   ;; Note that this is evaluated while making the list.
   ;; It makes a mode line construct which is just a string.
   (getenv "HOST")
   ":"
   'default-directory
   "   "
   'global-mode-string
   "   %[("
```

```
        '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.)

22.3.2 Variables Used in the Mode Line

This section describes variables incorporated by the standard value of `mode-line-format` into the text of the mode line. There is nothing inherently special about these variables; any other variables could have the same effects on the mode line if `mode-line-format` were changed 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 32 [Non-ASCII Characters], page 629.

mode-line-modified Variable

This variable holds the value of the mode-line construct that displays whether the current buffer is modified.

The default value of `mode-line-modified` is `("%1*%1+")`. This means that the mode line displays '`**`' if the buffer is modified, '`--`' if the buffer is not modified, '`%%`' if the buffer is read only, and '`%*`' if the buffer is read only and modified.

Changing this variable does not force an update of the mode line.

mode-line-frame-identification Variable

This variable identifies the current frame. The default value is `" "` if you are using a window system which can show multiple frames, or `"-%F "` on an ordinary terminal which shows only one frame at a time.

mode-line-buffer-identification Variable

This variable identifies the buffer being displayed in the window. Its default value is `("%12b")`, which displays the buffer name, padded with spaces to at least 12 columns.

global-mode-string Variable

This variable holds a mode line spec that appears in the mode line by default, just after the buffer name. 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`.

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.

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 spec. It appears in the mode line when the value of *minor-mode-variable* is non-`nil`, and not otherwise. These strings should begin with spaces so that they don't run together. Conventionally, the *minor-mode-variable* for a specific mode is set to a non-`nil` value when that minor mode is activated.

The default value of `minor-mode-alist` is:

```
minor-mode-alist
⇒ ((vc-mode vc-mode)
    (abbrev-mode " Abbrev")
    (overwrite-mode overwrite-mode)
    (auto-fill-function " Fill")
    (defining-kbd-macro " Def")
    (isearch-mode isearch-mode))
```

`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.

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`.

default-mode-line-format Variable

This variable holds the default `mode-line-format` for buffers that do not override it. This is the same as `(default-value 'mode-line-format)`.

The default value of `default-mode-line-format` is this list:

```
("-"
 mode-line-mule-info
 mode-line-modified
 mode-line-frame-identification
 mode-line-buffer-identification
 "   "
 global-mode-string
 "   %[("
 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")
 "-%-")
```

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 `nil` for no version control, or a string that appears in the mode line.

22.3.3 %-Constructs in the Mode Line

The following table lists the recognized `%`-constructs and what they mean. In any construct except '`%%`', you can add a decimal integer after the '`%`' to specify how many characters to display.

`%b` The current buffer name, obtained with the `buffer-name` function. See Section 26.3 [Buffer Names], page 482.

`%f` The visited file name, obtained with the `buffer-file-name` function. See Section 26.4 [Buffer File Name], page 483.

`%F` The title (only on a window system) or the name of the selected frame. See Section 28.3.3 [Window Frame Parameters], page 528.

`%c` The current column number of point.

`%l` The current line number of point.

`%*` '`%`' if the buffer is read only (see `buffer-read-only`);
'`*`' if the buffer is modified (see `buffer-modified-p`);
'`-`' otherwise. See Section 26.5 [Buffer Modification], page 485.

`%+` '`*`' 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 26.5 [Buffer Modification], page 485.

`%&` '`*`' if the buffer is modified, and '`-`' otherwise.

`%s` The status of the subprocess belonging to the current buffer, obtained with `process-status`. See Section 36.6 [Process Information], page 697.

`%t` Whether the visited file is a text file or a binary file. (This is a meaningful distinction only on certain operating systems.)

`%p` The percentage of the buffer text above the **top** of window, or '`Top`', '`Bottom`' or '`All`'.

`%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`'.

`%n` '`Narrow`' when narrowing is in effect; nothing otherwise (see `narrow-to-region` in Section 29.4 [Narrowing], page 561).

`%[` An indication of the depth of recursive editing levels (not counting minibuffer levels): one '`[`' for each editing level. See Section 20.11 [Recursive Editing], page 355.

`%]` One '`]`' for each recursive editing level (not counting minibuffer levels).

`%%` The character '`%`'—this is how to include a literal '`%`' in a string in which `%`-constructs are allowed.

`%-` Dashes sufficient to fill the remainder of the mode line.

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`. Currently, only `display-time` modifies the value of `global-mode-string`.

22.4 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 positions of the definitions or portions of in the buffer, so the user can pick one of them to move to. This section explains how to customize Imenu for a major mode.

The usual and simplest way is to set the variable `imenu-generic-expression`:

imenu-generic-expression Variable

This variable, if non-`nil`, specifies regular expressions for finding definitions for Imenu. In the simplest case, elements should look like this:

(*menu-title regexp subexp*)

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 33.2 [Regular Expressions], page 649); wherever it matches, that is a definition to mention in the buffer index. The third item, *subexp*, 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 a special index item which, if selected by the user, calls *function* with arguments *item-name*, the buffer position, and *arguments*.

For Emacs Lisp mode, *pattern* 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 *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 34.3 [Syntax Table Functions], page 674).

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` patterns can then use '`\\sw+`' instead of '`\\(\\sw\\|\\s_\\)+`'. Note that this technique may be inconvenient to use 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 for finding the next definition to mention in the buffer index, moving backwards in the file.

The function should leave point at the place to be connected to the index item; it should return `nil` if it doesn't find another item.

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 variables `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 for the current

buffer. It is called within `save-excursion`, so where it leaves point makes no difference.

The default value is a function that uses `imenu-generic-expression` to produce the index alist. If you specify a different function, then `imenu-generic-expression` is not used.

Setting this variable makes it buffer-local in the current buffer.

imenu-index-alist Variable

This variable holds the index alist for the current buffer. Setting it makes it buffer-local in the current buffer.

Simple elements in the alist look like (*index-name* . *index-position*). Selecting a simple element has the effect of moving to position *index-position* in the buffer.

Special elements look like (*index-name position function arguments*...). Selecting a special element performs

```
(funcall function index-name position arguments...)
```

A nested sub-alist element looks like (*index-name sub-alist*).

22.5 Font Lock Mode

Font Lock mode is a feature that automatically attaches `face` properties to certain parts of the buffer based on their syntactic role. How it parses the buffer depends on the major mode; most major modes define syntactic criteria for which faces to use, in which contexts. This section explains how to customize Font Lock for a particular language—in other words, 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 using `font-lock-comment-face` and `font-lock-string-face` (see Section 22.5.5 [Faces for Font Lock], page 419); search-based fontification follows.

22.5.1 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, it should set `font-lock-defaults` as a buffer-local variable. The value assigned to this variable is used, if and when Font Lock mode is enabled, to set all the other variables.

font-lock-defaults Variable

This variable is set by major modes, as a buffer-local variable, to specify how to fontify text in that mode. 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`. It can be a symbol, a variable whose value is list to use for `font-lock-keywords`. It can also be a list of several such symbols, one for each possible level of fontification. The first symbol specifies how to do level 1 fontification, the second symbol how to do level 2, and so on.

The second element, *keywords-only*, specifies the value of the variable `font-lock-keywords-only`. If this is non-`nil`, syntactic fontification (of strings and comments) is not performed.

The third element, *case-fold*, specifies the value of `font-lock-case-fold-search`. If it is non-`nil`, Font Lock mode ignores case when searching as directed by `font-lock-keywords`.

If the fourth element, *syntax-alist*, is non-`nil`, it should be a list of cons cells of the form (*char-or-string* . *string*). These are used to set up a syntax table for fontification (see Section 34.3 [Syntax Table Functions], page 674). The resulting syntax table is stored in `font-lock-syntax-table`.

The fifth element, *syntax-begin*, specifies the value of `font-lock-beginning-of-syntax-function` (see below).

Any further elements *other-vars* are have form (*variable* . *value*). This kind of element means to make *variable* buffer-local and then set it to *value*. This is used to set other variables that affect fontification.

22.5.2 Search-based Fontification

The most important variable for customizing Font Lock mode is `font-lock-keywords`. It specifies the search criteria for search-based fontification.

font-lock-keywords Variable

This variable's value is a list of the keywords to highlight. Be careful when composing regular expressions for this list; a poorly written pattern can dramatically slow things down!

Each element of `font-lock-keywords` specifies how to find certain cases of text, and how to highlight those cases. Font Lock mode processes the elements of `font-lock-keywords` one by one, and for each element, it finds and handles all matches. Ordinarily, once part of the text has been fontified already, this cannot be overridden by a subsequent match in the same text; but you can specify different behavior using the *override* element of a *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 discrete occurrences of 'foo'
;; using font-lock-keyword-face.
"\\<foo\\>"
```

The function `regexp-opt` (see Section 33.2.1 [Syntax of Regexps], page 649) is useful for calculating optimal regular expressions to match a number of different keywords.

function Find text by calling *function*, and highlight the matches it finds using `font-lock-keyword-face`.

When *function* is called, it receives one argument, the limit of the search. It should return non-`nil` if it succeeds, and set the match data to describe the match that was found.

(*matcher* . *match*)
In this kind of element, *matcher* stands for either a regular expression or a function, as described above. The CDR, *match*, specifies which subexpression of *matcher* should be highlighted (instead of the entire text that *matcher* matched).

```
;; Highlight the 'bar' in each occurrences of 'fubar',
;; using font-lock-keyword-face.
("fu\\(bar\\)" . 1)
```

If you use `regexp-opt` to produce the regular expression *matcher*, then you can use `regexp-opt-depth` (see Section 33.2.1 [Syntax of Regexps], page 649) to calculate the value for *match*.

(*matcher* . *facename*)
In this kind of element, *facename* is an expression whose value specifies the face name to use for highlighting.

```
;; Highlight occurrences of 'fubar',
;; using the face which is the value of fubar-face.
("fubar" . fubar-face)
```

(*matcher* . *highlighter*)
In this kind of element, *highlighter* is a list which specifies how to highlight matches found by *matcher*. It has the form

(*subexp facename 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, *facename*, specifies the face, as described above.

The last two values in *highlighter*, *override* and *laxmatch*, are 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

facename is added to the beginning of the `face` property. If it is `append`, the face *facename* is added to the end of the `face` property.

If *laxmatch* is non-`nil`, it means there should be no error if there is no subexpression numbered *subexp* in *matcher*.

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 occurrences
;; that the function fubar-match finds,
;; using the face which is the value of fubar-face.
(fubar-match 1 fubar-face)
```

(*matcher highlighters*...)
: This sort of element specifies several *highlighter* lists for a single *matcher*. In order for this to be useful, each *highlighter* should have a different value of *subexp*; that is, each one should apply to a different subexpression of *matcher*.

`(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. While `font-lock-fontify-buffer` handles multi-line patterns correctly, updating when you edit the buffer does not, since it considers text one line at a time.

22.5.3 Other Font Lock Variables

This section describes additional variables that a major mode can set by means of `font-lock-defaults`.

font-lock-keywords-only Variable
: Non-`nil` means Font Lock should not fontify comments or strings syntactically; it should only fontify based on `font-lock-keywords`.

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.

font-lock-syntax-table Variable

This variable specifies the syntax table to use for fontification of comments and strings.

font-lock-beginning-of-syntax-function Variable

If this variable is non-`nil`, it should be a function to move point back to a position that is syntactically at "top level" and outside of strings or comments. Font Lock uses this when necessary to get the right results for syntactic fontification.

This function is called with no arguments. It should leave point at the beginning of any enclosing syntactic block. Typical values are `beginning-of-line` (i.e., 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 (i.e., the mode-dependent function is known to move outside a syntactic block).

If the value is `nil`, the beginning of the buffer is used as a position outside of a syntactic block. This cannot be wrong, but it can be slow.

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-g M-g* (`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.

22.5.4 Levels of Font Lock

Many major modes offer three different levels of fontification. You can define multiple levels by using a list of symbols for *keywords* in `font-lock-defaults`. Each symbol specifies one level of fontification; it is up to the user to choose one of these levels. The chosen level's symbol value is used to initialize `font-lock-keywords`.

Here are the conventions for how to define the levels of fontification:

- Level 1: highlight function declarations, file directives (such as include or import directives), strings and comments. The idea is speed, so only the most important and top-level components are fontified.
- Level 2: in addition to level 1, highlight all language keywords, including type names that act like keywords, as well as named constant values. The idea is that all keywords (either syntactic or semantic) should be fontified appropriately.
- Level 3: in addition to level 2, highlight the symbols being defined in function and variable declarations, and all builtin function names, wherever they appear.

22.5.5 Faces for Font Lock

You can make Font Lock mode use any face, but several faces are defined specifically for Font Lock mode. Each of these symbols is both a face name, and a variable whose default value is the symbol itself. Thus, the default value of `font-lock-comment-face` is `font-lock-comment-face`. This means you can write `font-lock-comment-face` in a context such as `font-lock-keywords` where a face-name-valued expression is used.

`font-lock-comment-face`
: Used (typically) for comments.

`font-lock-string-face`
: Used (typically) for string constants.

`font-lock-keyword-face`
: Used (typically) for keywords—names that have special syntactic significance, like `for` and `if` in C.

`font-lock-builtin-face`
: Used (typically) for built-in function names.

`font-lock-function-name-face`
: Used (typically) for the name of a function being defined or declared, in a function definition or declaration.

`font-lock-variable-name-face`
: Used (typically) for the name of a variable being defined or declared, in a variable definition or declaration.

`font-lock-type-face`
: Used (typically) for names of user-defined data types, where they are defined and where they are used.

`font-lock-constant-face`
: Used (typically) for constant names.

`font-lock-warning-face`
: Used (typically) for constructs that are peculiar, or that greatly change the meaning of other text. For example, this is used for '`;;;###autoload`' cookies in Emacs Lisp, and for `#error` directives in C.

22.5.6 Syntactic Font Lock

Font Lock mode can be used to update `syntax-table` properties automatically. This is useful in languages for which a single syntax table by itself is not sufficient.

font-lock-syntactic-keywords Variable

This variable enables and controls syntactic Font Lock. Its value should be a list of elements of this form:

(*matcher subexp syntax override laxmatch*)

The parts of this element have the same meanings as in the corresponding sort of element of `font-lock-keywords`,

(*matcher subexp facename override laxmatch*)

However, instead of specifying the value *facename* to use for the `face` property, it specifies the value *syntax* to use for the `syntax-table` property. Here, *syntax* can be a variable whose value is a syntax table, a syntax entry of the form (*syntax-code* . *matching-char*), or an expression whose value is one of those two types.

22.6 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 '`.emacs`' file, but Lisp programs can set them also. See Appendix F [Standard Hooks], page 821, for a list of standard hook variables.

Most of the hooks in Emacs are *normal hooks*. These variables contain lists of functions to be called with no arguments. When the hook name ends in '`-hook`', that tells you it is normal. We try to make all hooks normal, as much as possible, so that you can use them in a uniform way.

Every major mode function is supposed to run a normal hook called the *mode hook* as the last step 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. But hooks are used in other contexts too. For example, the hook `suspend-hook` runs just before Emacs suspends itself (see Section 37.2.2 [Suspending Emacs], page 717).

The recommended way to add a hook function to a normal hook is by calling `add-hook` (see below). The hook functions may be any of the valid kinds of functions that `funcall` accepts (see Section 11.1 [What Is a Function], page 171). Most normal hook variables are initially void; `add-hook` knows how to deal with this.

If the hook variable's name does not end with '`-hook`', that indicates it is probably an *abnormal hook*; you should look at its documentation to see how to use the hook properly.

If the variable's name ends in '`-functions`' or '`-hooks`', then the value is a list of functions, but it is abnormal in that either these functions are called with arguments or their values are used in some way. You can use `add-hook` to add a function to the list, but you must take care in writing the function. (A few of these variables are actually normal hooks which were named before we established the convention of using '`-hook`' for them.)

If the variable's name ends in '`-function`', then its value is just a single function, not a list of functions.

Here's an example that uses a mode hook to turn on Auto Fill mode when in Lisp Interaction mode:

```
(add-hook 'lisp-interaction-mode-hook 'turn-on-auto-fill)
```

At the appropriate time, Emacs uses the `run-hooks` function to run particular hooks. This function calls the hook functions that have been added with `add-hook`.

run-hooks &rest *hookvar* Function

This function takes one or more hook variable names as arguments, and runs each hook in turn. Each *hookvar* argument should be a symbol that is a hook variable. These arguments are processed in the order specified.

If a hook variable has a non-`nil` value, that value may be a function or a list of functions. If the value is a function (either a lambda expression or a symbol with a function definition), it is called. If it is a list, the elements are called, in order. The hook functions are called with no arguments. Nowadays, storing a single function in the hook variable is semi-obsolete; you should always use a list of functions.

For example, here's how `emacs-lisp-mode` runs its mode hook:

```
(run-hooks 'emacs-lisp-mode-hook)
```

run-hook-with-args *hook* &rest *args* Function

This function is the way to run an abnormal hook which passes arguments to the hook functions. It calls each of the hook functions, passing each of them the arguments *args*.

run-hook-with-args-until-failure *hook* &rest *args* Function

This function is the way to run an abnormal hook which passes arguments to the hook functions, and stops as soon as any hook function fails. It calls each of the hook functions, passing each of them the arguments *args*, until some hook function returns `nil`. Then it stops, and returns `nil` if some hook function did, and otherwise returns a non-`nil` value.

run-hook-with-args-until-success *hook* &rest *args* Function

This function is the way to run an abnormal hook which passes arguments to the hook functions, and stops as soon as any hook function succeeds. It calls each of the hook functions, passing each of them the arguments *args*, until some hook function returns non-`nil`. Then it stops, and returns whatever was returned by the last hook function that was called.

add-hook *hook function* &optional *append local* Function

This function is the handy way to add function *function* to hook variable *hook*. The argument *function* may be any valid Lisp function with the proper number of arguments. For example,

```
(add-hook 'text-mode-hook 'my-text-hook-function)
```

adds `my-text-hook-function` to the hook called `text-mode-hook`.

You can use `add-hook` for abnormal hooks as well as for normal hooks.

It is best to design your hook functions so that the order in which they are executed does not matter. Any dependence on the order is "asking for trouble." However, the order is predictable: normally, *function* goes at the front of the hook list, so it will be executed first (barring another `add-hook` call). If the optional argument *append* is non-`nil`, the new hook function goes at the end of the hook list and will be executed last.

If *local* is non-`nil`, that says to make the new hook function buffer-local in the current buffer. Before you can do this, you must make the hook itself buffer-local by calling `make-local-hook` (**not** `make-local-variable`). If the hook itself is not buffer-local, then the value of *local* makes no difference—the hook function is always global.

remove-hook *hook function* &optional *local* Function

This function removes *function* from the hook variable *hook*.

If *local* is non-`nil`, that says to remove *function* from the buffer-local hook list instead of from the global hook list. If the hook variable itself is not buffer-local, then the value of *local* makes no difference.

make-local-hook *hook* Function

This function makes the hook variable `hook` buffer-local in the current buffer. When a hook variable is buffer-local, it can have buffer-local and global hook functions, and `run-hooks` runs all of them.

This function works by making `t` an element of the buffer-local value. That serves as a flag to use the hook functions in the default value of the hook variable as well as those in the buffer-local value. Since `run-hooks` understands this flag, `make-local-hook` works with all normal hooks. It works for only some non-normal hooks—those whose callers have been updated to understand this meaning of `t`.

Do not use `make-local-variable` directly for hook variables; it is not sufficient.

23 Documentation

GNU Emacs Lisp has convenient on-line help facilities, most of which derive their information from the documentation strings associated with functions and variables. This chapter describes how to write good documentation strings for your Lisp programs, as well as how to write programs to access documentation.

Note that the documentation strings for Emacs are not the same thing as the Emacs manual. Manuals have their own source files, written in the Texinfo language; documentation strings are specified in the definitions of the functions and variables they apply to. A collection of documentation strings is not sufficient as a manual because a good manual is not organized in that fashion; it is organized in terms of topics of discussion.

23.1 Documentation Basics

A documentation string is written using the Lisp syntax for strings, with double-quote characters surrounding the text of the string. This is because it really is a Lisp string object. The string serves as documentation when it is written in the proper place in the definition of a function or variable. In a function definition, the documentation string follows the argument list. In a variable definition, the documentation string follows the initial value of the variable.

When you write a documentation string, make the first line a complete sentence (or two complete sentences) since some commands, such as `apropos`, show only the first line of a multi-line documentation string. Also, you should not indent the second line of a documentation string, if it has one, because that looks odd when you use *C-h f* (`describe-function`) or *C-h v* (`describe-variable`) to view the documentation string. See Section A.3 [Documentation Tips], page 786.

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 23.2 [Accessing Documentation], page 424.)

In Emacs Lisp, a documentation string is accessible through the function or variable that it describes:

- The documentation for a function is stored in the function definition itself (see Section 11.2 [Lambda Expressions], page 172). The function `documentation` knows how to extract it.
- The documentation for a variable is stored in the variable's property list under the property name `variable-documentation`. The function `documentation-property` knows how to retrieve it.

To save space, the documentation for preloaded functions and variables (including primitive functions and autoloaded functions) is stored in the file '`emacs/etc/DOC-`*version*'—not inside Emacs. The documentation strings for functions and variables loaded during the Emacs session from byte-compiled files are stored in those files (see Section 15.3 [Docs and Compilation], page 226).

The data structure inside Emacs has an integer offset into the file, or a list containing a file name and an integer, in place of the documentation string. The functions `documentation` and `documentation-property` use that information to fetch the documentation string from the appropriate file; this is transparent to the user.

For information on the uses of documentation strings, see section "Help" in *The GNU Emacs Manual*.

The '`emacs/lib-src`' directory contains two utilities that you can use to print nice-looking hardcopy for the file '`emacs/etc/DOC-`*version*'. These are '`sorted-doc`' and '`digest-doc`'.

23.2 Access to Documentation Strings

documentation-property *symbol property* &optional *verbatim* Function

This function returns the documentation string that is recorded in *symbol*'s property list under property *property*. It retrieves the text from a file if necessary, and runs `substitute-command-keys` to substitute actual key bindings. (This substitution is not done 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 *function* &optional *verbatim* Function

This function returns the documentation string of *function*. It reads the text from a file if necessary. Then (unless *verbatim* is non-`nil`) it calls `substitute-command-keys`, 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`.

Here is an example of using the two functions, `documentation` and `documentation-property`, to display the documentation strings for several symbols in a '`*Help*`' buffer.

```
(defun describe-symbols (pattern)
  "Describe the Emacs Lisp symbols matching PATTERN.
All symbols that have PATTERN in their name are described
in the `*Help*' buffer."
  (interactive "sDescribe symbols matching: ")
  (let ((describe-func
         (function
          (lambda (s)
            ;; Print description of symbol.
            (if (fboundp s)             ; It is a function.
                (princ
                 (format "%s\t%s\n%s\n\n" s
                   (if (commandp s)
                       (let ((keys (where-is-internal s)))
                         (if keys
                             (concat
                              "Keys: "
                              (mapconcat 'key-description
                                         keys " "))
                           "Keys: none"))
                     "Function")
                   (or (documentation s)
                       "not documented"))))

            (if (boundp s)              ; It is a variable.
                (princ
                 (format "%s\t%s\n%s\n\n" s
                   (if (user-variable-p s)
                       "Option " "Variable")
                   (or (documentation-property
                         s 'variable-documentation)
                       "not documented")))))))
        sym-list)

    ;; Build a list of symbols that match pattern.
    (mapatoms (function
               (lambda (sym)
                 (if (string-match pattern (symbol-name sym))
                     (setq sym-list (cons sym sym-list))))))

    ;; Display the data.
    (with-output-to-temp-buffer "*Help*"
      (mapcar describe-func (sort sym-list 'string<))
      (print-help-return-message))))
```

The `describe-symbols` function works like `apropos`, but provides more information.

```
(describe-symbols "goal")

---------- Buffer: *Help* ----------
goal-column     Option
*Semipermanent goal column for vertical motion, as set by ...

set-goal-column Keys: C-x C-n
Set the current horizontal position as a goal for C-n and C-p.
Those commands will move to this position in the line moved to
rather than trying to keep the same horizontal position.
With a non-nil argument, clears out the goal column
so that C-n and C-p resume vertical motion.
The goal column is stored in the variable ‘goal-column’.

temporary-goal-column   Variable
Current goal column for vertical motion.
It is the column where point was
at the start of current run of vertical motion commands.
When the ‘track-eol’ feature is doing its job, the value is 9999.
---------- Buffer: *Help* ----------
```

Snarf-documentation *filename* Function

This function is used only during Emacs initialization, just before the runnable Emacs is dumped. It finds the file offsets of the documentation strings stored in the file *filename*, and records them in the in-core function definitions and variable property lists in place of the actual strings. See Section B.1 [Building Emacs], page 793.

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 `data-directory` in Section 23.5 [Help Functions], page 429.

In older Emacs versions, `exec-directory` was used for this.

23.3 Substituting Key Bindings in Documentation

When documentation strings refer to key sequences, they should use the current, actual key bindings. They can do so using certain special text sequences described below. Accessing documentation strings in the usual way substitutes current key binding information for these special sequences. This works by calling `substitute-command-keys`. You can also call that function yourself.

Here is a list of the special sequences and what they mean:

`\[`*command*`]`
: stands for a key sequence that will invoke *command*, or '`M-x` *command*' if *command* has no key bindings.

`\{`*mapvar*`}`
: stands for a summary of the keymap which is the value of the variable *mapvar*. The summary is made using `describe-bindings`.

`\<`*mapvar*`>`
: stands for no text itself. It is used only for a side effect: it specifies *mapvar*'s value as the keymap for any following '`\[`*command*`]`' sequences in this documentation string.

`\=`
: quotes the following character and is discarded; thus, '`\=\[`' puts '`\[`' into the output, and '`\=\=`' puts '`\=`' into the output.

Please note: Each '`\`' must be doubled when written in a string in Emacs Lisp.

substitute-command-keys *string* Function

This function scans *string* for the above special sequences and replaces them by what they stand for, returning the result as a string. This permits display of documentation that refers accurately to the user's own customized key bindings.

Here are examples of the special sequences:

```
(substitute-command-keys
   "To abort recursive edit, type: \\[abort-recursive-edit]")
   "To abort recursive edit, type: C-]"

(substitute-command-keys
   "The keys that are defined for the minibuffer here are:
  \\{minibuffer-local-must-match-map}")
   "The keys that are defined for the minibuffer here are:

?               minibuffer-completion-help
SPC             minibuffer-complete-word
TAB             minibuffer-complete
```

```
C-j             minibuffer-complete-and-exit
RET             minibuffer-complete-and-exit
C-g             abort-recursive-edit
"

(substitute-command-keys
   "To abort a recursive edit from the minibuffer, type\
\\<minibuffer-local-must-match-map>\\[abort-recursive-edit].")
   "To abort a recursive edit from the minibuffer, type C-g."
```

23.4 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* Function

This function returns a string containing the Emacs standard notation for the input events in *sequence*. The argument *sequence* may be a string, vector or list. See Section 20.5 [Input Events], page 330, for more information about valid events. See also the examples for `single-key-description`, below.

single-key-description *event* 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 as itself. An event that is a list appears as the name of the symbol in the CAR of the list.

```
(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 'C-mouse-1)
        "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 lead-

ing caret (which is how control characters in Emacs buffers are usually displayed).

```
(text-char-description ?\C-c)
        "^C"
(text-char-description ?\M-m)
        "M-m"
(text-char-description ?\C-\M-m)
        "M-^M"
```

read-kbd-macro *string* Function

This function is used mainly for operating on keyboard macros, but it can also be used as a rough inverse for `key-description`. You call it with a string containing key descriptions, separated by spaces; it returns a string or vector containing the corresponding events. (This may or may not be a single valid key sequence, depending on what events you use; see Section 21.1 [Keymap Terminology], page 361.)

23.5 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 *regexp* &optional *do-all* Command

This function finds all symbols whose names contain a match for the regular expression *regexp*, and returns a list of them (see Section 33.2 [Regular Expressions], page 649). It also displays the symbols in a buffer named '`*Help*`', each with a one-line description taken from the beginning of its documentation string.

If *do-all* is non-`nil`, then `apropos` also shows key bindings for the functions that are found; it also shows all symbols, even those that are neither functions nor variables.

In the first of the following examples, `apropos` finds all the symbols with names containing '`exec`'. (We don't show here the output that results in the '`*Help*`' buffer.)

```
(apropos "exec")
        (Buffer-menu-execute command-execute exec-directory
    exec-path execute-extended-command execute-kbd-macro
    executing-kbd-macro executing-macro)
```

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 "\C-h" 'help-command)
(fset 'help-command help-map)
```

print-help-return-message &optional *function* Function

This function builds a string that explains how to restore the previous state of the windows after a help command. After building the message, it applies *function* to it if *function* is non-`nil`. Otherwise it calls `message` to display it in the echo area.

This function expects to be called inside a `with-output-to-temp-buffer` special form, and expects `standard-output` to have the value bound by that special form. For an example of its use, see the long example in Section 23.2 [Accessing Documentation], page 424.

help-char Variable

The value of this variable is the help character—the character that Emacs recognizes as meaning Help. By default, its value is 8, which stands for *C-h*. When Emacs reads this character, if `help-form` is a non-`nil` Lisp expression, it evaluates that expression, and displays the result in a window if it is a string.

Usually the value of `help-form` is `nil`. Then the help character has no special meaning at the level of command input, and it becomes part of a key sequence in the normal way. The standard key binding of *C-h* is a prefix key for several general-purpose help features.

The help character is special after prefix keys, too. If it has no binding as a subcommand of the prefix key, it runs `describe-prefix-bindings`, which displays a list of all the subcommands of the prefix key.

help-event-list Variable

The value of this variable is a list of event types that serve as alternative "help characters." These events are handled just like the event specified by `help-char`.

help-form Variable

If this variable is non-`nil`, its value is a form to evaluate whenever the character `help-char` is read. If evaluating the form produces a string, that string is displayed.

A command that calls `read-event` or `read-char` probably should bind `help-form` to a non-`nil` expression while it does input. (The time when you should not do this is when *C-h* has some other meaning.) Evaluating this expression should result in a string that explains what the input is for and how to enter it properly.

Entry to the minibuffer binds this variable to the value of `minibuffer-help-form` (see Section 19.9 [Minibuffer Misc], page 316).

prefix-help-command Variable

This variable holds a function to print help for a prefix key. The function is called when the user types a prefix key followed by the help character, and the help character has no binding after that prefix. The variable's default value is `describe-prefix-bindings`.

describe-prefix-bindings Function

This function calls `describe-bindings` to display a list of all the subcommands of the prefix key of the most recent key sequence. The prefix described consists of all but the last event of that key sequence. (The last event is, presumably, the help character.)

The following two functions are meant for modes that want to provide help without relinquishing control, such as the "electric" modes. Their names begin with '`Helper`' to distinguish them from the ordinary help functions.

Helper-describe-bindings Command

This command pops up a window displaying a help buffer containing a listing of all of the key bindings from both the local and global keymaps. It works by calling `describe-bindings`.

Helper-help Command

This command provides help for the current mode. It prompts the user in the minibuffer with the message '`Help (Type ? for further options)`', and then provides assistance in finding out what the key bindings are, and what the mode is intended for. It returns `nil`.

This can be customized by changing the map `Helper-help-map`.

data-directory Variable

This variable holds the name of the directory in which Emacs finds certain documentation and text files that come with Emacs. In older Emacs versions, `exec-directory` was used for this.

make-help-screen *fname help-line help-text help-map* Macro

This macro defines a help command named *fname* that acts like a prefix key that shows a list of the subcommands it offers.

When invoked, *fname* displays *help-text* in a window, then reads and executes a key sequence according to *help-map*. The string *help-text* should describe the bindings available in *help-map*.

The command *fname* is defined to handle a few events itself, by scrolling the display of *help-text*. When *fname* reads one of those special events, it

does the scrolling and then reads another event. When it reads an event that is not one of those few, and which has a binding in *help-map*, it executes that key's binding and then returns.

The argument *help-line* should be a single-line summary of the alternatives in *help-map*. In the current version of Emacs, this argument is used only if you set the option `three-step-help` to `t`.

This macro is used in the command `help-for-help` which is the binding of `C-h C-h`.

three-step-help User Option

If this variable is non-`nil`, commands defined with `make-help-screen` display their *help-line* strings in the echo area at first, and display the longer *help-text* strings only if the user types the help character again.

24 Files

In Emacs, you can find, create, view, save, and otherwise work with files and file directories. This chapter describes most of the file-related functions of Emacs Lisp, but a few others are described in Chapter 26 [Buffers], page 479, and those related to backups and auto-saving are described in Chapter 25 [Backups and Auto-Saving], page 467.

Many of the file functions take one or more arguments that are file names. A file name is actually a string. Most of these functions expand file name arguments by calling `expand-file-name`, so that '`~`' is handled correctly, as are relative file names (including '`../`'). These functions don't recognize environment variable substitutions such as '`$HOME`'. See Section 24.8.4 [File Name Expansion], page 454.

24.1 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.

24.1.1 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 26.4 [Buffer File Name], page 483, 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 24.3 [Reading from Files], page 439.

find-file *filename* 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.

The body of the `find-file` function is very simple and looks like this:

```
(switch-to-buffer (find-file-noselect filename))
```

(See `switch-to-buffer` in Section 27.7 [Displaying Buffers], page 504.)

When `find-file` is called interactively, it prompts for *filename* in the minibuffer.

find-file-noselect *filename* &optional *nowarn rawfile* Function

This function is the guts of all the file-visiting functions. It finds or creates a buffer visiting the file *filename*, and returns it. It uses an existing buffer if there is one, and otherwise creates a new buffer and reads the file into it. You may make the buffer current or display it in a window if you wish, but this function does not do so.

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, then this function asks the user whether to reread the changed file. If the user says '`yes`', any changes previously made in the buffer are lost.

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 24.1.2 [Subroutines of Visiting], page 435). 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-hooks`.

If the optional argument *rawfile* is non-`nil`, then `after-find-file` is not called, and the `find-file-not-found-hooks` are not run in case of failure. What's more, a non-`nil` *rawfile* value suppresses coding system conversion (see Section 32.10 [Coding Systems], page 636) and format conversion (see Section 24.12 [Format Conversion], page 463).

The `find-file-noselect` function returns the buffer that is visiting the file *filename*.

```
(find-file-noselect "/etc/fstab")
     ⇒ #<buffer fstab>
```

find-file-other-window *filename* 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 27.7 [Displaying Buffers], page 504.

When this command is called interactively, it prompts for *filename*.

find-file-read-only *filename* Command
This command selects a buffer visiting the file *filename*, like `find-file`, but it marks the buffer as read-only. See Section 26.7 [Read Only Buffers], page 487, for related functions and variables.

When this command is called interactively, it prompts for *filename*.

view-file *filename* Command
This command visits *filename* using View mode, returning to the previous buffer when you exit View mode. View mode is a minor mode that provides commands to skim rapidly through the file, but does not let you modify the text. Entering View mode runs the normal hook `view-mode-hook`. See Section 22.6 [Hooks], page 420.

When `view-file` is called interactively, it prompts for *filename*.

find-file-hooks Variable
The value of this variable is a list of functions to be called after a file is visited. The file's local-variables specification (if any) will have been processed before the hooks are run. The buffer visiting the file is current when the hook functions are run.

This variable works just like a normal hook, but we think that renaming it would not be advisable. See Section 22.6 [Hooks], page 420.

find-file-not-found-hooks 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.

24.1.2 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 26.9 [Creating Buffers], page 490.

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 26.9 [Creating Buffers], page 490).

after-find-file &optional *error warn* Function

This function sets the buffer major mode, and parses local variables (see Section 22.1.3 [Auto Major Mode], page 398). It is called by `find-file-noselect` and by the default revert function (see Section 25.3 [Reverting], page 475).

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.

The last thing `after-find-file` does is call all the functions in the list `find-file-hooks`.

24.2 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.

save-some-buffers &optional *save-silently-p exiting* 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 *exiting* argument, if non-`nil`, requests this function to offer also to save certain other buffers that are not visiting files. These are buffers that have a non-`nil` buffer-local value of `buffer-offer-save`. (A user who says yes to saving one of these is asked to specify a file name to use.) The `save-buffers-kill-emacs` function passes a non-`nil` value for this argument.

write-file *filename* 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 26.4 [Buffer File Name], page 483) and `save-buffer`.

Saving a buffer runs several hooks. It also performs format conversion (see Section 24.12 [Format Conversion], page 463), and may save text properties in "annotations" (see Section 31.19.7 [Saving Properties], page 619).

write-file-hooks 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-hooks` 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 to set the mode bits of the file that you write. This is what `save-buffer` normally does.

The hook functions in `write-file-hooks` are also responsible for encoding the data (if desired): they must choose a suitable coding system (see Section 32.10.3 [Lisp and Coding Systems], page 638), perform the encoding (see Section 32.10.7 [Explicit Encoding], page 643), and set `last-coding-system-used` to the coding system that was used (see Section 32.10.2 [Encoding and I/O], page 637).

Do not make this variable buffer-local. To set up buffer-specific hook functions, use `write-contents-hooks` instead.

Even though this is not a normal hook, you can use `add-hook` and `remove-hook` to manipulate the list. See Section 22.6 [Hooks], page 420.

local-write-file-hooks Variable

This works just like `write-file-hooks`, but it is intended to be made buffer-local in particular buffers, and used for hooks that pertain to the file name or the way the buffer contents were obtained.

The variable is marked as a permanent local, so that changing the major mode does not alter a buffer-local value. This is convenient for packages that read "file" contents in special ways, and set up hooks to save the data in a corresponding way.

write-contents-hooks Variable

This works just like `write-file-hooks`, but it is intended for hooks that pertain to the contents of the file, as opposed to hooks that pertain to where the file came from. 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. When you use `add-hooks` to add an element to this hook, you should *not* specify a non-`nil` *local* argument, since this variable is used *only* buffer-locally.

after-save-hook Variable

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 Variable

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 25.1.2 [Rename or Copy], page 468. Yet, at the same time, saving a precious file always breaks all hard links between the file you save and other file names.

Some modes give this variable a non-`nil` buffer-local value in particular buffers.

require-final-newline User Option

This variable determines whether files may be written out that do *not* end with a newline. If the value of the variable is `t`, then `save-buffer` silently adds a newline at the end of the file whenever the buffer being saved does not already end in one. If the value of the variable is non-`nil`, but not `t`, then `save-buffer` asks the user whether to add a newline each time the case arises.

If the value of the variable is `nil`, then `save-buffer` doesn't add newlines at all. `nil` is the default value, but a few major modes set it to `t` in particular buffers.

See also the function `set-visited-file-name` (see Section 26.4 [Buffer File Name], page 483).

24.3 Reading from Files

You can copy a file from the disk and insert it into a buffer using the `insert-file-contents` function. Don't use the user-level command `insert-file` in a Lisp program, as that sets the mark.

insert-file-contents *filename* &optional *visit beg end replace* Function

This function inserts the contents of file *filename* into the current buffer after point. It returns a list of the absolute file name and the length of the data inserted. An error is signaled if *filename* is not the name of a file that can be read.

The function `insert-file-contents` checks the file contents against the defined file formats, and converts the file contents if appropriate. See Section 24.12 [Format Conversion], page 463. It also calls the functions in the list `after-insert-file-functions`; see Section 31.19.7 [Saving Properties], page 619.

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 do format decoding (see Section 24.12 [Format Conversion], page 463), does not do character code conversion (see Section 32.10 [Coding Systems], page 636), does not run `find-file-hooks`, does not perform automatic uncompression, and so on.

If you want to pass a file name to another process so that another program can read the file, use the function `file-local-copy`; see Section 24.11 [Magic File Names], page 460.

24.4 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.

write-region *start end filename* &optional *append visit* *confirm* Command

This function writes the region delimited by *start* and *end* in the current buffer into the file specified by *filename*.

If *start* is a string, then `write-region` writes or appends that string, rather than text from the buffer.

If *append* is non-`nil`, then the specified text is appended to the existing file contents (if any).

If *confirm* is non-`nil`, then `write-region` asks for confirmation if *filename* names an existing file.

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 function `write-region` converts the data which it writes to the appropriate file formats specified by `buffer-file-format`. See Section 24.12 [Format Conversion], page 463. It also calls the functions in the list `write-region-annotate-functions`; see Section 31.19.7 [Saving Properties], page 619.

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 9.5 [Nonlocal Exits], page 135).

See also `with-temp-buffer` in Section 26.2 [Current Buffer], page 479.

24.5 File Locks

When two users edit the same file at the same time, they are likely to interfere with each other. Emacs tries to prevent this situation from arising by recording a *file lock* when a file is being modified. Emacs can then detect the first attempt to modify a buffer visiting a file that is locked by another Emacs job, and ask the user what to do.

File locks are not completely reliable when multiple machines can share file systems. When file locks do not work, it is possible for two users to make changes simultaneously, but Emacs can 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 26.6 [Modification Time], page 486.

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.

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.

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`'.

24.6 Information about Files

The functions described in this section all operate on strings that designate file names. 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.

24.6.1 Testing Accessibility

These functions test for permission to access a file in specific ways.

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, this is true if the file exists and you have execute permission on the containing directories, regardless of the protection of the file itself.)

If the file does not exist, or if fascist access control policies prevent you from finding the attributes of the file, this function returns `nil`.

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. 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 or the file name of a file which is a directory.

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.

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 *filename2* does not exist, 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 24.6.4 [File Attributes], page 446.

24.6.2 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 file name to which it is linked. This may be the name of a text file, a directory, or even another symbolic link, or it may be a nonexistent file name.

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"
```

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, symbolic link, named pipe, terminal, or other I/O device).

24.6.3 Truenames

The *truename* of a file is the name that you get by following symbolic links until none remain, then simplifying away '`.`' and '`..`' appearing as components. Strictly speaking, a file need not have a unique truename; the number of distinct truenames a file has is equal to the number of hard links to the file. However, truenames are useful because they eliminate symbolic links as a cause of name variation.

file-truename *filename* Function

The function `file-truename` returns the true name of the file *filename*. This is the name that you get by following symbolic links until none remain. The argument must be an absolute file name.

See Section 26.4 [Buffer File Name], page 483, for related information.

24.6.4 Other Information about Files

This section describes the functions for getting detailed information about a file, other than its contents. This information includes the mode bits that control access permission, the owner and group numbers, the number of names, the inode number, the size, and the times of access and modification.

file-modes *filename* Function

This function returns the mode bits of *filename*, as an integer. The mode bits are also called the file permissions, and they specify access control in the usual Unix fashion. If the low-order bit is 1, then the file is executable by all users, if the second-lowest-order bit is 1, then the file is writable by all users, etc.

The highest value returnable is 4095 (7777 octal), meaning that everyone has read, write, and execute permission, that the SUID bit is set for both others and group, and that the sticky bit is set.

```
(file-modes "~/junk/diffs")
     ⇒ 492                ; Decimal integer.
(format "%o" 492)
     ⇒ "754"              ; Convert to octal.

(set-file-modes "~/junk/diffs" 438)
     ⇒ nil

(format "%o" 438)
     ⇒ "666"              ; Convert to octal.

% ls -l diffs
  -rw-rw-rw-  1 lewis 0 3063 Oct 30 16:00 diffs
```

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* Function

This function returns a list of attributes of file *filename*. If the specified file cannot be opened, it returns `nil`.

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 24.7 [Changing Files], page 448).
2. The file's UID.
3. The file's GID.
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 37.5 [Time of Day], page 723.)
5. The time of last modification as a list of two integers (as above).
6. The time of last status change as a list of two integers (as above).
7. The size of the file in bytes.
8. The file's modes, as a string of ten letters or dashes, as in '`ls -l`'.
9. `t` if the file's GID would change if file were deleted and recreated; `nil` otherwise.
10. The file's inode number. If possible, this is an integer. If the inode number is too large to be represented as an integer in Emacs Lisp, then the value has the form (*high* . *low*), where *low* holds the low 16 bits.
11. The file system number of the file system that the file is in. 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")
     ⇒  (nil 1 2235 75
         (8489 20284)
         (8489 20284)
         (8489 20285)
         14906 "-rw-rw-rw-"
         nil 129500 -32252)
```

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).

`2235` is owned by the user with UID 2235.

`75` is in the group with GID 75.

`(8489 20284)`
was last accessed on Aug 19 00:09.

`(8489 20284)`
was last modified on Aug 19 00:09.

`(8489 20285)`
last had its inode changed on Aug 19 00:09.

`14906` is 14906 characters long.

`"-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.

`129500` has an inode number of 129500.

`-32252` is on file system number -32252.

24.7 Changing File Names and Attributes

The functions in this section rename, copy, delete, link, and set the modes of files.

In the functions that have an argument *newname*, if a file by the name of *newname* already exists, the actions taken depend on the value of the argument *ok-if-already-exists*:

- Signal a `file-already-exists` error if *ok-if-already-exists* is `nil`.
- Request confirmation if *ok-if-already-exists* is a number.
- Replace the old file without confirmation if *ok-if-already-exists* is any other value.

add-name-to-file *oldname newname* &optional *ok-if-already-exists* Function

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.

See also `file-nlinks` in Section 24.6.4 [File Attributes], page 446.

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.

In an interactive call, this function prompts for *filename* and *newname* in the minibuffer; also, it requests confirmation if *newname* already exists.

copy-file *oldname newname* &optional *ok-if-exists time* Command

This command copies the file *oldname* to *newname*. An error is signaled if *oldname* does not exist.

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, this function prompts for *filename* and *newname* in the minibuffer; also, it requests confirmation if *newname* already exists.

delete-file *filename* Command

This command deletes the file *filename*, like the shell command '`rm` *filename*'. If the file has multiple names, it continues to exist under the other names.

A suitable kind of `file-error` error is signaled if the file does not exist, or is not deletable. (On Unix, a file is deletable if its directory is writable.)

See also `delete-directory` in Section 24.10 [Create/Delete Dirs], page 460.

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*'.

In an interactive call, this function prompts for *filename* and *newname* in the minibuffer; also, it requests confirmation if *newname* already exists.

define-logical-name *varname string* Function

This function defines the logical name *name* to have the value *string*. It is available only on VMS.

set-file-modes *filename mode* Function

This function sets mode bits of *filename* to *mode* (which must be an integer). Only the low 12 bits of *mode* are used.

set-default-file-modes *mode* Function

This function sets the default file protection for new files created by Emacs and its subprocesses. Every file created with Emacs initially has this protection. On Unix, the default protection is the bitwise complement of the "umask" value.

The argument *mode* must be an integer. On most systems, only the low 9 bits of *mode* are meaningful.

Saving a modified version of an existing file does not count as creating the file; it does not change the file's mode, and does not use the default file protection.

default-file-modes Function

This function returns the current default protection value.

On MS-DOS, there is no such thing as an "executable" file mode bit. So Emacs considers a file executable if its name ends in '`.com`', '`.bat`' or '`.exe`'. This is reflected in the values returned by `file-modes` and `file-attributes`.

24.8 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 VMS, all these functions understand both VMS file-name syntax and Unix syntax. This is so that all the standard Lisp libraries can specify file names in Unix syntax and work properly on VMS without change. On MS-DOS and MS-Windows, these functions understand MS-DOS or MS-Windows file-name syntax as well as Unix syntax.

24.8.1 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 Unix, the directory part is everything up to and including the last slash; the nondirectory part is the rest. The rules in VMS syntax are complicated.

For some purposes, the nondirectory part is further subdivided into the name proper and the *version number*. On Unix, only backup files have version numbers in their names. On VMS, every file has a version number, but most of the time the file name actually used in Emacs omits the version number, so that version numbers in Emacs are found mostly in directory lists.

file-name-directory *filename* Function

This function returns the directory part of *filename* (or `nil` if *filename* does not include a directory part). On Unix, the function returns a string ending in a slash. On VMS, it returns a string ending in one of the three characters ':', ']', or '>'.

```
(file-name-directory "lewis/foo")  ; Unix example
     ⇒ "lewis/"
(file-name-directory "foo")        ; Unix example
     ⇒ nil
```

```
(file-name-directory "[X]FOO.TMP") ; VMS example
     ⇒ "[X]"
```

file-name-nondirectory *filename* Function

This function returns the nondirectory part of *filename*.

```
(file-name-nondirectory "lewis/foo")
     ⇒ "foo"
(file-name-nondirectory "foo")
     ⇒ "foo"
;; The following example is accurate only on VMS.
(file-name-nondirectory "[X]FOO.TMP")
     ⇒ "FOO.TMP"
```

file-name-sans-versions *filename* Function

This function returns *filename* with any file version numbers, backup version numbers, or trailing tildes deleted.

```
(file-name-sans-versions "~rms/foo.~1~")
     ⇒ "~rms/foo"
(file-name-sans-versions "~rms/foo~")
     ⇒ "~rms/foo"
(file-name-sans-versions "~rms/foo")
     ⇒ "~rms/foo"
;; The following example applies to VMS only.
(file-name-sans-versions "foo;23")
     ⇒ "foo"
```

file-name-sans-extension *filename* Function

This function returns *filename* minus its "extension," if any. The extension, in a file name, is the part that starts with the last '.' in the last name component. For example,

```
(file-name-sans-extension "foo.lose.c")
     ⇒ "foo.lose"
(file-name-sans-extension "big.hack/foo")
     ⇒ "big.hack/foo"
```

24.8.2 Directory Names

A *directory name* is the name of a directory. A directory is a kind of file, and 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 Unix, this is simple: a directory name ends in a slash, whereas the directory's name as a file lacks that slash. On VMS, the relationship is more complicated.

The difference between a directory name and its name as a file is subtle but crucial. When an Emacs variable or function argument is described as being a directory name, a file name of a directory is not acceptable.

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. In Unix, this means appending a slash to the string (if it does not already end in one). On VMS, the function converts a string of the form '`[X]Y.DIR.1`' to the form '`[X.Y]`'.

```
(file-name-as-directory "~rms/lewis")
     ⇒ "~rms/lewis/"
```

directory-file-name *dirname* Function

This function returns a string representing *dirname* in a form that the operating system will interpret as the name of a file. On Unix, this means removing the final slash from the string. On VMS, the function converts a string of the form '`[X.Y]`' to '`[X]Y.DIR.1`'.

```
(directory-file-name "~lewis/")
     ⇒ "~lewis"
```

Directory name abbreviations are useful for directories that are normally accessed through symbolic links. Sometimes the users recognize primarily the link's name as "the name" of the directory, and find it annoying to see the directory's "real" name. If you define the link name as an abbreviation for the "real" name, Emacs shows users the abbreviation instead.

directory-abbrev-alist Variable

The variable `directory-abbrev-alist` contains an alist of abbreviations to use for file directories. Each element has the form (*from* . *to*), and says to replace *from* with *to* when it appears in a directory name. The *from* string is actually a regular expression; it should always start with '`^`'. The function `abbreviate-file-name` performs these substitutions.

You can set this variable in '`site-init.el`' to describe the abbreviations appropriate for your site.

Here's an example, from a system on which file system '`/home/fsf`' and so on are normally accessed through symbolic links named '`/fsf`' and so on.

```
(("^/home/fsf" . "/fsf")
 ("^/home/gp" . "/gp")
 ("^/home/gd" . "/gd"))
```

To convert a directory name to its abbreviation, use this function:

abbreviate-file-name *dirname* Function
This function applies abbreviations from `directory-abbrev-alist` to its argument, and substitutes '`~`' for the user's home directory.

24.8.3 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, an absolute file name starts with a slash or a tilde ('`~`'), and a relative one does not. The rules on VMS are complicated.

file-name-absolute-p *filename* Function
This function returns `t` if file *filename* is an absolute file name, `nil` otherwise. On VMS, this function understands both Unix syntax and VMS syntax.

```
(file-name-absolute-p "~rms/foo")
     ⇒ t
(file-name-absolute-p "rms/foo")
     ⇒ nil
(file-name-absolute-p "/user/rms/foo")
     ⇒ t
```

24.8.4 Functions that Expand Filenames

Expansion of a file name means converting a relative file name to an absolute one. Since this is done relative to a default directory, you must specify the default directory name as well as the file name to be expanded. Expansion also simplifies file names by eliminating redundancies such as '`./`' and '*name*`/../`'.

expand-file-name *filename* &optional *directory* Function
This function converts *filename* to an absolute file name. If *directory* is supplied, it is the default directory to start with if *filename* is relative. (The value of *directory* should itself be an absolute directory name; 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"
```

Filenames containing '`.`' or '`..`' are simplified to their canonical form:

```
(expand-file-name "bar/../foo")
     ⇒ "/xcssun/users/rms/lewis/foo"
```

Note that `expand-file-name` does *not* expand environment variables; only `substitute-in-file-name` does that.

file-relative-name *filename directory* Function

This function does the inverse of expansion—it tries to return a relative name that is equivalent to *filename* when interpreted relative to *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"
```

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`.

On Unix systems, 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 variables references in *filename* with the environment variable values. Following standard Unix shell syntax, '`$`' is the prefix to substitute an environment variable value.

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 '`}`'.

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 following a '`/`', everything before the following '`/`' is discarded:

```
(substitute-in-file-name "bar/~/foo")
     ⇒ "~/foo"
(substitute-in-file-name "/usr/local/$HOME/foo")
     ⇒ "/xcssun/users/rms/foo"
     ;; '/usr/local/' has been discarded.
```

On VMS, '`$`' substitution is not done, so this function does nothing on VMS except discard superfluous initial components as shown above.

24.8.5 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-name
 (expand-file-name name-of-application
                   temporary-file-directory))
```

The job of `make-temp-name` is to prevent two different users or two different jobs from trying to use the exact same file name. This example uses the variable `temporary-file-directory` to decide where to put the temporary file. All Emacs Lisp programs should use `temporary-file-directory` for this purpose, to give the user a uniform way to specify the directory for all temporary files.

make-temp-name *string* Function

This function generates a string that can be used as a unique file name. The name starts with *string*, and contains a number that is different in each Emacs job.

```
(make-temp-name "/tmp/foo")
     ⇒ "/tmp/foo232J6v"
```

To prevent conflicts among different libraries running in the same Emacs, each Lisp program that uses `make-temp-name` should have its own *string*. The number added to the end of *string* 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.

temporary-file-directory Variable

This variable specifies the directory name for creating temporary files. Its value should be a directory name (see Section 24.8.2 [Directory Names], page 452), 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; on GNU and Unix systems it is based on the `TMP` and `TMPDIR` environment variables.

Even if you do not use `make-temp-name` to choose the temporary file's name, you should still use this variable to decide which directory to put the file in.

24.8.6 File Name Completion

This section describes low-level subroutines for completing a file name. For other completion functions, see Section 19.5 [Completion], page 301.

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. 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* 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 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 or when a buffer showing all possible completions is displayed.

A typical value might look like this:

```
completion-ignored-extensions
     ⇒ (".o" ".elc" "~" ".dvi")
```

24.8.7 Standard File Names

Most of the file names used in Lisp programs are entered by the user. But occasionally a Lisp program needs to specify a standard file name for a particular use—typically, to hold customization information about each user. For example, abbrev definitions are stored (by default) in the file '`~/.abbrev_defs`'; the `completion` package stores completions in the file '`~/.completions`'. These are two of the many standard file names used by parts of Emacs for certain purposes.

Various operating systems have their own conventions for valid file names and for which file names to use for user profile data. A Lisp program which reads a file using a standard file name ought to use, on each type of system, a file name suitable for that system. The function `convert-standard-filename` makes this easy to do.

convert-standard-filename *filename* Function

This function alters the file name *filename* to fit the conventions of the operating system in use, and returns the result as a new string.

The recommended way to specify a standard file name in a Lisp program is to choose a name which fits the conventions of GNU and Unix systems, usually with a nondirectory part that starts with a period, and pass it to `convert-standard-filename` instead of using it directly. Here is an example from the `completion` package:

```
(defvar save-completions-file-name
        (convert-standard-filename "~/.completions")
  "*The file name to save completions to.")
```

On GNU and Unix systems, and on some other systems as well, `convert-standard-filename` returns its argument unchanged. On some other systems, it alters the name to fit the system's conventions.

For example, on MS-DOS the alterations made by this function include converting a leading '.' to '_', converting a '_' in the middle of the name to '.' if there is no other '.', inserting a '.' after eight characters if there is none, and truncating to three characters after the '.'. (It makes other changes as well.) Thus, '`.abbrev_defs`' becomes '`_abbrev.def`', and '`.completions`' becomes '`_complet.ion`'.

24.9 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.

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.

file-name-all-versions *file dirname* Function

This function returns a list of all versions of the file named *file* in directory *dirname*.

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.

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.)

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.

insert-directory-program Variable

This variable's value is the program to run to generate a directory listing for the function `insert-directory`.

24.10 Creating and Deleting Directories

Most Emacs Lisp file-manipulation functions get errors when used on files that are directories. For example, you cannot delete a directory with `delete-file`. These special functions exist to create and delete directories.

make-directory *dirname* Function

This function creates a directory named *dirname*.

delete-directory *dirname* Function

This function deletes the directory named *dirname*. The function `delete-file` does not work for files that are directories; you must use `delete-directory` for them. If the directory contains any files, `delete-directory` signals an error.

24.11 Making Certain File Names "Magic"

You can implement special handling for certain file names. This is called making those names *magic*. The principal use for this feature is in implementing remote file names (see section "Remote Files" in *The GNU Emacs Manual*).

To define a kind of magic file name, you must supply a regular expression to define the class of names (all those that match the regular expression),

plus a handler that implements all the primitive Emacs file operations for file names that do match.

The variable `file-name-handler-alist` holds a list of handlers, together with regular expressions that determine when to apply each handler. Each element has this form:

(*regexp* . *handler*)

All the Emacs primitives for file access and file name transformation check the given file name against `file-name-handler-alist`. If the file name matches *regexp*, the primitives handle that file by calling *handler*.

The first argument given to *handler* is the name of the primitive; the remaining arguments are the arguments that were passed to that operation. (The first of these arguments is typically 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*)

Here are the operations that a magic file name handler gets to handle:

`add-name-to-file`, `copy-file`, `delete-directory`, `delete-file`, `diff-latest-backup-file`, `directory-file-name`, `directory-files`, `dired-call-process`, `dired-compress-file`, `dired-uncache`, `expand-file-name`, `file-accessible-directory-p`, `file-attributes`, `file-directory-p`, `file-executable-p`, `file-exists-p`, `file-local-copy`, `file-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-symbolic-link`, `rename-file`, `set-file-modes`, `set-visited-file-modtime`, `shell-command`, `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.

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. The operation is needed for comparison with `inhibit-file-name-operation`.

file-local-copy *filename* Function

This function copies file *filename* to an ordinary non-magic file, if it isn't one already.

If *filename* specifies a magic file name, which programs outside Emacs cannot directly read or write, this copies the contents to an ordinary file and returns that file's name.

If *filename* is an ordinary file name, not magic, then this function does nothing and returns `nil`.

unhandled-file-name-directory *filename* Function

This function returns the name of a directory that is not magic. It uses the directory part of *filename* if that is not magic. For a magic file name,

it invokes the file name handler, which therefore decides what value to return.

This is useful for running a subprocess; every subprocess must have a non-magic directory to serve as its current directory, and this function is a good way to come up with one.

24.12 File Format Conversion

The variable `format-alist` defines a list of *file formats*, which describe textual representations used in files for the data (text, text-properties, and possibly other information) in an Emacs buffer. Emacs performs format conversion if appropriate when reading and writing files.

format-alist Variable

This list contains one format definition for each defined file format.

Each format definition is a list of this form:

(*name doc-string regexp from-fn to-fn modify mode-fn*)

Here is what the elements in a format definition mean:

name The name of this format.

doc-string A documentation string for the format.

regexp A regular expression which is used to recognize files represented in this format.

from-fn A shell command or function to decode data in this format (to convert file data into the usual Emacs data representation).

A shell command is represented as a string; Emacs runs the command as a filter to perform the conversion.

If *from-fn* is a function, it is called with two arguments, *begin* and *end*, which specify the part of the buffer it should convert. It should convert the text by editing it in place. Since this can change the length of the text, *from-fn* should return the modified end position.

One responsibility of *from-fn* is to make sure that the beginning of the file no longer matches *regexp*. Otherwise it is likely to get called again.

to-fn A shell command or function to encode data in this format—that is, to convert the usual Emacs data representation into this format.

If *to-fn* is a string, it is a shell command; Emacs runs the command as a filter to perform the conversion.

If *to-fn* is a function, it is called with two arguments, *begin* and *end*, which specify the part of the buffer it should convert. 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 A mode function to call after visiting a file converted from this 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* Command

This command writes the current buffer contents into the file *file* in format *format*, and makes that format the default for future saves of the buffer. The argument *format* is a list of format names.

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 24.3 [Reading from Files], page 439).

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`.

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. This variable is always buffer-local in all buffers.

25 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.

25.1 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. Normally, this means that 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.

25.1.1 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.

buffer-backed-up Variable

This buffer-local variable indicates whether this buffer's file has been backed up on account of this buffer. If it is non-`nil`, then 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-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 '`.emacs`' file.)

```
(add-hook 'rmail-mode-hook
          (function (lambda ()
                      (make-local-variable
                       'make-backup-files)
                      (setq make-backup-files nil))))
```

backup-enable-predicate Variable

This variable's value is a function to be called on certain occasions to decide whether a file should have backup files. The function receives one argument, a 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 this:

```
(lambda (name)
  (or (< (length name) 5)
      (not (string-equal "/tmp/"
                         (substring name 0 5)))))
```

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.

25.1.2 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 24.2 [Saving Buffers], page 436.

backup-by-copying Variable

If this variable is non-`nil`, Emacs always makes backup files by copying.

The following two 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 Variable

If this variable is non-`nil`, Emacs makes backups by copying for files with multiple names (hard links).

This variable is significant only if `backup-by-copying` is `nil`, since copying is always used when that variable is non-`nil`.

backup-by-copying-when-mismatch Variable

If this variable is non-`nil`, Emacs makes backups by copying in cases where renaming would change either the owner or the group of the file.

The value has no effect when renaming would not alter the owner or group of the file; that is, for files which are owned by the user and whose group matches the default for a new file created there by the user.

This variable is significant only if `backup-by-copying` is `nil`, since copying is always used when that variable is non-`nil`.

25.1.3 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.

`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 25.1.4 [Backup Names], page 470) is responsible for determining which backup versions to delete, but does not delete them itself.

delete-old-versions User Option
If this variable is non-`nil`, then saving a file deletes excess backup versions silently. Otherwise, it asks the user whether to delete them.

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 value is 2.

25.1.4 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. A file with the name *filename* need not exist; the function just checks the name.

```
(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.

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 25.1.3 [Numbered Backups], page 469.

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.

25.2 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/#files.texi#"
```

auto-save-mode *arg* Command

When used interactively without an argument, this command is a toggle switch: it turns on auto-saving of the current buffer if it is off, and vice-versa. With an argument *arg*, the command turns auto-saving on if the value of *arg* is `t`, a nonempty list, or a positive integer. Otherwise, it turns auto-saving off.

auto-save-file-name-p *filename* Function

This function returns a non-`nil` value if *filename* is a string that could be the name of an auto-save file. It works based on knowledge of the 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/#files.texi#"
```

```
(auto-save-file-name-p "#files.texi#")
     ⇒ 0
(auto-save-file-name-p "files.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 (`#`) appended and prepended to it. This function does not look at the variable `auto-save-visited-file-name` (described below); you should check that before calling this function.

```
(make-auto-save-file-name)
     ⇒ "/xcssun/users/rms/lewis/#backup.texi#"
```

The standard definition of this function is as follows:

```
(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 Variable

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 value does not take effect until the next time auto-save mode is reenabled in any given buffer. 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 is the number of characters that Emacs reads from the keyboard between auto-saves. Each time this many more characters are read, auto-saving is done for all buffers in which it is enabled.

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 auto-saves any buffers that need it. (Actually, the specified timeout is multiplied by a factor depending on the size of the current buffer.)

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.

Normally, if any buffers are auto-saved, a message that says '`Auto-saving...`' is displayed 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 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.

delete-auto-save-files Variable

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 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 as of the last time it was 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 deletion. Explicitly saving the buffer stores a positive value in this variable, thus reenabling auto-saving. Turning auto-save mode off or on also alters this variable.

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.

If 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 these files.

The default name for this file is in your home directory and starts with '`.saves-`'. It also contains the Emacs process ID and the host name.

25.3 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* 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, `revert-buffer` asks the user whether to use that instead. But if

the argument *ignore-auto* is non-`nil`, then only the the visited file itself is used. Interactively, *ignore-auto* is `t` unless there is a numeric prefix argument; thus, the interactive default is 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.

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; then it preserves the markers in the unchanged text (if any) at the beginning and end of the buffer. Preserving any additional markers would be problematical.

You can customize how `revert-buffer` does its work by setting these variables—typically, as buffer-local variables.

revert-without-query Variable

This variable holds a list of files that should be reverted without query. The value is a list of regular expressions. If a file name matches one of these regular expressions, then `revert-buffer` reverts the file without asking the user for confirmation, if the file has changed on disk and the buffer is not modified.

revert-buffer-function Variable

The value of this variable is the function to use to revert this buffer. If non-`nil`, it is called as a function with no arguments to do the work of reverting. 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, 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`, is 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.

before-revert-hook Variable

This normal hook is run by `revert-buffer` before actually inserting the modified contents—but only if `revert-buffer-function` is `nil`.

Font Lock mode uses this hook to record that the buffer contents are no longer fontified.

after-revert-hook Variable

This normal hook is run by `revert-buffer` after actually inserting the modified contents—but only if `revert-buffer-function` is `nil`.

Font Lock mode uses this hook to recompute the fonts for the updated buffer contents.

26 Buffers

A *buffer* is a Lisp object containing text to be edited. Buffers are used to hold the contents of files that are being visited; there may also be buffers that are not visiting files. While several buffers may exist at one time, exactly one buffer is designated the *current buffer* at any time. Most editing commands act on the contents of the current buffer. Each buffer, including the current buffer, may or may not be displayed in any windows.

26.1 Buffer Basics

Buffers in Emacs editing are objects that have distinct names and hold text that can be edited. Buffers appear to Lisp programs as a special data type. You can think of the contents of a buffer as a string that you can extend; insertions and deletions may occur in any part of the buffer. See Chapter 31 [Text], page 575.

A Lisp buffer object contains numerous pieces of information. Some of this information is directly accessible to the programmer through variables, while other information is accessible only through special-purpose functions. For example, the visited file name is directly accessible through a variable, while the value of point is accessible only through a primitive function.

Buffer-specific information that is directly accessible is stored in *buffer-local* variable bindings, which are variable values that are effective only in a particular buffer. This feature allows each buffer to override the values of certain variables. Most major modes override variables such as `fill-column` or `comment-column` in this way. For more information about buffer-local variables and functions related to them, see Section 10.10 [Buffer-Local Variables], page 161.

For functions and variables related to visiting files in buffers, see Section 24.1 [Visiting Files], page 433 and Section 24.2 [Saving Buffers], page 436. For functions and variables related to the display of buffers in windows, see Section 27.6 [Buffers and Windows], page 503.

bufferp *object* Function

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

26.2 The Current Buffer

There are, in general, many buffers in an Emacs session. At any time, one of them is designated as the *current buffer*. This is the buffer in which most editing takes place, because most of the primitives for examining or changing text in a buffer operate implicitly on the current buffer (see Chapter 31 [Text], page 575). Normally the buffer that is displayed on the screen in the selected window is the current buffer, but this is not always so: a Lisp

program can temporarily designate any buffer as current in order to operate on its contents, without changing what is displayed on the screen.

The way to designate a current buffer in a Lisp program is by calling `set-buffer`. The specified buffer remains current until a new one is designated.

When an editing command returns to the editor command loop, the command loop designates the buffer displayed in the selected window as current, to prevent confusion: the buffer that the cursor is in when Emacs reads a command is the buffer that the command will apply to. (See Chapter 20 [Command Loop], page 319.) Therefore, `set-buffer` is not the way to switch visibly to a different buffer so that the user can edit it. For this, you must use the functions described in Section 27.7 [Displaying Buffers], page 504.

However, Lisp functions that change to a different current buffer should not depend on the command loop to set it back afterwards. Editing commands written in Emacs Lisp can be called from other programs as well as from the command loop. It is convenient for the caller if the subroutine does not change which buffer is current (unless, of course, that is the subroutine's purpose). Therefore, you should normally use `set-buffer` within a `save-current-buffer` or `save-excursion` (see Section 29.3 [Excursions], page 560) form that will restore the current buffer when your function is done. Here is an example, the code for the command `append-to-buffer` (with the documentation string abridged):

```
(defun append-to-buffer (buffer start end)
  "Append to specified buffer the text of the region.
..."
  (interactive "BAppend to buffer: \nr")
  (let ((oldbuf (current-buffer)))
    (save-current-buffer
      (set-buffer (get-buffer-create buffer))
      (insert-buffer-substring oldbuf start end))))
```

This function binds a local variable to record the current buffer, and then `save-current-buffer` arranges to make it current again. Next, `set-buffer` makes the specified buffer current. Finally, `insert-buffer-substring` copies the string from the original current buffer to the specified (and now current) buffer.

If the buffer appended to happens to be displayed in some window, the next redisplay will show how its text has changed. Otherwise, you will not see the change immediately on the screen. The buffer becomes current temporarily during the execution of the command, but this does not cause it to be displayed.

If you make local bindings (with `let` or function arguments) for a variable that may also have buffer-local bindings, make sure that the same buffer is current at the beginning and at the end of the local binding's scope. Otherwise you might bind it in one buffer and unbind it in another! There are two ways to do this. In simple cases, you may see that nothing ever changes

the current buffer within the scope of the binding. Otherwise, use `save-current-buffer` or `save-excursion` to make sure that the buffer current at the beginning is current again whenever the variable is unbound.

It is not reliable to change the current buffer back with `set-buffer`, because that won't do the job if a quit happens while the wrong buffer is current. Here is what *not* to do:

```
(let (buffer-read-only
      (obuf (current-buffer)))
  (set-buffer ...)
  ...
  (set-buffer obuf))
```

Using `save-current-buffer`, as shown here, handles quitting, errors, and `throw`, as well as ordinary evaluation.

```
(let (buffer-read-only)
  (save-current-buffer
    (set-buffer ...)
    ...))
```

current-buffer Function

This function returns the current buffer.

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

set-buffer *buffer-or-name* Function

This function makes *buffer-or-name* the current buffer. It does not display the buffer in the currently selected window or in any other window, so the user cannot necessarily see the buffer. But Lisp programs can in any case work on it.

This function returns the buffer identified by *buffer-or-name*. An error is signaled if *buffer-or-name* does not identify an existing buffer.

save-current-buffer *body...* Special Form

The `save-current-buffer` macro saves the identity of the current buffer, evaluates the *body* forms, and finally restores that buffer as current. The return value is the value of the last form in *body*. The current buffer is restored even in case of an abnormal exit via `throw` or error (see Section 9.5 [Nonlocal Exits], page 135).

If the buffer that used to be current has been killed by the time of exit from `save-current-buffer`, then it is not made current again, of course. Instead, whichever buffer was current just before exit remains current.

with-current-buffer *buffer body...* Macro

The `with-current-buffer` macro saves the identity of the current buffer, makes *buffer* current, evaluates the *body* forms, and finally restores the

buffer. The return value is the value of the last form in *body*. The current buffer is restored even in case of an abnormal exit via `throw` or error (see Section 9.5 [Nonlocal Exits], page 135).

with-temp-buffer *body...* Macro

The `with-temp-buffer` macro evaluates the *body* forms with a temporary buffer as the current buffer. It saves the identity of the current buffer, creates a temporary buffer and makes it current, evaluates the *body* forms, and finally restores the previous current buffer while killing the temporary buffer.

The return value is the value of the last form in *body*. You can return the contents of the temporary buffer by using `(buffer-string)` as the last form.

The current buffer is restored even in case of an abnormal exit via `throw` or error (see Section 9.5 [Nonlocal Exits], page 135).

See also `with-temp-file` in Section 24.4 [Writing to Files], page 440.

26.3 Buffer Names

Each buffer has a unique name, which is a string. Many of the functions that work on buffers accept either a buffer or a buffer name as an argument. Any argument called *buffer-or-name* is of this sort, and an error is signaled if it is neither a string nor a buffer. Any argument called *buffer* must be an actual buffer object, not a name.

Buffers that are ephemeral and generally uninteresting to the user have names starting with a space, so that the `list-buffers` and `buffer-menu` commands don't mention them. A name starting with space also initially disables recording undo information; see Section 31.9 [Undo], page 590.

buffer-name &optional *buffer* Function

This function returns the name of *buffer* as a string. If *buffer* is not supplied, it defaults to the current buffer.

If `buffer-name` returns `nil`, it means that *buffer* has been killed. See Section 26.10 [Killing Buffers], page 491.

```
(buffer-name)
     ⇒ "buffers.texi"

(setq foo (get-buffer "temp"))
     ⇒ #<buffer temp>
(kill-buffer foo)
     ⇒ nil
(buffer-name foo)
     ⇒ nil
```

```
foo
     ⇒ #<killed buffer>
```

rename-buffer *newname* &optional *unique* Command

This function renames the current buffer to *newname*. An error is signaled if *newname* is not a string, or if there is already a buffer with that name. The function returns *newname*.

Ordinarily, `rename-buffer` signals an error if *newname* is already in use. However, if *unique* is non-`nil`, it modifies *newname* to make a name that is not in use. Interactively, you can make *unique* non-`nil` with a numeric prefix argument.

One application of this command is to rename the '`*shell*`' buffer to some other name, thus making it possible to create a second shell buffer under the name '`*shell*`'.

get-buffer *buffer-or-name* Function

This function returns the buffer specified by *buffer-or-name*. If *buffer-or-name* is a string and there is no buffer with that name, the value is `nil`. If *buffer-or-name* is a buffer, it is returned as given. (That is not very useful, so the argument is usually a name.) For example:

```
(setq b (get-buffer "lewis"))
     ⇒ #<buffer lewis>
(get-buffer b)
     ⇒ #<buffer lewis>
(get-buffer "Frazzle-nots")
     ⇒ nil
```

See also the function `get-buffer-create` in Section 26.9 [Creating Buffers], page 490.

generate-new-buffer-name *starting-name* Function

This function returns a name that would be unique for a new buffer—but does not create the buffer. It starts with *starting-name*, and produces a name not currently in use for any buffer by appending a number inside of '`<...>`'.

See the related function `generate-new-buffer` in Section 26.9 [Creating Buffers], page 490.

26.4 Buffer File Name

The *buffer file name* is the name of the file that is visited in that buffer. When a buffer is not visiting a file, its buffer file name is `nil`. Most of the time, the buffer name is the same as the nondirectory part of the buffer file name, but the buffer file name and the buffer name are distinct and can be set independently. See Section 24.1 [Visiting Files], page 433.

buffer-file-name &optional *buffer* Function

This function returns the absolute file name of the file that *buffer* is visiting. If *buffer* is not visiting any file, `buffer-file-name` returns `nil`. If *buffer* is not supplied, it defaults to the current buffer.

```
(buffer-file-name (other-buffer))
     ⇒ "/usr/user/lewis/manual/files.texi"
```

buffer-file-name Variable

This buffer-local variable contains the name of the file being visited in the current buffer, or `nil` if it is not visiting a file. It is a permanent local, unaffected by `kill-local-variables`.

```
buffer-file-name
     ⇒ "/usr/user/lewis/manual/buffers.texi"
```

It is risky to change this variable's value without doing various other things. Normally it is better to use `set-visited-file-name` (see below); some of the things done there, such as changing the buffer name, are not strictly necessary, but others are essential to avoid confusing Emacs.

buffer-file-truename Variable

This buffer-local variable holds the truename of the file visited in the current buffer, or `nil` if no file is visited. It is a permanent local, unaffected by `kill-local-variables`. See Section 24.6.3 [Truenames], page 445.

buffer-file-number Variable

This buffer-local variable holds the file number and directory device number of the file visited in the current buffer, or `nil` if no file or a nonexistent file is visited. It is a permanent local, unaffected by `kill-local-variables`.

The value is normally a list of the form (*filenum devnum*). This pair of numbers uniquely identifies the file among all files accessible on the system. See the function `file-attributes`, in Section 24.6.4 [File Attributes], page 446, for more information about them.

get-file-buffer *filename* Function

This function returns the buffer visiting file *filename*. If there is no such buffer, it returns `nil`. The argument *filename*, which must be a string, is expanded (see Section 24.8.4 [File Name Expansion], page 454), then compared against the visited file names of all live buffers.

```
(get-file-buffer "buffers.texi")
    ⇒ #<buffer buffers.texi>
```

In unusual circumstances, there can be more than one buffer visiting the same file name. In such cases, this function returns the first such buffer in the buffer list.

set-visited-file-name *filename* &optional *no-query along-with-file* Command

If *filename* is a non-empty string, this function changes the name of the file visited in current buffer to *filename*. (If the buffer had no visited file, this gives it one.) The *next time* the buffer is saved it will go in the newly-specified file. This command marks the buffer as modified, since it does not (as far as Emacs knows) match the contents of *filename*, even if it matched the former visited file.

If *filename* is `nil` or the empty string, that stands for "no visited file". In this case, `set-visited-file-name` marks the buffer as having no visited file.

Normally, this function asks the user for confirmation if the specified file already exists. If *no-query* is non-`nil`, that prevents asking this question.

If *along-with-file* is non-`nil`, that means to assume that the former visited file has been renamed to *filename*.

When the function `set-visited-file-name` is called interactively, it prompts for *filename* in the minibuffer.

list-buffers-directory Variable

This buffer-local variable specifies a string to display in a buffer listing where the visited file name would go, for buffers that don't have a visited file name. Dired buffers use this variable.

26.5 Buffer Modification

Emacs keeps a flag called the *modified flag* for each buffer, to record whether you have changed the text of the buffer. This flag is set to `t` whenever you alter the contents of the buffer, and cleared to `nil` when you save it. Thus, the flag shows whether there are unsaved changes. The flag value is normally shown in the mode line (see Section 22.3.2 [Mode Line Variables], page 408), and controls saving (see Section 24.2 [Saving Buffers], page 436) and auto-saving (see Section 25.2 [Auto-Saving], page 472).

Some Lisp programs set the flag explicitly. For example, the function `set-visited-file-name` sets the flag to `t`, because the text does not match the newly-visited file, even if it is unchanged from the file formerly visited.

The functions that modify the contents of buffers are described in Chapter 31 [Text], page 575.

buffer-modified-p &optional *buffer* Function

This function returns `t` if the buffer *buffer* has been modified since it was last read in from a file or saved, or `nil` otherwise. If *buffer* is not supplied, the current buffer is tested.

set-buffer-modified-p *flag* Function

This function marks the current buffer as modified if *flag* is non-`nil`, or as unmodified if the flag is `nil`.

Another effect of calling this function is to cause unconditional redisplay of the mode line for the current buffer. In fact, the function `force-mode-line-update` works by doing this:

```
(set-buffer-modified-p (buffer-modified-p))
```

not-modified Command

This command marks the current buffer as unmodified, and not needing to be saved. With prefix arg, it marks the buffer as modified, so that it will be saved at the next suitable occasion.

Don't use this function in programs, since it prints a message in the echo area; use `set-buffer-modified-p` (above) instead.

buffer-modified-tick &optional *buffer* Function

This function returns *buffer*'s modification-count. This is a counter that increments every time the buffer is modified. If *buffer* is `nil` (or omitted), the current buffer is used.

26.6 Comparison of Modification Time

Suppose that you visit a file and make changes in its buffer, and meanwhile the file itself is changed on disk. At this point, saving the buffer would overwrite the changes in the file. Occasionally this may be what you want, but usually it would lose valuable information. Emacs therefore checks the file's modification time using the functions described below before saving the file.

verify-visited-file-modtime *buffer* Function

This function compares what *buffer* has recorded for the modification time of its visited file against the actual modification time of the file as recorded by the operating system. The two should be the same unless some other process has written the file since Emacs visited or saved it.

The function returns `t` if the last actual modification time and Emacs's recorded modification time are the same, `nil` otherwise.

clear-visited-file-modtime Function

This function clears out the record of the last modification time of the file being visited by the current buffer. As a result, the next attempt to save this buffer will not complain of a discrepancy in file modification times.

This function is called in `set-visited-file-name` and other exceptional places where the usual test to avoid overwriting a changed file should not be done.

visited-file-modtime Function

This function returns the buffer's recorded last file modification time, as a list of the form (*high* . *low*). (This is the same format that `file-attributes` uses to return time values; see Section 24.6.4 [File Attributes], page 446.)

set-visited-file-modtime &optional *time* Function

This function updates the buffer's record of the last modification time of the visited file, to the value specified by *time* if *time* is not `nil`, and otherwise to the last modification time of the visited file.

If *time* is not `nil`, it should have the form (*high* . *low*) or (*high low*), in either case containing two integers, each of which holds 16 bits of the time.

This function is useful if the buffer was not read from the file normally, or if the file itself has been changed for some known benign reason.

ask-user-about-supersession-threat *filename* Function

This function is used to ask a user how to proceed after an attempt to modify an obsolete buffer visiting file *filename*. An *obsolete buffer* is an unmodified buffer for which the associated file on disk is newer than the last save-time of the buffer. This means some other program has probably altered the file.

Depending on the user's answer, the function may return normally, in which case the modification of the buffer proceeds, or it may signal a `file-supersession` error with data (*filename*), in which case the proposed buffer modification is not allowed.

This function is called automatically by Emacs on the proper occasions. It exists so you can customize Emacs by redefining it. See the file '`userlock.el`' for the standard definition.

See also the file locking mechanism in Section 24.5 [File Locks], page 441.

26.7 Read-Only Buffers

If a buffer is *read-only*, then you cannot change its contents, although you may change your view of the contents by scrolling and narrowing.

Read-only buffers are used in two kinds of situations:

- A buffer visiting a write-protected file is normally read-only.

  Here, the purpose is to inform the user that editing the buffer with the aim of saving it in the file may be futile or undesirable. The user who wants to change the buffer text despite this can do so after clearing the read-only flag with `C-x C-q`.

- Modes such as Dired and Rmail make buffers read-only when altering the contents with the usual editing commands is probably a mistake.

The special commands of these modes bind `buffer-read-only` to `nil` (with `let`) or bind `inhibit-read-only` to `t` around the places where they themselves change the text.

buffer-read-only Variable

This buffer-local variable specifies whether the buffer is read-only. The buffer is read-only if this variable is non-`nil`.

inhibit-read-only Variable

If this variable is non-`nil`, then read-only buffers and read-only characters may be modified. Read-only characters in a buffer are those that have non-`nil` `read-only` properties (either text properties or overlay properties). See Section 31.19.4 [Special Properties], page 615, for more information about text properties. See Section 38.8 [Overlays], page 748, for more information about overlays and their properties.

If `inhibit-read-only` is `t`, all `read-only` character properties have no effect. If `inhibit-read-only` is a list, then `read-only` character properties have no effect if they are members of the list (comparison is done with `eq`).

toggle-read-only Command

This command changes whether the current buffer is read-only. It is intended for interactive use; don't use it in programs. At any given point in a program, you should know whether you want the read-only flag on or off; so you can set `buffer-read-only` explicitly to the proper value, `t` or `nil`.

barf-if-buffer-read-only Function

This function signals a `buffer-read-only` error if the current buffer is read-only. See Section 20.3 [Interactive Call], page 325, for another way to signal an error if the current buffer is read-only.

26.8 The Buffer List

The *buffer list* is a list of all live buffers. Creating a buffer adds it to this list, and killing a buffer excises it. The order of the buffers in the list is based primarily on how recently each buffer has been displayed in the selected window. Buffers move to the front of the list when they are selected and to the end when they are buried (see `bury-buffer`, below). Several functions, notably `other-buffer`, use this ordering. A buffer list displayed for the user also follows this order.

In addition to the fundamental Emacs buffer list, each frame has its own version of the buffer list, in which the buffers that have been selected in that frame come first, starting with the buffers most recently selected *in*

that frame. (This order is recorded in *frame*'s `buffer-list` frame parameter; see Section 28.3.3 [Window Frame Parameters], page 528.) The buffers that were never selected in *frame* come afterward, ordered according to the fundamental Emacs buffer list.

buffer-list &optional *frame* Function

This function returns the buffer list, including all buffers, even those whose names begin with a space. The elements are actual buffers, not their names.

If *frame* is a frame, this returns *frame*'s buffer list. If *frame* is `nil`, the fundamental Emacs buffer list is used: all the buffers appear in order of most recent selection, regardless of which frames they were selected in.

```
(buffer-list)
     ⇒ (#<buffer buffers.texi>
         #<buffer  *Minibuf-1*> #<buffer buffer.c>
         #<buffer *Help*> #<buffer TAGS>)

;; Note that the name of the minibuffer
;;    begins with a space!
(mapcar (function buffer-name) (buffer-list))
    ⇒ ("buffers.texi" " *Minibuf-1*"
        "buffer.c" "*Help*" "TAGS")
```

The list that `buffer-list` returns is constructed specifically by `buffer-list`; it is not an internal Emacs data structure, and modifying it has no effect on the order of buffers. If you want to change the order of buffers in the frame-independent buffer list, here is an easy way:

```
(defun reorder-buffer-list (new-list)
  (while new-list
    (bury-buffer (car new-list))
    (setq new-list (cdr new-list))))
```

With this method, you can specify any order for the list, but there is no danger of losing a buffer or adding something that is not a valid live buffer.

To change the order or value of a frame's buffer list, set the frame's `buffer-list` frame parameter with `modify-frame-parameters` (see Section 28.3.1 [Parameter Access], page 527).

other-buffer &optional *buffer visible-ok frame* Function

This function returns the first buffer in the buffer list other than *buffer*. Usually this is the buffer selected most recently (in frame *frame* or else the currently selected frame), aside from *buffer*. Buffers whose names start with a space are not considered at all.

If *buffer* is not supplied (or if it is not a buffer), then `other-buffer` returns the first buffer in the selected frame's buffer list that is not now visible in any window in a visible frame.

If *frame* has a non-`nil` `buffer-predicate` parameter, then `other-buffer` uses that predicate to decide which buffers to consider. It calls the predicate once for each buffer, and if the value is `nil`, that buffer is ignored. See Section 28.3.3 [Window Frame Parameters], page 528.

If *visible-ok* is `nil`, `other-buffer` avoids returning a buffer visible in any window on any visible frame, except as a last resort. If *visible-ok* is non-`nil`, then it does not matter whether a buffer is displayed somewhere or not.

If no suitable buffer exists, the buffer '`*scratch*`' is returned (and created, if necessary).

bury-buffer &optional *buffer-or-name* Command

This function puts *buffer-or-name* at the end of the buffer list, without changing the order of any of the other buffers on the list. This buffer therefore becomes the least desirable candidate for `other-buffer` to return.

`bury-buffer` operates on each frame's `buffer-list` parameter as well as the frame-independent Emacs buffer list; therefore, the buffer that you bury will come last in the value of `(buffer-list` *frame*`)` and in the value of `(buffer-list nil)`.

If *buffer-or-name* is `nil` or omitted, this means to bury the current buffer. In addition, if the buffer is displayed in the selected window, this switches to some other buffer (obtained using `other-buffer`) in the selected window. But if the buffer is displayed in some other window, it remains displayed there.

To replace a buffer in all the windows that display it, use `replace-buffer-in-windows`. See Section 27.6 [Buffers and Windows], page 503.

26.9 Creating Buffers

This section describes the two primitives for creating buffers. `get-buffer-create` creates a buffer if it finds no existing buffer with the specified name; `generate-new-buffer` always creates a new buffer and gives it a unique name.

Other functions you can use to create buffers include `with-output-to-temp-buffer` (see Section 38.7 [Temporary Displays], page 746) and `create-file-buffer` (see Section 24.1 [Visiting Files], page 433). Starting a subprocess can also create a buffer (see Chapter 36 [Processes], page 689).

get-buffer-create *name* Function

This function returns a buffer named *name*. It returns an existing buffer with that name, if one exists; otherwise, it creates a new buffer. The buffer does not become the current buffer—this function does not change which buffer is current.

An error is signaled if *name* is not a string.

```
(get-buffer-create "foo")
     ⇒ #<buffer foo>
```

The major mode for the new buffer is set to Fundamental mode. The variable `default-major-mode` is handled at a higher level. See Section 22.1.3 [Auto Major Mode], page 398.

generate-new-buffer *name* Function

This function returns a newly created, empty buffer, but does not make it current. If there is no buffer named *name*, then that is the name of the new buffer. If that name is in use, this function adds suffixes of the form '`<`*n*`>`' to *name*, where *n* is an integer. It tries successive integers starting with 2 until it finds an available name.

An error is signaled if *name* is not a string.

```
(generate-new-buffer "bar")
     ⇒ #<buffer bar>
(generate-new-buffer "bar")
     ⇒ #<buffer bar<2>>
(generate-new-buffer "bar")
     ⇒ #<buffer bar<3>>
```

The major mode for the new buffer is set to Fundamental mode. The variable `default-major-mode` is handled at a higher level. See Section 22.1.3 [Auto Major Mode], page 398.

See the related function `generate-new-buffer-name` in Section 26.3 [Buffer Names], page 482.

26.10 Killing Buffers

Killing a buffer makes its name unknown to Emacs and makes its text space available for other use.

The buffer object for the buffer that has been killed remains in existence as long as anything refers to it, but it is specially marked so that you cannot make it current or display it. Killed buffers retain their identity, however; two distinct buffers, when killed, remain distinct according to `eq`.

If you kill a buffer that is current or displayed in a window, Emacs automatically selects or displays some other buffer instead. This means that killing a buffer can in general change the current buffer. Therefore, when you kill a buffer, you should also take the precautions associated with changing the current buffer (unless you happen to know that the buffer being killed isn't current). See Section 26.2 [Current Buffer], page 479.

If you kill a buffer that is the base buffer of one or more indirect buffers, the indirect buffers are automatically killed as well.

The `buffer-name` of a killed buffer is `nil`. You can use this feature to test whether a buffer has been killed:

```
(defun buffer-killed-p (buffer)
  "Return t if BUFFER is killed."
  (not (buffer-name buffer)))
```

kill-buffer *buffer-or-name* Command

This function kills the buffer *buffer-or-name*, freeing all its memory for other uses or to be returned to the operating system. It returns `nil`.

Any processes that have this buffer as the `process-buffer` are sent the `SIGHUP` signal, which normally causes them to terminate. (The basic meaning of `SIGHUP` is that a dialup line has been disconnected.) See Section 36.5 [Deleting Processes], page 696.

If the buffer is visiting a file and contains unsaved changes, `kill-buffer` asks the user to confirm before the buffer is killed. It does this even if not called interactively. To prevent the request for confirmation, clear the modified flag before calling `kill-buffer`. See Section 26.5 [Buffer Modification], page 485.

Killing a buffer that is already dead has no effect.

```
(kill-buffer "foo.unchanged")
        nil
(kill-buffer "foo.changed")

---------- Buffer: Minibuffer ----------
Buffer foo.changed modified; kill anyway? (yes or no) yes
---------- Buffer: Minibuffer ----------

        nil
```

kill-buffer-query-functions Variable

After confirming unsaved changes, `kill-buffer` calls the functions in the list `kill-buffer-query-functions`, in order of appearance, with no arguments. The buffer being killed is the current buffer when they are called. The idea is that these functions ask for confirmation from the user for various nonstandard reasons. If any of them returns `nil`, `kill-buffer` spares the buffer's life.

kill-buffer-hook Variable

This is a normal hook run by `kill-buffer` after asking all the questions it is going to ask, just before actually killing the buffer. The buffer to be killed is current when the hook functions run. See Section 22.6 [Hooks], page 420.

buffer-offer-save Variable

This variable, if non-`nil` in a particular buffer, tells `save-buffers-kill-emacs` and `save-some-buffers` to offer to save that buffer, just as they

offer to save file-visiting buffers. The variable `buffer-offer-save` automatically becomes buffer-local when set for any reason. See Section 10.10 [Buffer-Local Variables], page 161.

26.11 Indirect Buffers

An *indirect buffer* shares the text of some other buffer, which is called the *base buffer* of the indirect buffer. In some ways it is the analogue, for buffers, of a symbolic link among files. The base buffer may not itself be an indirect buffer.

The text of the indirect buffer is always identical to the text of its base buffer; changes made by editing either one are visible immediately in the other. This includes the text properties as well as the characters themselves.

But in all other respects, the indirect buffer and its base buffer are completely separate. They have different names, different values of point, different narrowing, different markers and overlays (though inserting or deleting text in either buffer relocates the markers and overlays for both), different major modes, and different buffer-local variables.

An indirect buffer cannot visit a file, but its base buffer can. If you try to save the indirect buffer, that actually works by saving the base buffer.

Killing an indirect buffer has no effect on its base buffer. Killing the base buffer effectively kills the indirect buffer in that it cannot ever again be the current buffer.

make-indirect-buffer *base-buffer name* Command

This creates an indirect buffer named *name* whose base buffer is *base-buffer*. The argument *base-buffer* may be a buffer or a string.

If *base-buffer* is an indirect buffer, its base buffer is used as the base for the new buffer.

buffer-base-buffer *buffer* Function

This function returns the base buffer of *buffer*. If *buffer* is not indirect, the value is `nil`. Otherwise, the value is another buffer, which is never an indirect buffer.

27 Windows

This chapter describes most of the functions and variables related to Emacs windows. See Chapter 38 [Display], page 739, for information on how text is displayed in windows.

27.1 Basic Concepts of Emacs Windows

A *window* in Emacs is the physical area of the screen in which a buffer is displayed. The term is also used to refer to a Lisp object that represents that screen area in Emacs Lisp. It should be clear from the context which is meant.

Emacs groups windows into frames. A frame represents an area of screen available for Emacs to use. Each frame always contains at least one window, but you can subdivide it vertically or horizontally into multiple nonoverlapping Emacs windows.

In each frame, at any time, one and only one window is designated as *selected within the frame*. The frame's cursor appears in that window. At any time, one frame is the selected frame; and the window selected within that frame is *the selected window*. The selected window's buffer is usually the current buffer (except when `set-buffer` has been used). See Section 26.2 [Current Buffer], page 479.

For practical purposes, a window exists only while it is displayed in a frame. Once removed from the frame, the window is effectively deleted and should not be used, *even though there may still be references to it* from other Lisp objects. Restoring a saved window configuration is the only way for a window no longer on the screen to come back to life. (See Section 27.3 [Deleting Windows], page 499.)

Each window has the following attributes:

- containing frame
- window height
- window width
- window edges with respect to the screen or frame
- the buffer it displays
- position within the buffer at the upper left of the window
- amount of horizontal scrolling, in columns
- point
- the mark
- how recently the window was selected

Users create multiple windows so they can look at several buffers at once. Lisp libraries use multiple windows for a variety of reasons, but most often to display related information. In Rmail, for example, you can move through a

summary buffer in one window while the other window shows messages one at a time as they are reached.

The meaning of "window" in Emacs is similar to what it means in the context of general-purpose window systems such as X, but not identical. The X Window System places X windows on the screen; Emacs uses one or more X windows as frames, and subdivides them into Emacs windows. When you use Emacs on a character-only terminal, Emacs treats the whole terminal screen as one frame.

Most window systems support arbitrarily located overlapping windows. In contrast, Emacs windows are *tiled*; they never overlap, and together they fill the whole screen or frame. Because of the way in which Emacs creates new windows and resizes them, not all conceivable tilings of windows on an Emacs frame are actually possible. See Section 27.2 [Splitting Windows], page 496, and Section 27.13 [Size of Window], page 516.

See Chapter 38 [Display], page 739, for information on how the contents of the window's buffer are displayed in the window.

windowp *object* Function

This function returns `t` if *object* is a window.

27.2 Splitting Windows

The functions described here are the primitives used to split a window into two windows. Two higher level functions sometimes split a window, but not always: `pop-to-buffer` and `display-buffer` (see Section 27.7 [Displaying Buffers], page 504).

The functions described here do not accept a buffer as an argument. The two "halves" of the split window initially display the same buffer previously visible in the window that was split.

split-window &optional *window size horizontal* Command

This function splits *window* into two windows. The original window *window* remains the selected window, but occupies only part of its former screen area. The rest is occupied by a newly created window which is returned as the value of this function.

If *horizontal* is non-`nil`, then *window* splits into two side by side windows. The original window *window* keeps the leftmost *size* columns, and gives the rest of the columns to the new window. Otherwise, it splits into windows one above the other, and *window* keeps the upper *size* lines and gives the rest of the lines to the new window. The original window is therefore the left-hand or upper of the two, and the new window is the right-hand or lower.

If *window* is omitted or `nil`, then the selected window is split. If *size* is omitted or `nil`, then *window* is divided evenly into two parts. (If there

is an odd line, it is allocated to the new window.) When `split-window` is called interactively, all its arguments are `nil`.

The following example starts with one window on a screen that is 50 lines high by 80 columns wide; then the window is split.

```
(setq w (selected-window))
        #<window 8 on windows.texi>
(window-edges)          ; Edges in order:
        (0 0 80 50)     ;    left-top-right-bottom
;; Returns window created
(setq w2 (split-window w 15))
        #<window 28 on windows.texi>
(window-edges w2)
        (0 15 80 50)    ; Bottom window;
                        ;    top is line 15
(window-edges w)
        (0 0 80 15)     ; Top window
```

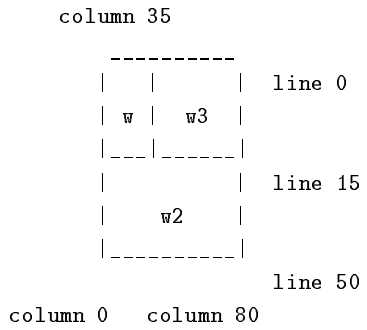
The screen looks like this:

```
          ----------
         |          |  line 0
         |    w     |
         |----------|
         |          |  line 15
         |    w2    |
         |----------|
                        line 50
 column 0   column 80
```

Next, the top window is split horizontally:

```
(setq w3 (split-window w 35 t))
        #<window 32 on windows.texi>
(window-edges w3)
        (35 0 80 15)  ; Left edge at column 35
(window-edges w)
        (0 0 35 15)   ; Right edge at column 35
(window-edges w2)
        (0 15 80 50)  ; Bottom window unchanged
```

Now, the screen looks like this:



Normally, Emacs indicates the border between two side-by-side windows with a scroll bar (see Section 28.3.3 [Window Frame Parameters], page 528) or '|' characters. The display table can specify alternative border characters; see Section 38.14 [Display Tables], page 762.

split-window-vertically *size* Command

This function splits the selected window into two windows, one above the other, leaving the upper of the two windows selected, with *size* lines. (If *size* is negative, then the lower of the two windows gets − *size* lines and the upper window gets the rest, but the upper window is still the one selected.)

This function is simply an interface to `split-window`. Here is the complete function definition for it:

```
(defun split-window-vertically (&optional arg)
  "Split current window into two windows, ..."
  (interactive "P")
  (split-window nil (and arg (prefix-numeric-value arg))))
```

split-window-horizontally *size* Command

This function splits the selected window into two windows side-by-side, leaving the selected window with *size* columns.

This function is simply an interface to `split-window`. Here is the complete definition for `split-window-horizontally` (except for part of the documentation string):

```
(defun split-window-horizontally (&optional arg)
  "Split selected window into two windows, side by side..."
  (interactive "P")
  (split-window nil (and arg (prefix-numeric-value arg)) t))
```

one-window-p &optional *no-mini all-frames* Function

This function returns non-`nil` if there is only one window. The argument *no-mini*, if non-`nil`, means don't count the minibuffer even if it is active;

otherwise, the minibuffer window is included, if active, in the total number of windows, which is compared against one.

The argument *all-frames* specifies which frames to consider. Here are the possible values and their meanings:

`nil` Count the windows in the selected frame, plus the minibuffer used by that frame even if it lies in some other frame.

`t` Count all windows in all existing frames.

`visible` Count all windows in all visible frames.

0 Count all windows in all visible or iconified frames.

anything else
Count precisely the windows in the selected frame, and no others.

27.3 Deleting Windows

A window remains visible on its frame unless you *delete* it by calling certain functions that delete windows. A deleted window cannot appear on the screen, but continues to exist as a Lisp object until there are no references to it. There is no way to cancel the deletion of a window aside from restoring a saved window configuration (see Section 27.16 [Window Configurations], page 521). Restoring a window configuration also deletes any windows that aren't part of that configuration.

When you delete a window, the space it took up is given to one adjacent sibling.

window-live-p *window* Function

This function returns `nil` if *window* is deleted, and `t` otherwise.

Warning: Erroneous information or fatal errors may result from using a deleted window as if it were live.

delete-window &optional *window* Command

This function removes *window* from display, and returns `nil`. If *window* is omitted, then the selected window is deleted. An error is signaled if there is only one window when `delete-window` is called.

delete-other-windows &optional *window* Command

This function makes *window* the only window on its frame, by deleting the other windows in that frame. If *window* is omitted or `nil`, then the selected window is used by default.

The return value is `nil`.

delete-windows-on *buffer* &optional *frame* Command

This function deletes all windows showing *buffer*. If there are no windows showing *buffer*, it does nothing.

`delete-windows-on` operates frame by frame. If a frame has several windows showing different buffers, then those showing *buffer* are removed, and the others expand to fill the space. If all windows in some frame are showing *buffer* (including the case where there is only one window), then the frame reverts to having a single window showing another buffer chosen with `other-buffer`. See Section 26.8 [The Buffer List], page 488.

The argument *frame* controls which frames to operate on. This function does not use it in quite the same way as the other functions which scan all windows; specifically, the values `t` and `nil` have the opposite of their meanings in other functions. Here are the full details:

- If it is `nil`, operate on all frames.
- If it is `t`, operate on the selected frame.
- If it is `visible`, operate on all visible frames.
- If it is 0, operate on all visible or iconified frames.
- If it is a frame, operate on that frame.

This function always returns `nil`.

27.4 Selecting Windows

When a window is selected, the buffer in the window becomes the current buffer, and the cursor will appear in it.

selected-window Function

This function returns the selected window. This is the window in which the cursor appears and to which many commands apply.

select-window *window* Function

This function makes *window* the selected window. The cursor then appears in *window* (on redisplay). The buffer being displayed in *window* is immediately designated the current buffer.

The return value is *window*.

```
(setq w (next-window))
(select-window w)
     ⇒ #<window 65 on windows.texi>
```

save-selected-window *forms...* Macro

This macro records the selected window, executes *forms* in sequence, then restores the earlier selected window.

This macro does not save or restore anything about the sizes, arrangement or contents of windows; therefore, if the *forms* change them, the change persists.

Each frame, at any time, has a window selected within the frame. This macro saves only *the* selected window; it does not save anything about other frames. If the *forms* select some other frame and alter the window selected within it, the change persists.

The following functions choose one of the windows on the screen, offering various criteria for the choice.

get-lru-window &optional *frame* Function

This function returns the window least recently "used" (that is, selected). The selected window is always the most recently used window.

The selected window can be the least recently used window if it is the only window. A newly created window becomes the least recently used window until it is selected. A minibuffer window is never a candidate.

The argument *frame* controls which windows are considered.

- If it is `nil`, consider windows on the selected frame.
- If it is `t`, consider windows on all frames.
- If it is `visible`, consider windows on all visible frames.
- If it is 0, consider windows on all visible or iconified frames.
- If it is a frame, consider windows on that frame.

get-largest-window &optional *frame* Function

This function returns the window with the largest area (height times width). If there are no side-by-side windows, then this is the window with the most lines. A minibuffer window is never a candidate.

If there are two windows of the same size, then the function returns the window that is first in the cyclic ordering of windows (see following section), starting from the selected window.

The argument *frame* controls which set of windows to consider. See `get-lru-window`, above.

27.5 Cyclic Ordering of Windows

When you use the command `C-x o` (`other-window`) to select the next window, it moves through all the windows on the screen in a specific cyclic order. For any given configuration of windows, this order never varies. It is called the *cyclic ordering of windows*.

This ordering generally goes from top to bottom, and from left to right. But it may go down first or go right first, depending on the order in which the windows were split.

If the first split was vertical (into windows one above each other), and then the subwindows were split horizontally, then the ordering is left to right in the top of the frame, and then left to right in the next lower part of the frame, and so on. If the first split was horizontal, the ordering is top to bottom in the left part, and so on. In general, within each set of siblings at any level in the window tree, the order is left to right, or top to bottom.

next-window &optional *window minibuf all-frames* Function

This function returns the window following *window* in the cyclic ordering of windows. This is the window that `C-x o` would select if typed when *window* is selected. If *window* is the only window visible, then this function returns *window*. If omitted, *window* defaults to the selected window.

The value of the argument *minibuf* determines whether the minibuffer is included in the window order. Normally, when *minibuf* is `nil`, the minibuffer is included if it is currently active; this is the behavior of `C-x o`. (The minibuffer window is active while the minibuffer is in use. See Chapter 19 [Minibuffers], page 295.)

If *minibuf* is `t`, then the cyclic ordering includes the minibuffer window even if it is not active.

If *minibuf* is neither `t` nor `nil`, then the minibuffer window is not included even if it is active.

The argument *all-frames* specifies which frames to consider. Here are the possible values and their meanings:

`nil` Consider all the windows in *window*'s frame, plus the minibuffer used by that frame even if it lies in some other frame.

`t` Consider all windows in all existing frames.

`visible` Consider all windows in all visible frames. (To get useful results, you must ensure *window* is in a visible frame.)

0 Consider all windows in all visible or iconified frames.

anything else
Consider precisely the windows in *window*'s frame, and no others.

This example assumes there are two windows, both displaying the buffer '`windows.texi`':

```
(selected-window)
     ⇒ #<window 56 on windows.texi>
(next-window (selected-window))
     ⇒ #<window 52 on windows.texi>
(next-window (next-window (selected-window)))
     ⇒ #<window 56 on windows.texi>
```

previous-window &optional *window minibuf all-frames* Function
This function returns the window preceding *window* in the cyclic ordering of windows. The other arguments specify which windows to include in the cycle, as in `next-window`.

other-window *count* Command
This function selects the *count*th following window in the cyclic order. If count is negative, then it moves back $-$*count* windows in the cycle, rather than forward. It returns `nil`.

In an interactive call, *count* is the numeric prefix argument.

walk-windows *proc* &optional *minibuf all-frames* Function
This function cycles through all windows, calling `proc` once for each window with the window as its sole argument.

The optional arguments *minibuf* and *all-frames* specify the set of windows to include in the scan. See `next-window`, above, for details.

27.6 Buffers and Windows

This section describes low-level functions to examine windows or to display buffers in windows in a precisely controlled fashion. See the following section for related functions that find a window to use and specify a buffer for it. The functions described there are easier to use than these, but they employ heuristics in choosing or creating a window; use these functions when you need complete control.

set-window-buffer *window buffer-or-name* Function
This function makes *window* display *buffer-or-name* as its contents. It returns `nil`. This is the fundamental primitive for changing which buffer is displayed in a window, and all ways of doing that call this function.

```
(set-window-buffer (selected-window) "foo")
     ⇒ nil
```

window-buffer &optional *window* Function
This function returns the buffer that *window* is displaying. If *window* is omitted, this function returns the buffer for the selected window.

```
(window-buffer)
     ⇒ #<buffer windows.texi>
```

get-buffer-window *buffer-or-name* &optional *all-frames* Function
This function returns a window currently displaying *buffer-or-name*, or `nil` if there is none. If there are several such windows, then the function returns the first one in the cyclic ordering of windows, starting from the selected window. See Section 27.5 [Cyclic Window Ordering], page 501.

The argument *all-frames* controls which windows to consider.

- If it is `nil`, consider windows on the selected frame.
- If it is `t`, consider windows on all frames.
- If it is `visible`, consider windows on all visible frames.
- If it is 0, consider windows on all visible or iconified frames.
- If it is a frame, consider windows on that frame.

get-buffer-window-list *buffer-or-name* &optional *minibuf* *all-frames* Function

This function returns a list of all the windows currently displaying *buffer-or-name*.

The two optional arguments work like the optional arguments of `next-window` (see Section 27.5 [Cyclic Window Ordering], page 501); they are *not* like the single optional argument of `get-buffer-window`. Perhaps we should change `get-buffer-window` in the future to make it compatible with the other functions.

The argument *all-frames* controls which windows to consider.

- If it is `nil`, consider windows on the selected frame.
- If it is `t`, consider windows on all frames.
- If it is `visible`, consider windows on all visible frames.
- If it is 0, consider windows on all visible or iconified frames.
- If it is a frame, consider windows on that frame.

buffer-display-time Variable

This variable records the time at which a buffer was last made visible in a window. It is always local in each buffer; each time `set-window-buffer` is called, it sets this variable to `(current-time)` in the specified buffer (see Section 37.5 [Time of Day], page 723). When a buffer is first created, `buffer-display-time` starts out with the value `nil`.

27.7 Displaying Buffers in Windows

In this section we describe convenient functions that choose a window automatically and use it to display a specified buffer. These functions can also split an existing window in certain circumstances. We also describe variables that parameterize the heuristics used for choosing a window. See the preceding section for low-level functions that give you more precise control. All of these functions work by calling `set-window-buffer`.

Do not use the functions in this section in order to make a buffer current so that a Lisp program can access or modify it; they are too drastic for that purpose, since they change the display of buffers in windows, which would be gratuitous and surprise the user. Instead, use `set-buffer` and `save-current-buffer` (see Section 26.2 [Current Buffer], page 479), which designate buffers as current for programmed access without affecting the display of buffers in windows.

switch-to-buffer *buffer-or-name* &optional *norecord* Command

This function makes *buffer-or-name* the current buffer, and also displays the buffer in the selected window. This means that a human can see the buffer and subsequent keyboard commands will apply to it. Contrast this with `set-buffer`, which makes *buffer-or-name* the current buffer but does not display it in the selected window. See Section 26.2 [Current Buffer], page 479.

If *buffer-or-name* does not identify an existing buffer, then a new buffer by that name is created. The major mode for the new buffer is set according to the variable `default-major-mode`. See Section 22.1.3 [Auto Major Mode], page 398.

Normally the specified buffer is put at the front of the buffer list (both the selected frame's buffer list and the frame-independent buffer list). This affects the operation of `other-buffer`. However, if *norecord* is non-`nil`, this is not done. See Section 26.8 [The Buffer List], page 488.

The `switch-to-buffer` function is often used interactively, as the binding of `C-x b`. It is also used frequently in programs. It always returns `nil`.

switch-to-buffer-other-window *buffer-or-name* &optional *norecord* Command

This function makes *buffer-or-name* the current buffer and displays it in a window not currently selected. It then selects that window. The handling of the buffer is the same as in `switch-to-buffer`.

The currently selected window is absolutely never used to do the job. If it is the only window, then it is split to make a distinct window for this purpose. If the selected window is already displaying the buffer, then it continues to do so, but another window is nonetheless found to display it in as well.

This function updates the buffer list just like `switch-to-buffer` unless *norecord* is non-`nil`.

pop-to-buffer *buffer-or-name* &optional *other-window* *norecord* Function

This function makes *buffer-or-name* the current buffer and switches to it in some window, preferably not the window previously selected. The "popped-to" window becomes the selected window within its frame.

If the variable `pop-up-frames` is non-`nil`, `pop-to-buffer` looks for a window in any visible frame already displaying the buffer; if there is one, it returns that window and makes it be selected within its frame. If there is none, it creates a new frame and displays the buffer in it.

If `pop-up-frames` is `nil`, then `pop-to-buffer` operates entirely within the selected frame. (If the selected frame has just a minibuffer, `pop-to-buffer` operates within the most recently selected frame that was not just a minibuffer.)

If the variable `pop-up-windows` is non-`nil`, windows may be split to create a new window that is different from the original window. For details, see Section 27.8 [Choosing Window], page 506.

If *other-window* is non-`nil`, `pop-to-buffer` finds or creates another window even if *buffer-or-name* is already visible in the selected window. Thus *buffer-or-name* could end up displayed in two windows. On the other hand, if *buffer-or-name* is already displayed in the selected window and *other-window* is `nil`, then the selected window is considered sufficient display for *buffer-or-name*, so that nothing needs to be done.

All the variables that affect `display-buffer` affect `pop-to-buffer` as well. See Section 27.8 [Choosing Window], page 506.

If *buffer-or-name* is a string that does not name an existing buffer, a buffer by that name is created. The major mode for the new buffer is set according to the variable `default-major-mode`. See Section 22.1.3 [Auto Major Mode], page 398.

This function updates the buffer list just like `switch-to-buffer` unless *norecord* is non-`nil`.

replace-buffer-in-windows *buffer* Command

This function replaces *buffer* with some other buffer in all windows displaying it. The other buffer used is chosen with `other-buffer`. In the usual applications of this function, you don't care which other buffer is used; you just want to make sure that *buffer* is no longer displayed.

This function returns `nil`.

27.8 Choosing a Window for Display

This section describes the basic facility that chooses a window to display a buffer in—`display-buffer`. All the higher-level functions and commands use this subroutine. Here we describe how to use `display-buffer` and how to customize it.

display-buffer *buffer-or-name* &optional *not-this-window* *frame* Command

This command makes *buffer-or-name* appear in some window, like `pop-to-buffer`, but it does not select that window and does not make the buffer current. The identity of the selected window is unaltered by this function.

If *not-this-window* is non-`nil`, it means to display the specified buffer in a window other than the selected one, even if it is already on display in the selected window. This can cause the buffer to appear in two windows at once. Otherwise, if *buffer-or-name* is already being displayed in any window, that is good enough, so this function does nothing.

`display-buffer` returns the window chosen to display *buffer-or-name*.

If the argument *frame* is non-`nil`, it specifies which frames to check when deciding whether the buffer is already displayed. If the buffer is already displayed in some window on one of these frames, `display-buffer` simply returns that window. Here are the possible values of *frame*:

- If it is `nil`, consider windows on the selected frame.
- If it is `t`, consider windows on all frames.
- If it is `visible`, consider windows on all visible frames.
- If it is 0, consider windows on all visible or iconified frames.
- If it is a frame, consider windows on that frame.

Precisely how `display-buffer` finds or creates a window depends on the variables described below.

pop-up-windows User Option

This variable controls whether `display-buffer` makes new windows. If it is non-`nil` and there is only one window, then that window is split. If it is `nil`, then `display-buffer` does not split the single window, but uses it whole.

split-height-threshold User Option

This variable determines when `display-buffer` may split a window, if there are multiple windows. `display-buffer` always splits the largest window if it has at least this many lines. If the largest window is not this tall, it is split only if it is the sole window and `pop-up-windows` is non-`nil`.

pop-up-frames User Option

This variable controls whether `display-buffer` makes new frames. If it is non-`nil`, `display-buffer` looks for an existing window already displaying the desired buffer, on any visible frame. If it finds one, it returns that window. Otherwise it makes a new frame. The variables `pop-up-windows` and `split-height-threshold` do not matter if `pop-up-frames` is non-`nil`.

If `pop-up-frames` is `nil`, then `display-buffer` either splits a window or reuses one.

See Chapter 28 [Frames], page 525, for more information.

pop-up-frame-function Variable

This variable specifies how to make a new frame if `pop-up-frames` is non-`nil`.

Its value should be a function of no arguments. When `display-buffer` makes a new frame, it does so by calling that function, which should return a frame. The default value of the variable is a function that creates a frame using parameters from `pop-up-frame-alist`.

pop-up-frame-alist Variable

This variable holds an alist specifying frame parameters used when `display-buffer` makes a new frame. See Section 28.3 [Frame Parameters], page 527, for more information about frame parameters.

special-display-buffer-names User Option

A list of buffer names for buffers that should be displayed specially. If the buffer's name is in this list, `display-buffer` handles the buffer specially.

By default, special display means to give the buffer a dedicated frame.

If an element is a list, instead of a string, then the CAR of the list is the buffer name, and the rest of the list says how to create the frame. There are two possibilities for the rest of the list. It can be an alist, specifying frame parameters, or it can contain a function and arguments to give to it. (The function's first argument is always the buffer to be displayed; the arguments from the list come after that.)

special-display-regexps User Option

A list of regular expressions that specify buffers that should be displayed specially. If the buffer's name matches any of the regular expressions in this list, `display-buffer` handles the buffer specially.

By default, special display means to give the buffer a dedicated frame.

If an element is a list, instead of a string, then the CAR of the list is the regular expression, and the rest of the list says how to create the frame. See above, under `special-display-buffer-names`.

special-display-function Variable

This variable holds the function to call to display a buffer specially. It receives the buffer as an argument, and should return the window in which it is displayed.

The default value of this variable is `special-display-popup-frame`.

special-display-popup-frame *buffer* Function

This function makes *buffer* visible in a frame of its own. If *buffer* is already displayed in a window in some frame, it makes the frame visible and raises it, to use that window. Otherwise, it creates a frame that will be dedicated to *buffer*.

This function uses an existing window displaying *buffer* whether or not it is in a frame of its own; but if you set up the above variables in your init file, before *buffer* was created, then presumably the window was previously made by this function.

special-display-frame-alist User Option

This variable holds frame parameters for `special-display-popup-frame` to use when it creates a frame.

same-window-buffer-names User Option
A list of buffer names for buffers that should be displayed in the selected window. If the buffer's name is in this list, `display-buffer` handles the buffer by switching to it in the selected window.

same-window-regexps User Option
A list of regular expressions that specify buffers that should be displayed in the selected window. If the buffer's name matches any of the regular expressions in this list, `display-buffer` handles the buffer by switching to it in the selected window.

display-buffer-function Variable
This variable is the most flexible way to customize the behavior of `display-buffer`. If it is non-`nil`, it should be a function that `display-buffer` calls to do the work. The function should accept two arguments, the same two arguments that `display-buffer` received. It should choose or create a window, display the specified buffer, and then return the window.

This hook takes precedence over all the other options and hooks described above.

A window can be marked as "dedicated" to its buffer. Then `display-buffer` will not try to use that window to display any other buffer.

window-dedicated-p *window* Function
This function returns `t` if *window* is marked as dedicated; otherwise `nil`.

set-window-dedicated-p *window flag* Function
This function marks *window* as dedicated if *flag* is non-`nil`, and nondedicated otherwise.

27.9 Windows and Point

Each window has its own value of point, independent of the value of point in other windows displaying the same buffer. This makes it useful to have multiple windows showing one buffer.

- The window point is established when a window is first created; it is initialized from the buffer's point, or from the window point of another window opened on the buffer if such a window exists.
- Selecting a window sets the value of point in its buffer from the window's value of point. Conversely, deselecting a window sets the window's value of point from that of the buffer. Thus, when you switch between windows that display a given buffer, the point value for the selected window is in effect in the buffer, while the point values for the other windows are stored in those windows.

- As long as the selected window displays the current buffer, the window's point and the buffer's point always move together; they remain equal.
- See Chapter 29 [Positions], page 551, for more details on buffer positions.

As far as the user is concerned, point is where the cursor is, and when the user switches to another buffer, the cursor jumps to the position of point in that buffer.

window-point *window* Function

This function returns the current position of point in *window*. For a nonselected window, this is the value point would have (in that window's buffer) if that window were selected.

When *window* is the selected window and its buffer is also the current buffer, the value returned is the same as point in that buffer.

Strictly speaking, it would be more correct to return the "top-level" value of point, outside of any `save-excursion` forms. But that value is hard to find.

set-window-point *window position* Function

This function positions point in *window* at position *position* in *window*'s buffer.

27.10 The Window Start Position

Each window contains a marker used to keep track of a buffer position that specifies where in the buffer display should start. This position is called the *display-start* position of the window (or just the *start*). The character after this position is the one that appears at the upper left corner of the window. It is usually, but not inevitably, at the beginning of a text line.

window-start &optional *window* Function

This function returns the display-start position of window *window*. If *window* is `nil`, the selected window is used. For example,

```
(window-start)
     ⇒ 7058
```

When you create a window, or display a different buffer in it, the display-start position is set to a display-start position recently used for the same buffer, or 1 if the buffer doesn't have any.

Redisplay updates the window-start position (if you have not specified it explicitly since the previous redisplay) so that point appears on the screen. Nothing except redisplay automatically changes the window-start position; if you move point, do not expect the window-start position to change in response until after the next redisplay.

For a realistic example of using `window-start`, see the description of `count-lines` in Section 29.2.4 [Text Lines], page 554.

window-end &optional *window update* Function

This function returns the position of the end of the display in window *window*. If *window* is `nil`, the selected window is used.

Simply changing the buffer text or moving point does not update the value that `window-end` returns. The value is updated only when Emacs redisplays and redisplay completes without being preempted.

If the last redisplay of *window* was preempted, and did not finish, Emacs does not know the position of the end of display in that window. In that case, this function returns `nil`.

If *update* is non-`nil`, `window-end` always returns an up-to-date value for where the window ends. If the saved value is valid, `window-end` returns that; otherwise it computes the correct value by scanning the buffer text.

set-window-start *window position* &optional *noforce* Function

This function sets the display-start position of *window* to *position* in *window*'s buffer. It returns *position*.

The display routines insist that the position of point be visible when a buffer is displayed. Normally, they change the display-start position (that is, scroll the window) whenever necessary to make point visible. However, if you specify the start position with this function using `nil` for *noforce*, it means you want display to start at *position* even if that would put the location of point off the screen. If this does place point off screen, the display routines move point to the left margin on the middle line in the window.

For example, if point is 1 and you set the start of the window to 2, then point would be "above" the top of the window. The display routines will automatically move point if it is still 1 when redisplay occurs. Here is an example:

```
;; Here is what 'foo' looks like before executing
;;    the set-window-start expression.

---------- Buffer: foo ----------
*This is the contents of buffer foo.
2
3
4
5
6
---------- Buffer: foo ----------

(set-window-start
 (selected-window)
 (1+ (window-start)))
⇒ 2
```

```
;; Here is what 'foo' looks like after executing
;;    the set-window-start expression.
---------- Buffer: foo ----------
his is the contents of buffer foo.
2
3
*4
5
6
---------- Buffer: foo ----------
```

If *noforce* is non-`nil`, and *position* would place point off screen at the next redisplay, then redisplay computes a new window-start position that works well with point, and thus *position* is not used.

pos-visible-in-window-p &optional *position window* Function

This function returns `t` if *position* is within the range of text currently visible on the screen in *window*. It returns `nil` if *position* is scrolled vertically out of view. The argument *position* defaults to the current position of point; *window*, to the selected window. Here is an example:

```
(or (pos-visible-in-window-p
     (point) (selected-window))
    (recenter 0))
```

The `pos-visible-in-window-p` function considers only vertical scrolling. If *position* is out of view only because *window* has been scrolled horizontally, `pos-visible-in-window-p` returns `t`. See Section 27.12 [Horizontal Scrolling], page 515.

27.11 Vertical Scrolling

Vertical scrolling means moving the text up or down in a window. It works by changing the value of the window's display-start location. It may also change the value of `window-point` to keep it on the screen.

In the commands `scroll-up` and `scroll-down`, the directions "up" and "down" refer to the motion of the text in the buffer at which you are looking through the window. Imagine that the text is written on a long roll of paper and that the scrolling commands move the paper up and down. Thus, if you are looking at text in the middle of a buffer and repeatedly call `scroll-down`, you will eventually see the beginning of the buffer.

Some people have urged that the opposite convention be used: they imagine that the window moves over text that remains in place. Then "down" commands would take you to the end of the buffer. This view is more consistent with the actual relationship between windows and the text in the buffer, but it is less like what the user sees. The position of a window on the terminal does not move, and short scrolling commands clearly move the

text up or down on the screen. We have chosen names that fit the user's point of view.

The scrolling functions (aside from `scroll-other-window`) have unpredictable results if the current buffer is different from the buffer that is displayed in the selected window. See Section 26.2 [Current Buffer], page 479.

scroll-up &optional *count* Command

This function scrolls the text in the selected window upward *count* lines. If *count* is negative, scrolling is actually downward.

If *count* is `nil` (or omitted), then the length of scroll is `next-screen-context-lines` lines less than the usable height of the window (not counting its mode line).

`scroll-up` returns `nil`.

scroll-down &optional *count* Command

This function scrolls the text in the selected window downward *count* lines. If *count* is negative, scrolling is actually upward.

If *count* is omitted or `nil`, then the length of the scroll is `next-screen-context-lines` lines less than the usable height of the window (not counting its mode line).

`scroll-down` returns `nil`.

scroll-other-window &optional *count* Command

This function scrolls the text in another window upward *count* lines. Negative values of *count*, or `nil`, are handled as in `scroll-up`.

You can specify a buffer to scroll with the variable `other-window-scroll-buffer`. When the selected window is the minibuffer, the next window is normally the one at the top left corner. You can specify a different window to scroll with the variable `minibuffer-scroll-window`. This variable has no effect when any other window is selected. See Section 19.9 [Minibuffer Misc], page 316.

When the minibuffer is active, it is the next window if the selected window is the one at the bottom right corner. In this case, `scroll-other-window` attempts to scroll the minibuffer. If the minibuffer contains just one line, it has nowhere to scroll to, so the line reappears after the echo area momentarily displays the message "Beginning of buffer".

other-window-scroll-buffer Variable

If this variable is non-`nil`, it tells `scroll-other-window` which buffer to scroll.

scroll-margin User Option

This option specifies the size of the scroll margin—a minimum number of lines between point and the top or bottom of a window. Whenever

point gets within this many lines of the top or bottom of the window, the window scrolls automatically (if possible) to move point out of the margin, closer to the center of the window.

scroll-conservatively User Option

This variable controls how scrolling is done automatically when point moves off the screen (or into the scroll margin). If the value is zero, then redisplay scrolls the text to center point vertically in the window. If the value is a positive integer *n*, then redisplay scrolls the window up to *n* lines in either direction, if that will bring point back into view. Otherwise, it centers point. The default value is zero.

scroll-step User Option

This variable is an older variant of `scroll-conservatively`. The difference is that it if its value is *n*, that permits scrolling only by precisely *n* lines, not a smaller number. This feature does not work with `scroll-margin`. The default value is zero.

scroll-preserve-screen-position User Option

If this option is non-`nil`, the scroll functions move point so that the vertical position of the cursor is unchanged, when that is possible.

next-screen-context-lines User Option

The value of this variable is the number of lines of continuity to retain when scrolling by full screens. For example, `scroll-up` with an argument of `nil` scrolls so that this many lines at the bottom of the window appear instead at the top. The default value is `2`.

recenter &optional *count* Command

This function scrolls the selected window to put the text where point is located at a specified vertical position within the window.

If *count* is a nonnegative number, it puts the line containing point *count* lines down from the top of the window. If *count* is a negative number, then it counts upward from the bottom of the window, so that -1 stands for the last usable line in the window. If *count* is a non-`nil` list, then it stands for the line in the middle of the window.

If *count* is `nil`, `recenter` puts the line containing point in the middle of the window, then clears and redisplays the entire selected frame.

When `recenter` is called interactively, *count* is the raw prefix argument. Thus, typing `C-u` as the prefix sets the *count* to a non-`nil` list, while typing `C-u 4` sets *count* to 4, which positions the current line four lines from the top.

With an argument of zero, `recenter` positions the current line at the top of the window. This action is so handy that some people make a separate key binding to do this. For example,

```
(defun line-to-top-of-window ()
  "Scroll current line to top of window.
Replaces three keystroke sequence C-u 0 C-l."
  (interactive)
  (recenter 0))

(global-set-key [kp-multiply] 'line-to-top-of-window)
```

27.12 Horizontal Scrolling

Because we read English from left to right in the "inner loop", and from top to bottom in the "outer loop", horizontal scrolling is not like vertical scrolling. Vertical scrolling involves selection of a contiguous portion of text to display, but horizontal scrolling causes part of each line to go off screen. The amount of horizontal scrolling is therefore specified as a number of columns rather than as a position in the buffer. It has nothing to do with the display-start position returned by `window-start`.

Usually, no horizontal scrolling is in effect; then the leftmost column is at the left edge of the window. In this state, scrolling to the right is meaningless, since there is no data to the left of the screen to be revealed by it; so this is not allowed. Scrolling to the left is allowed; it scrolls the first columns of text off the edge of the window and can reveal additional columns on the right that were truncated before. Once a window has a nonzero amount of leftward horizontal scrolling, you can scroll it back to the right, but only so far as to reduce the net horizontal scroll to zero. There is no limit to how far left you can scroll, but eventually all the text will disappear off the left edge.

scroll-left *count* Command

This function scrolls the selected window *count* columns to the left (or to the right if *count* is negative). The return value is the total amount of leftward horizontal scrolling in effect after the change—just like the value returned by `window-hscroll` (below).

scroll-right *count* Command

This function scrolls the selected window *count* columns to the right (or to the left if *count* is negative). The return value is the total amount of leftward horizontal scrolling in effect after the change—just like the value returned by `window-hscroll` (below).

Once you scroll a window as far right as it can go, back to its normal position where the total leftward scrolling is zero, attempts to scroll any farther right have no effect.

window-hscroll &optional *window* Function

This function returns the total leftward horizontal scrolling of *window*—the number of columns by which the text in *window* is scrolled left past the left margin.

The value is never negative. It is zero when no horizontal scrolling has been done in *window* (which is usually the case).

If *window* is `nil`, the selected window is used.

```
(window-hscroll)
     ⇒ 0
(scroll-left 5)
     ⇒ 5
(window-hscroll)
     ⇒ 5
```

set-window-hscroll *window columns* Function

This function sets the number of columns from the left margin that *window* is scrolled from the value of *columns*. The argument *columns* should be zero or positive; if not, it is taken as zero.

The value returned is *columns*.

```
(set-window-hscroll (selected-window) 10)
     ⇒ 10
```

Here is how you can determine whether a given position *position* is off the screen due to horizontal scrolling:

```
(defun hscroll-on-screen (window position)
  (save-excursion
    (goto-char position)
    (and
     (>= (- (current-column) (window-hscroll window)) 0)
     (< (- (current-column) (window-hscroll window))
        (window-width window)))))
```

27.13 The Size of a Window

An Emacs window is rectangular, and its size information consists of the height (the number of lines) and the width (the number of character positions in each line). The mode line is included in the height. But the width does not count the scroll bar or the column of '|' characters that separates side-by-side windows.

The following three functions return size information about a window:

window-height &optional *window* Function

This function returns the number of lines in *window*, including its mode line. If *window* fills its entire frame, this is typically one less than the

value of `frame-height` on that frame (since the last line is always reserved for the minibuffer).

If *window* is `nil`, the function uses the selected window.

```
(window-height)
     ⇒ 23
(split-window-vertically)
     ⇒ #<window 4 on windows.texi>
(window-height)
     ⇒ 11
```

window-width &optional *window* Function

This function returns the number of columns in *window*. If *window* fills its entire frame, this is the same as the value of `frame-width` on that frame. The width does not include the window's scroll bar or the column of '`|`' characters that separates side-by-side windows.

If *window* is `nil`, the function uses the selected window.

```
(window-width)
     ⇒ 80
```

window-edges &optional *window* Function

This function returns a list of the edge coordinates of *window*. If *window* is `nil`, the selected window is used.

The order of the list is (*left top right bottom*), all elements relative to 0, 0 at the top left corner of the frame. The element *right* of the value is one more than the rightmost column used by *window*, and *bottom* is one more than the bottommost row used by *window* and its mode-line.

When you have side-by-side windows, the right edge value for a window with a neighbor on the right includes the width of the separator between the window and that neighbor. This separator may be a column of '`|`' characters or it may be a scroll bar. Since the width of the window does not include this separator, the width does not equal the difference between the right and left edges in this case.

Here is the result obtained on a typical 24-line terminal with just one window:

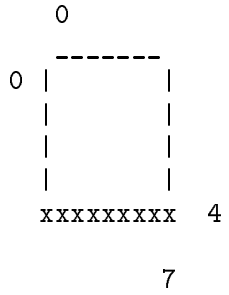
```
(window-edges (selected-window))
     ⇒ (0 0 80 23)
```

The bottom edge is at line 23 because the last line is the echo area.

If *window* is at the upper left corner of its frame, then *bottom* is the same as the value of `(window-height)`, *right* is almost the same as the value of `(window-width)`[1], and *top* and *left* are zero. For example, the edges

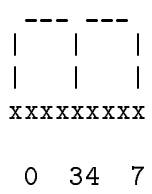
[1] They are not exactly equal because *right* includes the vertical separator line or scroll bar, while `(window-width)` does not.

of the following window are '`0 0 5 8`'. Assuming that the frame has more than 8 columns, the last column of the window (column 7) holds a border rather than text. The last row (row 4) holds the mode line, shown here with '`xxxxxxxxx`'.



When there are side-by-side windows, any window not at the right edge of its frame has a separator in its last column or columns. The separator counts as one or two columns in the width of the window. A window never includes a separator on its left, since that belongs to the window to the left.

In the following example, let's suppose that the frame is 7 columns wide. Then the edges of the left window are '`0 0 4 3`' and the edges of the right window are '`4 0 7 3`'.



27.14 Changing the Size of a Window

The window size functions fall into two classes: high-level commands that change the size of windows and low-level functions that access window size. Emacs does not permit overlapping windows or gaps between windows, so resizing one window affects other windows.

enlarge-window *size* &optional *horizontal* Command

This function makes the selected window *size* lines taller, stealing lines from neighboring windows. It takes the lines from one window at a time until that window is used up, then takes from another. If a window from which lines are stolen shrinks below `window-min-height` lines, that window disappears.

If *horizontal* is non-`nil`, this function makes *window* wider by *size* columns, stealing columns instead of lines. If a window from which

columns are stolen shrinks below `window-min-width` columns, that window disappears.

If the requested size would exceed that of the window's frame, then the function makes the window occupy the entire height (or width) of the frame.

If *size* is negative, this function shrinks the window by $-$*size* lines or columns. If that makes the window smaller than the minimum size (`window-min-height` and `window-min-width`), `enlarge-window` deletes the window.

`enlarge-window` returns `nil`.

enlarge-window-horizontally *columns* Command

This function makes the selected window *columns* wider. It could be defined as follows:

```
(defun enlarge-window-horizontally (columns)
  (enlarge-window columns t))
```

shrink-window *size* &optional *horizontal* Command

This function is like `enlarge-window` but negates the argument *size*, making the selected window smaller by giving lines (or columns) to the other windows. If the window shrinks below `window-min-height` or `window-min-width`, then it disappears.

If *size* is negative, the window is enlarged by $-$*size* lines or columns.

shrink-window-horizontally *columns* Command

This function makes the selected window *columns* narrower. It could be defined as follows:

```
(defun shrink-window-horizontally (columns)
  (shrink-window columns t))
```

shrink-window-if-larger-than-buffer *window* Command

This command shrinks *window* to be as small as possible while still showing the full contents of its buffer—but not less than `window-min-height` lines.

However, the command does nothing if the window is already too small to display the whole text of the buffer, or if part of the contents are currently scrolled off screen, or if the window is not the full width of its frame, or if the window is the only window in its frame.

The following two variables constrain the window-size-changing functions to a minimum height and width.

window-min-height User Option
The value of this variable determines how short a window may become before it is automatically deleted. Making a window smaller than `window-min-height` automatically deletes it, and no window may be created shorter than this. The absolute minimum height is two (allowing one line for the mode line, and one line for the buffer display). Actions that change window sizes reset this variable to two if it is less than two. The default value is 4.

window-min-width User Option
The value of this variable determines how narrow a window may become before it is automatically deleted. Making a window smaller than `window-min-width` automatically deletes it, and no window may be created narrower than this. The absolute minimum width is one; any value below that is ignored. The default value is 10.

27.15 Coordinates and Windows

This section describes how to relate screen coordinates to windows.

window-at *x y* &optional *frame* Function
This function returns the window containing the specified cursor position in the frame *frame*. The coordinates *x* and *y* are measured in characters and count from the top left corner of the frame. If they are out of range, `window-at` returns `nil`.

If you omit *frame*, the selected frame is used.

coordinates-in-window-p *coordinates window* Function
This function checks whether a particular frame position falls within the window *window*.

The argument *coordinates* is a cons cell of the form `(x . y)`. The coordinates *x* and *y* are measured in characters, and count from the top left corner of the screen or frame.

The value returned by `coordinates-in-window-p` is non-`nil` if the coordinates are inside *window*. The value also indicates what part of the window the position is in, as follows:

`(`*relx* `.` *rely*`)`
: The coordinates are inside *window*. The numbers *relx* and *rely* are the equivalent window-relative coordinates for the specified position, counting from 0 at the top left corner of the window.

`mode-line`
: The coordinates are in the mode line of *window*.

`vertical-split`
: The coordinates are in the vertical line between *window* and its neighbor to the right. This value occurs only if the window doesn't have a scroll bar; positions in a scroll bar are considered outside the window.

`nil`
: The coordinates are not in any part of *window*.

The function `coordinates-in-window-p` does not require a frame as argument because it always uses the frame that *window* is on.

27.16 Window Configurations

A *window configuration* records the entire layout of one frame—all windows, their sizes, which buffers they contain, what part of each buffer is displayed, and the values of point and the mark. You can bring back an entire previous layout by restoring a window configuration previously saved.

If you want to record all frames instead of just one, use a frame configuration instead of a window configuration. See Section 28.12 [Frame Configurations], page 541.

current-window-configuration Function

This function returns a new object representing the selected frame's current window configuration, including the number of windows, their sizes and current buffers, which window is the selected window, and for each window the displayed buffer, the display-start position, and the positions of point and the mark. It also includes the values of `window-min-height`, `window-min-width` and `minibuffer-scroll-window`. An exception is made for point in the current buffer, whose value is not saved.

set-window-configuration *configuration* Function

This function restores the configuration of windows and buffers as specified by *configuration*. The argument *configuration* must be a value that was previously returned by `current-window-configuration`. This configuration is restored in the frame from which *configuration* was made, whether that frame is selected or not. This always counts as a window size change and triggers execution of the `window-size-change-functions` (see Section 27.17 [Window Hooks], page 523), because `set-window-configuration` doesn't know how to tell whether the new configuration actually differs from the old one.

If the frame which *configuration* was saved from is dead, all this function does is restore the three variables `window-min-height`, `window-min-width` and `minibuffer-scroll-window`.

Here is a way of using this function to get the same effect as `save-window-excursion`:

```
(let ((config (current-window-configuration)))
  (unwind-protect
      (progn (split-window-vertically nil)
             ...)
    (set-window-configuration config)))
```

save-window-excursion *forms...* Special Form

This special form records the window configuration, executes *forms* in sequence, then restores the earlier window configuration. The window configuration includes the value of point and the portion of the buffer that is visible. It also includes the choice of selected window. However, it does not include the value of point in the current buffer; use `save-excursion` also, if you wish to preserve that.

Don't use this construct when `save-selected-window` is all you need.

Exit from `save-window-excursion` always triggers execution of the `window-size-change-functions`. (It doesn't know how to tell whether the restored configuration actually differs from the one in effect at the end of the *forms*.)

The return value is the value of the final form in *forms*. For example:

```
(split-window)
     ⇒ #<window 25 on control.texi>
(setq w (selected-window))
     ⇒ #<window 19 on control.texi>
(save-window-excursion
  (delete-other-windows w)
  (switch-to-buffer "foo")
  'do-something)
     ⇒ do-something
     ;; The screen is now split again.
```

window-configuration-p *object* Function

This function returns `t` if *object* is a window configuration.

compare-window-configurations *config1 config2* Function

This function compares two window configurations as regards the structure of windows, but ignores the values of point and mark and the saved scrolling positions—it can return `t` even if those aspects differ.

The function `equal` can also compare two window configurations; it regards configurations as unequal if they differ in any respect, even a saved point or mark.

Primitives to look inside of window configurations would make sense, but none are implemented. It is not clear they are useful enough to be worth implementing.

27.17 Hooks for Window Scrolling and Changes

This section describes how a Lisp program can take action whenever a window displays a different part of its buffer or a different buffer. There are three actions that can change this: scrolling the window, switching buffers in the window, and changing the size of the window. The first two actions run `window-scroll-functions`; the last runs `window-size-change-functions`. The paradigmatic use of these hooks is in the implementation of Lazy Lock mode; see section "Font Lock Support Modes" in *The GNU Emacs Manual*.

window-scroll-functions Variable

This variable holds a list of functions that Emacs should call before redisplaying a window with scrolling. It is not a normal hook, because each function is called with two arguments: the window, and its new display-start position.

Displaying a different buffer in the window also runs these functions.

These functions must be careful in using `window-end` (see Section 27.10 [Window Start], page 510); if you need an up-to-date value, you must use the *update* argument to ensure you get it.

window-size-change-functions Variable

This variable holds a list of functions to be called if the size of any window changes for any reason. The functions are called just once per redisplay, and just once for each frame on which size changes have occurred.

Each function receives the frame as its sole argument. There is no direct way to find out which windows on that frame have changed size, or precisely how. However, if a size-change function records, at each call, the existing windows and their sizes, it can also compare the present sizes and the previous sizes.

Creating or deleting windows counts as a size change, and therefore causes these functions to be called. Changing the frame size also counts, because it changes the sizes of the existing windows.

It is not a good idea to use `save-window-excursion` (see Section 27.16 [Window Configurations], page 521) in these functions, because that always counts as a size change, and it would cause these functions to be called over and over. In most cases, `save-selected-window` (see Section 27.4 [Selecting Windows], page 500) is what you need here.

redisplay-end-trigger-functions Variable

This abnormal hook is run whenever redisplay in a window uses text that extends past a specified end trigger position. You set the end trigger position with the function `set-window-redisplay-end-trigger`. The functions are called with two arguments: the window, and the end trigger position. Storing `nil` for the end trigger position turns off the feature,

and the trigger value is automatically reset to `nil` just after the hook is run.

set-window-redisplay-end-trigger *window position* Function
This function sets *window*'s end trigger position at *position*.

window-redisplay-end-trigger *window* Function
This function returns *window*'s current end trigger position.

window-configuration-change-hook Variable
A normal hook that is run every time you change the window configuration of an existing frame. This includes splitting or deleting windows, changing the sizes of windows, or displaying a different buffer in a window. The frame whose window configuration has changed is the selected frame when this hook runs.

28 Frames

A *frame* is a rectangle on the screen that contains one or more Emacs windows. A frame initially contains a single main window (plus perhaps a minibuffer window), which you can subdivide vertically or horizontally into smaller windows.

When Emacs runs on a text-only terminal, it starts with one *terminal frame*. If you create additional ones, Emacs displays one and only one at any given time—on the terminal screen, of course.

When Emacs communicates directly with a supported window system, such as X Windows, it does not have a terminal frame; instead, it starts with a single *window frame*, but you can create more, and Emacs can display several such frames at once as is usual for window systems.

framep *object* Function

This predicate returns `t` if *object* is a frame, and `nil` otherwise.

See Chapter 38 [Display], page 739, for information about the related topic of controlling Emacs redisplay.

28.1 Creating Frames

To create a new frame, call the function `make-frame`.

make-frame &optional *alist* Function

This function creates a new frame. If you are using a supported window system, it makes a window frame; otherwise, it makes a terminal frame.

The argument is an alist specifying frame parameters. Any parameters not mentioned in *alist* default according to the value of the variable `default-frame-alist`; parameters not specified even there default from the standard X resources or whatever is used instead on your system.

The set of possible parameters depends in principle on what kind of window system Emacs uses to display its frames. See Section 28.3.3 [Window Frame Parameters], page 528, for documentation of individual parameters you can specify.

before-make-frame-hook Variable

A normal hook run by `make-frame` before it actually creates the frame.

after-make-frame-hook Variable

An abnormal hook run by `make-frame` after it creates the frame. Each function in `after-make-frame-hook` receives one argument, the frame just created.

28.2 Multiple Displays

A single Emacs can talk to more than one X display. Initially, Emacs uses just one display—the one chosen with the `DISPLAY` environment variable or with the '`--display`' option (see section "Initial Options" in *The GNU Emacs Manual*). To connect to another display, use the command `make-frame-on-display` or specify the `display` frame parameter when you create the frame.

Emacs treats each X server as a separate terminal, giving each one its own selected frame and its own minibuffer windows.

A few Lisp variables are *terminal-local*; that is, they have a separate binding for each terminal. The binding in effect at any time is the one for the terminal that the currently selected frame belongs to. These variables include `default-minibuffer-frame`, `defining-kbd-macro`, `last-kbd-macro`, and `system-key-alist`. They are always terminal-local, and can never be buffer-local (see Section 10.10 [Buffer-Local Variables], page 161) or frame-local.

A single X server can handle more than one screen. A display name '*host*:*server*.*screen*' has three parts; the last part specifies the screen number for a given server. When you use two screens belonging to one server, Emacs knows by the similarity in their names that they share a single keyboard, and it treats them as a single terminal.

make-frame-on-display *display* &optional *parameters* Command
This creates a new frame on display *display*, taking the other frame parameters from *parameters*. Aside from the *display* argument, it is like `make-frame` (see Section 28.1 [Creating Frames], page 525).

x-display-list Function
This returns a list that indicates which X displays Emacs has a connection to. The elements of the list are strings, and each one is a display name.

x-open-connection *display* &optional *xrm-string* Function
This function opens a connection to the X display *display*. It does not create a frame on that display, but it permits you to check that communication can be established with that display.

The optional argument *xrm-string*, if not `nil`, is a string of resource names and values, in the same format used in the '`.Xresources`' file. The values you specify override the resource values recorded in the X server itself; they apply to all Emacs frames created on this display. Here's an example of what this string might look like:

```
"*BorderWidth: 3\n*InternalBorder: 2\n"
```

See Section 28.22 [Resources], page 548.

x-close-connection *display* Function
This function closes the connection to display *display*. Before you can do this, you must first delete all the frames that were open on that display (see Section 28.5 [Deleting Frames], page 535).

28.3 Frame Parameters

A frame has many parameters that control its appearance and behavior. Just what parameters a frame has depends on what display mechanism it uses.

Frame parameters exist for the sake of window systems. A terminal frame has a few parameters, mostly for compatibility's sake; only the `height`, `width`, `name`, `title`, `buffer-list` and `buffer-predicate` parameters do something special.

28.3.1 Access to Frame Parameters

These functions let you read and change the parameter values of a frame.

frame-parameters *frame* Function
The function `frame-parameters` returns an alist listing all the parameters of *frame* and their values.

modify-frame-parameters *frame alist* Function
This function alters the parameters of frame *frame* based on the elements of *alist*. Each element of *alist* has the form (*parm* . *value*), where *parm* is a symbol naming a parameter. If you don't mention a parameter in *alist*, its value doesn't change.

28.3.2 Initial Frame Parameters

You can specify the parameters for the initial startup frame by setting `initial-frame-alist` in your '`.emacs`' file.

initial-frame-alist Variable
This variable's value is an alist of parameter values used when creating the initial window frame. You can set this variable to specify the appearance of the initial frame without altering subsequent frames. Each element has the form:

(*parameter* . *value*)

Emacs creates the initial frame before it reads your '`~/.emacs`' file. After reading that file, Emacs checks `initial-frame-alist`, and applies the parameter settings in the altered value to the already created initial frame.

If these settings affect the frame geometry and appearance, you'll see the frame appear with the wrong ones and then change to the specified ones.

If that bothers you, you can specify the same geometry and appearance with X resources; those do take affect before the frame is created. See section "X Resources" in *The GNU Emacs Manual*.

X resource settings typically apply to all frames. If you want to specify some X resources solely for the sake of the initial frame, and you don't want them to apply to subsequent frames, here's how to achieve this. Specify parameters in `default-frame-alist` to override the X resources for subsequent frames; then, to prevent these from affecting the initial frame, specify the same parameters in `initial-frame-alist` with values that match the X resources.

If these parameters specify a separate minibuffer-only frame with `(minibuffer . nil)`, and you have not created one, Emacs creates one for you.

minibuffer-frame-alist Variable

This variable's value is an alist of parameter values used when creating an initial minibuffer-only frame—if such a frame is needed, according to the parameters for the main initial frame.

default-frame-alist Variable

This is an alist specifying default values of frame parameters for all Emacs frames—the first frame, and subsequent frames. When using the X Window System, you can get the same results by means of X resources in many cases.

See also `special-display-frame-alist`, in Section 27.8 [Choosing Window], page 506.

If you use options that specify window appearance when you invoke Emacs, they take effect by adding elements to `default-frame-alist`. One exception is '`-geometry`', which adds the specified position to `initial-frame-alist` instead. See section "Command Arguments" in *The GNU Emacs Manual*.

28.3.3 Window Frame Parameters

Just what parameters a frame has depends on what display mechanism it uses. Here is a table of the parameters that have special meanings in a window frame; of these, `name`, `title`, `height`, `width`, `buffer-list` and `buffer-predicate` provide meaningful information in terminal frames.

`display` The display on which to open this frame. It should be a string of the form "*host*`:`*dpy*`.`*screen*", just like the `DISPLAY` environment variable.

`title` If a frame has a non-`nil` title, it appears in the window system's border for the frame, and also in the mode line of windows in

that frame if `mode-line-frame-identification` uses '`%F`' (see Section 22.3.3 [%-Constructs], page 410). This is normally the case when Emacs is not using a window system, and can only display one frame at a time. See Section 28.4 [Frame Titles], page 534.

`name` The name of the frame. The frame name serves as a default for the frame title, if the `title` parameter is unspecified or `nil`. If you don't specify a name, Emacs sets the frame name automatically (see Section 28.4 [Frame Titles], page 534).

If you specify the frame name explicitly when you create the frame, the name is also used (instead of the name of the Emacs executable) when looking up X resources for the frame.

`left` The screen position of the left edge, in pixels, with respect to the left edge of the screen. The value may be a positive number *pos*, or a list of the form `(+` *pos*`)` which permits specifying a negative *pos* value.

A negative number $-pos$, or a list of the form `(-` *pos*`)`, actually specifies the position of the right edge of the window with respect to the right edge of the screen. A positive value of *pos* counts toward the left. **Reminder:** if the parameter is a negative integer $-pos$, then *pos* is positive.

Some window managers ignore program-specified positions. If you want to be sure the position you specify is not ignored, specify a non-`nil` value for the `user-position` parameter as well.

`top` The screen position of the top edge, in pixels, with respect to the top edge of the screen. The value may be a positive number *pos*, or a list of the form `(+` *pos*`)` which permits specifying a negative *pos* value.

A negative number $-pos$, or a list of the form `(-` *pos*`)`, actually specifies the position of the bottom edge of the window with respect to the bottom edge of the screen. A positive value of *pos* counts toward the top. **Reminder:** if the parameter is a negative integer $-pos$, then *pos* is positive.

Some window managers ignore program-specified positions. If you want to be sure the position you specify is not ignored, specify a non-`nil` value for the `user-position` parameter as well.

`icon-left`

The screen position of the left edge *of the frame's icon*, in pixels, counting from the left edge of the screen. This takes effect if and when the frame is iconified.

`icon-top` The screen position of the top edge *of the frame's icon*, in pixels, counting from the top edge of the screen. This takes effect if and when the frame is iconified.

`user-position`
When you create a frame and specify its screen position with the `left` and `top` parameters, use this parameter to say whether the specified position was user-specified (explicitly requested in some way by a human user) or merely program-specified (chosen by a program). A non-`nil` value says the position was user-specified.

Window managers generally heed user-specified positions, and some heed program-specified positions too. But many ignore program-specified positions, placing the window in a default fashion or letting the user place it with the mouse. Some window managers, including `twm`, let the user specify whether to obey program-specified positions or ignore them.

When you call `make-frame`, you should specify a non-`nil` value for this parameter if the values of the `left` and `top` parameters represent the user's stated preference; otherwise, use `nil`.

`height` The height of the frame contents, in characters. (To get the height in pixels, call `frame-pixel-height`; see Section 28.3.4 [Size and Position], page 532.)

`width` The width of the frame contents, in characters. (To get the height in pixels, call `frame-pixel-width`; see Section 28.3.4 [Size and Position], page 532.)

`window-id`
The number of the window-system window used by the frame.

`minibuffer`
Whether this frame has its own minibuffer. The value `t` means yes, `nil` means no, `only` means this frame is just a minibuffer. If the value is a minibuffer window (in some other frame), the new frame uses that minibuffer.

`buffer-predicate`
The buffer-predicate function for this frame. The function `other-buffer` uses this predicate (from the selected frame) to decide which buffers it should consider, if the predicate is not `nil`. It calls the predicate with one argument, a buffer, once for each buffer; if the predicate returns a non-`nil` value, it considers that buffer.

`buffer-list`
A list of buffers that have been selected in this frame, ordered most-recently-selected first.

`font` The name of the font for displaying text in the frame. This is a string, either a valid font name for your system or the name of an Emacs fontset (see Section 28.20 [Fontsets], page 546).

`auto-raise`
Whether selecting the frame raises it (non-`nil` means yes).

`auto-lower`
Whether deselecting the frame lowers it (non-`nil` means yes).

`vertical-scroll-bars`
Whether the frame has scroll bars for vertical scrolling, and which side of the frame they should be on. The possible values are `left`, `right`, and `nil` for no scroll bars.

`horizontal-scroll-bars`
Whether the frame has scroll bars for horizontal scrolling (non-`nil` means yes). (Horizontal scroll bars are not currently implemented.)

`scroll-bar-width`
The width of the vertical scroll bar, in pixels.

`icon-type`
The type of icon to use for this frame when it is iconified. If the value is a string, that specifies a file containing a bitmap to use. Any other non-`nil` value specifies the default bitmap icon (a picture of a gnu); `nil` specifies a text icon.

`icon-name`
The name to use in the icon for this frame, when and if the icon appears. If this is `nil`, the frame's title is used.

`foreground-color`
The color to use for the image of a character. This is a string; the window system defines the meaningful color names.

If you set the `foreground-color` frame parameter, you should call `frame-update-face-colors` to update faces accordingly.

`background-color`
The color to use for the background of characters.

If you set the `background-color` frame parameter, you should call `frame-update-face-colors` to update faces accordingly. See Section 38.10.4 [Face Functions], page 756.

`background-mode`
This parameter is either `dark` or `light`, according to whether the background color is a light one or a dark one.

`mouse-color`
The color for the mouse pointer.

`cursor-color`
: The color for the cursor that shows point.

`border-color`
: The color for the border of the frame.

`display-type`
: This parameter describes the range of possible colors that can be used in this frame. Its value is `color`, `grayscale` or `mono`.

`cursor-type`
: The way to display the cursor. The legitimate values are `bar`, `box`, and `(bar . width)`. The symbol `box` specifies an ordinary black box overlaying the character after point; that is the default. The symbol `bar` specifies a vertical bar between characters as the cursor. `(bar . width)` specifies a bar *width* pixels wide.

`border-width`
: The width in pixels of the window border.

`internal-border-width`
: The distance in pixels between text and border.

`unsplittable`
: If non-`nil`, this frame's window is never split automatically.

`visibility`
: The state of visibility of the frame. There are three possibilities: `nil` for invisible, `t` for visible, and `icon` for iconified. See Section 28.10 [Visibility of Frames], page 539.

`menu-bar-lines`
: The number of lines to allocate at the top of the frame for a menu bar. The default is 1. See Section 21.12.5 [Menu Bar], page 387. (In Emacs versions that use the X toolkit, there is only one menu bar line; all that matters about the number you specify is whether it is greater than zero.)

28.3.4 Frame Size And Position

You can read or change the size and position of a frame using the frame parameters `left`, `top`, `height`, and `width`. Whatever geometry parameters you don't specify are chosen by the window manager in its usual fashion.

Here are some special features for working with sizes and positions:

set-frame-position *frame left top* Function
: This function sets the position of the top left corner of *frame* to *left* and *top*. These arguments are measured in pixels, and normally count from the top left corner of the screen.

Negative parameter values position the bottom edge of the window up from the bottom edge of the screen, or the right window edge to the left of the right edge of the screen. It would probably be better if the values were always counted from the left and top, so that negative arguments would position the frame partly off the top or left edge of the screen, but it seems inadvisable to change that now.

frame-height &optional *frame* Function
frame-width &optional *frame* Function
These functions return the height and width of *frame*, measured in lines and columns. If you don't supply *frame*, they use the selected frame.

screen-height Function
screen-width Function
These functions are old aliases for `frame-height` and `frame-width`. When you are using a non-window terminal, the size of the frame is normally the same as the size of the terminal screen.

frame-pixel-height &optional *frame* Function
frame-pixel-width &optional *frame* Function
These functions return the height and width of *frame*, measured in pixels. If you don't supply *frame*, they use the selected frame.

frame-char-height &optional *frame* Function
frame-char-width &optional *frame* Function
These functions return the height and width of a character in *frame*, measured in pixels. The values depend on the choice of font. If you don't supply *frame*, these functions use the selected frame.

set-frame-size *frame cols rows* Function
This function sets the size of *frame*, measured in characters; *cols* and *rows* specify the new width and height.

To set the size based on values measured in pixels, use `frame-char-height` and `frame-char-width` to convert them to units of characters.

set-frame-height *frame lines* &optional *pretend* Function
This function resizes *frame* to a height of *lines* lines. The sizes of existing windows in *frame* are altered proportionally to fit.

If *pretend* is non-`nil`, then Emacs displays *lines* lines of output in *frame*, but does not change its value for the actual height of the frame. This is only useful for a terminal frame. Using a smaller height than the terminal actually implements may be useful to reproduce behavior observed on a smaller screen, or if the terminal malfunctions when using its whole screen. Setting the frame height "for real" does not always work, because knowing the correct actual size may be necessary for correct cursor positioning on a terminal frame.

set-frame-width *frame width* &optional *pretend* Function
This function sets the width of *frame*, measured in characters. The argument *pretend* has the same meaning as in `set-frame-height`.

The older functions `set-screen-height` and `set-screen-width` were used to specify the height and width of the screen, in Emacs versions that did not support multiple frames. They are semi-obsolete, but still work; they apply to the selected frame.

x-parse-geometry *geom* Function
The function `x-parse-geometry` converts a standard X window geometry string to an alist that you can use as part of the argument to `make-frame`.

The alist describes which parameters were specified in *geom*, and gives the values specified for them. Each element looks like (*parameter* . *value*). The possible *parameter* values are `left`, `top`, `width`, and `height`.

For the size parameters, the value must be an integer. The position parameter names `left` and `top` are not totally accurate, because some values indicate the position of the right or bottom edges instead. These are the *value* possibilities for the position parameters:

an integer A positive integer relates the left edge or top edge of the window to the left or top edge of the screen. A negative integer relates the right or bottom edge of the window to the right or bottom edge of the screen.

(`+` *position*)
This specifies the position of the left or top edge of the window relative to the left or top edge of the screen. The integer *position* may be positive or negative; a negative value specifies a position outside the screen.

(`-` *position*)
This specifies the position of the right or bottom edge of the window relative to the right or bottom edge of the screen. The integer *position* may be positive or negative; a negative value specifies a position outside the screen.

Here is an example:

```
(x-parse-geometry "35x70+0-0")
     ⇒ ((height . 70) (width . 35)
         (top - 0) (left . 0))
```

28.4 Frame Titles

Every frame has a `name` parameter; this serves as the default for the frame title which window systems typically display at the top of the frame. You can specify a name explicitly by setting the `name` frame property.

Normally you don't specify the name explicitly, and Emacs computes the frame name automatically based on a template stored in the variable `frame-title-format`. Emacs recomputes the name each time the frame is redisplayed.

frame-title-format Variable
This variable specifies how to compute a name for a frame when you have not explicitly specified one. The variable's value is actually a mode line construct, just like `mode-line-format`. See Section 22.3.1 [Mode Line Data], page 406.

icon-title-format Variable
This variable specifies how to compute the name for an iconified frame, when you have not explicitly specified the frame title. This title appears in the icon itself.

multiple-frames Variable
This variable is set automatically by Emacs. Its value is `t` when there are two or more frames (not counting minibuffer-only frames or invisible frames). The default value of `frame-title-format` uses `multiple-frames` so as to put the buffer name in the frame title only when there is more than one frame.

28.5 Deleting Frames

Frames remain potentially visible until you explicitly *delete* them. A deleted frame cannot appear on the screen, but continues to exist as a Lisp object until there are no references to it. There is no way to cancel the deletion of a frame aside from restoring a saved frame configuration (see Section 28.12 [Frame Configurations], page 541); this is similar to the way windows behave.

delete-frame &optional *frame* Command
This function deletes the frame *frame*. By default, *frame* is the selected frame.

frame-live-p *frame* Function
The function `frame-live-p` returns non-`nil` if the frame *frame* has not been deleted.

Some window managers provide a command to delete a window. These work by sending a special message to the program that operates the window. When Emacs gets one of these commands, it generates a `delete-frame` event, whose normal definition is a command that calls the function `delete-frame`. See Section 20.5.10 [Misc Events], page 337.

28.6 Finding All Frames

frame-list Function
The function `frame-list` returns a list of all the frames that have not been deleted. It is analogous to `buffer-list` for buffers. The list that you get is newly created, so modifying the list doesn't have any effect on the internals of Emacs.

visible-frame-list Function
This function returns a list of just the currently visible frames. See Section 28.10 [Visibility of Frames], page 539. (Terminal frames always count as "visible", even though only the selected one is actually displayed.)

next-frame &optional *frame minibuf* Function
The function `next-frame` lets you cycle conveniently through all the frames from an arbitrary starting point. It returns the "next" frame after *frame* in the cycle. If *frame* is omitted or `nil`, it defaults to the selected frame.

The second argument, *minibuf*, says which frames to consider:

`nil` Exclude minibuffer-only frames.

`visible` Consider all visible frames.

0 Consider all visible or iconified frames.

a window Consider only the frames using that particular window as their minibuffer.

anything else
Consider all frames.

previous-frame &optional *frame minibuf* Function
Like `next-frame`, but cycles through all frames in the opposite direction.

See also `next-window` and `previous-window`, in Section 27.5 [Cyclic Window Ordering], page 501.

28.7 Frames and Windows

Each window is part of one and only one frame; you can get the frame with `window-frame`.

window-frame *window* Function
This function returns the frame that *window* is on.

All the non-minibuffer windows in a frame are arranged in a cyclic order. The order runs from the frame's top window, which is at the upper left corner, down and to the right, until it reaches the window at the lower right corner (always the minibuffer window, if the frame has one), and then it moves back to the top. See Section 27.5 [Cyclic Window Ordering], page 501.

frame-top-window *frame* Function
This returns the topmost, leftmost window of frame *frame*.

At any time, exactly one window on any frame is *selected within the frame*. The significance of this designation is that selecting the frame also selects this window. You can get the frame's current selected window with `frame-selected-window`.

frame-selected-window *frame* Function
This function returns the window on *frame* that is selected within *frame*.

Conversely, selecting a window for Emacs with `select-window` also makes that window selected within its frame. See Section 27.4 [Selecting Windows], page 500.

Another function that (usually) returns one of the windows in a given frame is `minibuffer-window`. See Section 19.9 [Minibuffer Misc], page 316.

28.8 Minibuffers and Frames

Normally, each frame has its own minibuffer window at the bottom, which is used whenever that frame is selected. If the frame has a minibuffer, you can get it with `minibuffer-window` (see Section 19.9 [Minibuffer Misc], page 316).

However, you can also create a frame with no minibuffer. Such a frame must use the minibuffer window of some other frame. When you create the frame, you can specify explicitly the minibuffer window to use (in some other frame). If you don't, then the minibuffer is found in the frame which is the value of the variable `default-minibuffer-frame`. Its value should be a frame that does have a minibuffer.

If you use a minibuffer-only frame, you might want that frame to raise when you enter the minibuffer. If so, set the variable `minibuffer-auto-raise` to `t`. See Section 28.11 [Raising and Lowering], page 540.

default-minibuffer-frame Variable
This variable specifies the frame to use for the minibuffer window, by default. It is always local to the current terminal and cannot be buffer-local. See Section 28.2 [Multiple Displays], page 526.

28.9 Input Focus

At any time, one frame in Emacs is the *selected frame*. The selected window always resides on the selected frame.

selected-frame Function
This function returns the selected frame.

Some window systems and window managers direct keyboard input to the window object that the mouse is in; others require explicit clicks or commands to *shift the focus* to various window objects. Either way, Emacs automatically keeps track of which frame has the focus.

Lisp programs can also switch frames "temporarily" by calling the function `select-frame`. This does not alter the window system's concept of focus; rather, it escapes from the window manager's control until that control is somehow reasserted.

When using a text-only terminal, only the selected terminal frame is actually displayed on the terminal. `switch-frame` is the only way to switch frames, and the change lasts until overridden by a subsequent call to `switch-frame`. Each terminal screen except for the initial one has a number, and the number of the selected frame appears in the mode line before the buffer name (see Section 22.3.2 [Mode Line Variables], page 408).

select-frame *frame* Function
This function selects frame *frame*, temporarily disregarding the focus of the X server if any. The selection of *frame* lasts until the next time the user does something to select a different frame, or until the next time this function is called.

Emacs cooperates with the window system by arranging to select frames as the server and window manager request. It does so by generating a special kind of input event, called a *focus* event, when appropriate. The command loop handles a focus event by calling `handle-switch-frame`. See Section 20.5.9 [Focus Events], page 337.

handle-switch-frame *frame* Command
This function handles a focus event by selecting frame *frame*.

Focus events normally do their job by invoking this command. Don't call it for any other reason.

redirect-frame-focus *frame focus-frame* Function
This function redirects focus from *frame* to *focus-frame*. This means that *focus-frame* will receive subsequent keystrokes and events intended for *frame*. After such an event, the value of `last-event-frame` will be *focus-frame*. Also, switch-frame events specifying *frame* will instead select *focus-frame*.

If *focus-frame* is `nil`, that cancels any existing redirection for *frame*, which therefore once again receives its own events.

One use of focus redirection is for frames that don't have minibuffers. These frames use minibuffers on other frames. Activating a minibuffer on another frame redirects focus to that frame. This puts the focus on the minibuffer's frame, where it belongs, even though the mouse remains in the frame that activated the minibuffer.

Selecting a frame can also change focus redirections. Selecting frame `bar`, when `foo` had been selected, changes any redirections pointing to `foo` so that they point to `bar` instead. This allows focus redirection to work properly when the user switches from one frame to another using `select-window`.

This means that a frame whose focus is redirected to itself is treated differently from a frame whose focus is not redirected. `select-frame` affects the former but not the latter.

The redirection lasts until `redirect-frame-focus` is called to change it.

focus-follows-mouse User Option

This option is how you inform Emacs whether the window manager transfers focus when the user moves the mouse. Non-`nil` says that it does. When this is so, the command `other-frame` moves the mouse to a position consistent with the new selected frame.

28.10 Visibility of Frames

A window frame may be *visible*, *invisible*, or *iconified*. If it is visible, you can see its contents. If it is iconified, the frame's contents do not appear on the screen, but an icon does. If the frame is invisible, it doesn't show on the screen, not even as an icon.

Visibility is meaningless for terminal frames, since only the selected one is actually displayed in any case.

make-frame-visible &optional *frame* Command

This function makes frame *frame* visible. If you omit *frame*, it makes the selected frame visible.

make-frame-invisible &optional *frame* Command

This function makes frame *frame* invisible. If you omit *frame*, it makes the selected frame invisible.

iconify-frame &optional *frame* Command

This function iconifies frame *frame*. If you omit *frame*, it iconifies the selected frame.

frame-visible-p *frame* Function
This returns the visibility status of frame *frame*. The value is `t` if *frame* is visible, `nil` if it is invisible, and `icon` if it is iconified.

The visibility status of a frame is also available as a frame parameter. You can read or change it as such. See Section 28.3.3 [Window Frame Parameters], page 528.

The user can iconify and deiconify frames with the window manager. This happens below the level at which Emacs can exert any control, but Emacs does provide events that you can use to keep track of such changes. See Section 20.5.10 [Misc Events], page 337.

28.11 Raising and Lowering Frames

Most window systems use a desktop metaphor. Part of this metaphor is the idea that windows are stacked in a notional third dimension perpendicular to the screen surface, and thus ordered from "highest" to "lowest". Where two windows overlap, the one higher up covers the one underneath. Even a window at the bottom of the stack can be seen if no other window overlaps it.

A window's place in this ordering is not fixed; in fact, users tend to change the order frequently. *Raising* a window means moving it "up", to the top of the stack. *Lowering* a window means moving it to the bottom of the stack. This motion is in the notional third dimension only, and does not change the position of the window on the screen.

You can raise and lower Emacs frame Windows with these functions:

raise-frame &optional *frame* Command
This function raises frame *frame* (default, the selected frame).

lower-frame &optional *frame* Command
This function lowers frame *frame* (default, the selected frame).

minibuffer-auto-raise User Option
If this is non-`nil`, activation of the minibuffer raises the frame that the minibuffer window is in.

You can also enable auto-raise (raising automatically when a frame is selected) or auto-lower (lowering automatically when it is deselected) for any frame using frame parameters. See Section 28.3.3 [Window Frame Parameters], page 528.

28.12 Frame Configurations

A *frame configuration* records the current arrangement of frames, all their properties, and the window configuration of each one. (See Section 27.16 [Window Configurations], page 521.)

current-frame-configuration Function
This function returns a frame configuration list that describes the current arrangement of frames and their contents.

set-frame-configuration *configuration* Function
This function restores the state of frames described in *configuration*.

28.13 Mouse Tracking

Sometimes it is useful to *track* the mouse, which means to display something to indicate where the mouse is and move the indicator as the mouse moves. For efficient mouse tracking, you need a way to wait until the mouse actually moves.

The convenient way to track the mouse is to ask for events to represent mouse motion. Then you can wait for motion by waiting for an event. In addition, you can easily handle any other sorts of events that may occur. That is useful, because normally you don't want to track the mouse forever—only until some other event, such as the release of a button.

track-mouse *body...* Special Form
This special form executes *body*, with generation of mouse motion events enabled. Typically *body* would use `read-event` to read the motion events and modify the display accordingly. See Section 20.5.8 [Motion Events], page 337, for the format of mouse motion events.

The value of `track-mouse` is that of the last form in *body*. You should design *body* to return when it sees the up-event that indicates the release of the button, or whatever kind of event means it is time to stop tracking.

The usual purpose of tracking mouse motion is to indicate on the screen the consequences of pushing or releasing a button at the current position.

In many cases, you can avoid the need to track the mouse by using the `mouse-face` text property (see Section 31.19.4 [Special Properties], page 615). That works at a much lower level and runs more smoothly than Lisp-level mouse tracking.

28.14 Mouse Position

The functions `mouse-position` and `set-mouse-position` give access to the current position of the mouse.

mouse-position Function
This function returns a description of the position of the mouse. The value looks like (*frame x . y*), where *x* and *y* are integers giving the position in characters relative to the top left corner of the inside of *frame*.

set-mouse-position *frame x y* Function
This function *warps the mouse* to position *x*, *y* in frame *frame*. The arguments *x* and *y* are integers, giving the position in characters relative to the top left corner of the inside of *frame*. If *frame* is not visible, this function does nothing. The return value is not significant.

mouse-pixel-position Function
This function is like `mouse-position` except that it returns coordinates in units of pixels rather than units of characters.

set-mouse-pixel-position *frame x y* Function
This function warps the mouse like `set-mouse-position` except that *x* and *y* are in units of pixels rather than units of characters. These coordinates are not required to be within the frame.

If *frame* is not visible, this function does nothing. The return value is not significant.

28.15 Pop-Up Menus

When using a window system, a Lisp program can pop up a menu so that the user can choose an alternative with the mouse.

x-popup-menu *position menu* Function
This function displays a pop-up menu and returns an indication of what selection the user makes.

The argument *position* specifies where on the screen to put the menu. It can be either a mouse button event (which says to put the menu where the user actuated the button) or a list of this form:

```
((xoffset yoffset) window)
```

where *xoffset* and *yoffset* are coordinates, measured in pixels, counting from the top left corner of *window*'s frame.

If *position* is `t`, it means to use the current mouse position. If *position* is `nil`, it means to precompute the key binding equivalents for the keymaps specified in *menu*, without actually displaying or popping up the menu.

The argument *menu* says what to display in the menu. It can be a keymap or a list of keymaps (see Section 21.12 [Menu Keymaps], page 381). Alternatively, it can have the following form:

```
(title pane1 pane2...)
```

where each pane is a list of form

(*title* (*line* . *item*)...)

Each *line* should be a string, and each *item* should be the value to return if that *line* is chosen.

Usage note: Don't use `x-popup-menu` to display a menu if you could do the job with a prefix key defined with a menu keymap. If you use a menu keymap to implement a menu, `C-h c` and `C-h a` can see the individual items in that menu and provide help for them. If instead you implement the menu by defining a command that calls `x-popup-menu`, the help facilities cannot know what happens inside that command, so they cannot give any help for the menu's items.

The menu bar mechanism, which lets you switch between submenus by moving the mouse, cannot look within the definition of a command to see that it calls `x-popup-menu`. Therefore, if you try to implement a submenu using `x-popup-menu`, it cannot work with the menu bar in an integrated fashion. This is why all menu bar submenus are implemented with menu keymaps within the parent menu, and never with `x-popup-menu`. See Section 21.12.5 [Menu Bar], page 387,

If you want a menu bar submenu to have contents that vary, you should still use a menu keymap to implement it. To make the contents vary, add a hook function to `menu-bar-update-hook` to update the contents of the menu keymap as necessary.

28.16 Dialog Boxes

A dialog box is a variant of a pop-up menu—it looks a little different, it always appears in the center of a frame, and it has just one level and one pane. The main use of dialog boxes is for asking questions that the user can answer with "yes", "no", and a few other alternatives. The functions `y-or-n-p` and `yes-or-no-p` use dialog boxes instead of the keyboard, when called from commands invoked by mouse clicks.

x-popup-dialog *position contents* Function

This function displays a pop-up dialog box and returns an indication of what selection the user makes. The argument *contents* specifies the alternatives to offer; it has this format:

(*title* (*string* . *value*)...)

which looks like the list that specifies a single pane for `x-popup-menu`.

The return value is *value* from the chosen alternative.

An element of the list may be just a string instead of a cons cell (*string* . *value*). That makes a box that cannot be selected.

If `nil` appears in the list, it separates the left-hand items from the right-hand items; items that precede the `nil` appear on the left, and items that follow the `nil` appear on the right. If you don't include a `nil` in the list, then approximately half the items appear on each side.

Dialog boxes always appear in the center of a frame; the argument *position* specifies which frame. The possible values are as in `x-popup-menu`, but the precise coordinates don't matter; only the frame matters.

In some configurations, Emacs cannot display a real dialog box; so instead it displays the same items in a pop-up menu in the center of the frame.

28.17 Pointer Shapes

These variables specify which shape to use for the mouse pointer in various situations, when using the X Window System:

`x-pointer-shape`
: This variable specifies the pointer shape to use ordinarily in the Emacs frame.

`x-sensitive-text-pointer-shape`
: This variable specifies the pointer shape to use when the mouse is over mouse-sensitive text.

These variables affect newly created frames. They do not normally affect existing frames; however, if you set the mouse color of a frame, that also updates its pointer shapes based on the current values of these variables. See Section 28.3.3 [Window Frame Parameters], page 528.

The values you can use, to specify either of these pointer shapes, are defined in the file '`lisp/term/x-win.el`'. Use *M-x apropos* RET *x-pointer* RET to see a list of them.

28.18 Window System Selections

The X server records a set of *selections* which permit transfer of data between application programs. The various selections are distinguished by *selection types*, represented in Emacs by symbols. X clients including Emacs can read or set the selection for any given type.

x-set-selection *type data* Function

This function sets a "selection" in the X server. It takes two arguments: a selection type *type*, and the value to assign to it, *data*. If *data* is `nil`, it means to clear out the selection. Otherwise, *data* may be a string, a symbol, an integer (or a cons of two integers or list of two integers), an overlay, or a cons of two markers pointing to the same buffer. An overlay or a pair of markers stands for text in the overlay or between the markers.

The argument *data* may also be a vector of valid non-vector selection values.

Each possible *type* has its own selection value, which changes independently. The usual values of *type* are `PRIMARY` and `SECONDARY`; these are symbols with upper-case names, in accord with X Window System conventions. The default is `PRIMARY`.

x-get-selection &optional *type data-type* Function
This function accesses selections set up by Emacs or by other X clients. It takes two optional arguments, *type* and *data-type*. The default for *type*, the selection type, is `PRIMARY`.

The *data-type* argument specifies the form of data conversion to use, to convert the raw data obtained from another X client into Lisp data. Meaningful values include `TEXT`, `STRING`, `TARGETS`, `LENGTH`, `DELETE`, `FILE_NAME`, `CHARACTER_POSITION`, `LINE_NUMBER`, `COLUMN_NUMBER`, `OWNER_OS`, `HOST_NAME`, `USER`, `CLASS`, `NAME`, `ATOM`, and `INTEGER`. (These are symbols with upper-case names in accord with X conventions.) The default for *data-type* is `STRING`.

The X server also has a set of numbered *cut buffers* which can store text or other data being moved between applications. Cut buffers are considered obsolete, but Emacs supports them for the sake of X clients that still use them.

x-get-cut-buffer *n* Function
This function returns the contents of cut buffer number *n*.

x-set-cut-buffer *string* Function
This function stores *string* into the first cut buffer (cut buffer 0), moving the other values down through the series of cut buffers, much like the way successive kills in Emacs move down the kill ring.

selection-coding-system Variable
This variable specifies the coding system to use when reading and writing a selections, the clipboard, or a cut buffer. See Section 32.10 [Coding Systems], page 636. The default is `compound-text`.

28.19 Looking up Font Names

x-list-font *pattern* &optional *face frame maximum* Function
This function returns a list of available font names that match *pattern*. If the optional arguments *face* and *frame* are specified, then the list is limited to fonts that are the same size as *face* currently is on *frame*.

The argument *pattern* should be a string, perhaps with wildcard characters: the '`*`' character matches any substring, and the '`?`' character matches any single character. Pattern matching of font names ignores case.

If you specify *face* and *frame*, *face* should be a face name (a symbol) and *frame* should be a frame.

The optional argument *maximum* sets a limit on how many fonts to return. If this is non-`nil`, then the return value is truncated after the first

maximum matching fonts. Specifying a small value for *maximum* can make this function much faster, in cases where many fonts match the pattern.

28.20 Fontsets

A *fontset* is a list of fonts, each assigned to a range of character codes. An individual font cannot display the whole range of characters that Emacs supports, but a fontset can. Fontsets have names, just as fonts do, and you can use a fontset name in place of a font name when you specify the "font" for a frame or a face. Here is information about defining a fontset under Lisp program control.

create-fontset-from-fontset-spec *fontset-spec* Function
&optional *style-variant-p noerror*

This function defines a new fontset according to the specification string *fontset-spec*. The string should have this format:

fontpattern, [*charsetname*:*fontname*]...

Whitespace characters before and after the commas are ignored.

The first part of the string, *fontpattern*, should have the form of a standard X font name, except that the last two fields should be '`fontset-`*alias*'.

The new fontset has two names, one long and one short. The long name is *fontpattern* in its entirety. The short name is '`fontset-`*alias*'. You can refer to the fontset by either name. If a fontset with the same name already exists, an error is signaled, unless *noerror* is non-`nil`, in which case this function does nothing.

If optional argument *style-variant-p* is non-`nil`, that says to create bold, italic and bold-italic variants of the fontset as well. These variant fontsets do not have a short name, only a long one, which is made by altering *fontpattern* to indicate the bold or italic status.

The specification string also says which fonts to use in the fontset. See below for the details.

The construct '*charset*:*font*' specifies which font to use (in this fontset) for one particular character set. Here, *charset* is the name of a character set, and *font* is the font to use for that character set. You can use this construct any number of times in the specification string.

For the remaining character sets, those that you don't specify explicitly, Emacs chooses a font based on *fontpattern*: it replaces '`fontset-`*alias*' with a value that names one character set. For the ASCII character set, '`fontset-`*alias*' is replaced with '`ISO8859-1`'.

In addition, when several consecutive fields are wildcards, Emacs collapses them into a single wildcard. This is to prevent use of auto-scaled

fonts. Fonts made by scaling larger fonts are not usable for editing, and scaling a smaller font is not useful because it is better to use the smaller font in its own size, which Emacs does.

Thus if *fontpattern* is this,

```
-*-fixed-medium-r-normal-*-24-*-*-*-*-*-fontset-24
```

the font specification for ASCII characters would be this:

```
-*-fixed-medium-r-normal-*-24-*-ISO8859-1
```

and the font specification for Chinese GB2312 characters would be this:

```
-*-fixed-medium-r-normal-*-24-*-gb2312*-*
```

You may not have any Chinese font matching the above font specification. Most X distributions include only Chinese fonts that have '`song ti`' or '`fangsong ti`' in the *family* field. In such a case, '`Fontset-`*n*' can be specified as below:

```
Emacs.Fontset-0: -*-fixed-medium-r-normal-*-24-*-*-*-*-*-fontset-24,\
       chinese-gb2312:-*-*-medium-r-normal-*-24-*-gb2312*-*
```

Then, the font specifications for all but Chinese GB2312 characters have '`fixed`' in the *family* field, and the font specification for Chinese GB2312 characters has a wild card '`*`' in the *family* field.

28.21 Color Names

x-color-defined-p *color* &optional *frame* Function

This function reports whether a color name is meaningful. It returns `t` if so; otherwise, `nil`. The argument *frame* says which frame's display to ask about; if *frame* is omitted or `nil`, the selected frame is used.

Note that this does not tell you whether the display you are using really supports that color. You can ask for any defined color on any kind of display, and you will get some result—that is how the X server works. Here's an approximate way to test whether your display supports the color *color*:

```
(defun x-color-supported-p (color &optional frame)
  (and (x-color-defined-p color frame)
       (or (x-display-color-p frame)
           (member color '("black" "white"))
           (and (> (x-display-planes frame) 1)
                (equal color "gray")))))
```

x-color-values *color* &optional *frame* Function

This function returns a value that describes what *color* should ideally look like. If *color* is defined, the value is a list of three integers, which give the amount of red, the amount of green, and the amount of blue. Each integer ranges in principle from 0 to 65535, but in practice no value seems to be above 65280. If *color* is not defined, the value is `nil`.

```
(x-color-values "black")
     ⇒ (0 0 0)
(x-color-values "white")
     ⇒ (65280 65280 65280)
(x-color-values "red")
     ⇒ (65280 0 0)
(x-color-values "pink")
     ⇒ (65280 49152 51968)
(x-color-values "hungry")
     ⇒ nil
```

The color values are returned for *frame*'s display. If *frame* is omitted or `nil`, the information is returned for the selected frame's display.

28.22 X Resources

x-get-resource *attribute class* &optional *component subclass* Function

The function `x-get-resource` retrieves a resource value from the X Windows defaults database.

Resources are indexed by a combination of a *key* and a *class*. This function searches using a key of the form '*instance*`.`*attribute*' (where *instance* is the name under which Emacs was invoked), and using '`Emacs.`*class*' as the class.

The optional arguments *component* and *subclass* add to the key and the class, respectively. You must specify both of them or neither. If you specify them, the key is '*instance*`.`*component*`.`*attribute*', and the class is '`Emacs.`*class*`.`*subclass*'.

x-resource-class Variable

This variable specifies the application name that `x-get-resource` should look up. The default value is `"Emacs"`. You can examine X resources for application names other than "Emacs" by binding this variable to some other string, around a call to `x-get-resource`.

See section "X Resources" in *The GNU Emacs Manual*.

28.23 Data about the X Server

This section describes functions you can use to get information about the capabilities and origin of an X display that Emacs is using. Each of these functions lets you specify the display you are interested in: the *display* argument can be either a display name, or a frame (meaning use the display that frame is on). If you omit the *display* argument, or specify `nil`, that means to use the selected frame's display.

x-display-screens &optional *display* Function
This function returns the number of screens associated with the display.

x-server-version &optional *display* Function
This function returns the list of version numbers of the X server running the display.

x-server-vendor &optional *display* Function
This function returns the vendor that provided the X server software.

x-display-pixel-height &optional *display* Function
This function returns the height of the screen in pixels.

x-display-mm-height &optional *display* Function
This function returns the height of the screen in millimeters.

x-display-pixel-width &optional *display* Function
This function returns the width of the screen in pixels.

x-display-mm-width &optional *display* Function
This function returns the width of the screen in millimeters.

x-display-backing-store &optional *display* Function
This function returns the backing store capability of the screen. Values can be the symbols `always`, `when-mapped`, or `not-useful`.

x-display-save-under &optional *display* Function
This function returns non-`nil` if the display supports the SaveUnder feature.

x-display-planes &optional *display* Function
This function returns the number of planes the display supports.

x-display-visual-class &optional *display* Function
This function returns the visual class for the screen. The value is one of the symbols `static-gray`, `gray-scale`, `static-color`, `pseudo-color`, `true-color`, and `direct-color`.

x-display-grayscale-p &optional *display* Function
This function returns `t` if the screen can display shades of gray.

x-display-color-p &optional *display* Function
This function returns `t` if the screen is a color screen.

x-display-color-cells &optional *display* Function
This function returns the number of color cells the screen supports.

29 Positions

A *position* is the index of a character in the text of a buffer. More precisely, a position identifies the place between two characters (or before the first character, or after the last character), so we can speak of the character before or after a given position. However, we often speak of the character "at" a position, meaning the character after that position.

Positions are usually represented as integers starting from 1, but can also be represented as *markers*—special objects that relocate automatically when text is inserted or deleted so they stay with the surrounding characters. See Chapter 30 [Markers], page 565.

29.1 Point

Point is a special buffer position used by many editing commands, including the self-inserting typed characters and text insertion functions. Other commands move point through the text to allow editing and insertion at different places.

Like other positions, point designates a place between two characters (or before the first character, or after the last character), rather than a particular character. Usually terminals display the cursor over the character that immediately follows point; point is actually before the character on which the cursor sits.

The value of point is a number between 1 and the buffer size plus 1. If narrowing is in effect (see Section 29.4 [Narrowing], page 561), then point is constrained to fall within the accessible portion of the buffer (possibly at one end of it).

Each buffer has its own value of point, which is independent of the value of point in other buffers. Each window also has a value of point, which is independent of the value of point in other windows on the same buffer. This is why point can have different values in various windows that display the same buffer. When a buffer appears in only one window, the buffer's point and the window's point normally have the same value, so the distinction is rarely important. See Section 27.9 [Window Point], page 509, for more details.

point Function

This function returns the value of point in the current buffer, as an integer.

```
(point)
     ⇒ 175
```

point-min Function

This function returns the minimum accessible value of point in the current buffer. This is normally 1, but if narrowing is in effect, it is the

position of the start of the region that you narrowed to. (See Section 29.4 [Narrowing], page 561.)

point-max Function
This function returns the maximum accessible value of point in the current buffer. This is `(1+ (buffer-size))`, unless narrowing is in effect, in which case it is the position of the end of the region that you narrowed to. (See Section 29.4 [Narrowing], page 561).

buffer-end *flag* Function
This function returns `(point-min)` if *flag* is less than 1, `(point-max)` otherwise. The argument *flag* must be a number.

buffer-size Function
This function returns the total number of characters in the current buffer. In the absence of any narrowing (see Section 29.4 [Narrowing], page 561), `point-max` returns a value one larger than this.

```
(buffer-size)
     ⇒ 35
(point-max)
     ⇒ 36
```

29.2 Motion

Motion functions change the value of point, either relative to the current value of point, relative to the beginning or end of the buffer, or relative to the edges of the selected window. See Section 29.1 [Point], page 551.

29.2.1 Motion by Characters

These functions move point based on a count of characters. `goto-char` is the fundamental primitive; the other functions use that.

goto-char *position* Command
This function sets point in the current buffer to the value *position*. If *position* is less than 1, it moves point to the beginning of the buffer. If *position* is greater than the length of the buffer, it moves point to the end.

If narrowing is in effect, *position* still counts from the beginning of the buffer, but point cannot go outside the accessible portion. If *position* is out of range, `goto-char` moves point to the beginning or the end of the accessible portion.

When this function is called interactively, *position* is the numeric prefix argument, if provided; otherwise it is read from the minibuffer.

`goto-char` returns *position*.

forward-char &optional *count* Command
This function moves point *count* characters forward, towards the end of the buffer (or backward, towards the beginning of the buffer, if *count* is negative). If the function attempts to move point past the beginning or end of the buffer (or the limits of the accessible portion, when narrowing is in effect), an error is signaled with error code `beginning-of-buffer` or `end-of-buffer`.

In an interactive call, *count* is the numeric prefix argument.

backward-char &optional *count* Command
This function moves point *count* characters backward, towards the beginning of the buffer (or forward, towards the end of the buffer, if *count* is negative). If the function attempts to move point past the beginning or end of the buffer (or the limits of the accessible portion, when narrowing is in effect), an error is signaled with error code `beginning-of-buffer` or `end-of-buffer`.

In an interactive call, *count* is the numeric prefix argument.

29.2.2 Motion by Words

These functions for parsing words use the syntax table to decide whether a given character is part of a word. See Chapter 34 [Syntax Tables], page 669.

forward-word *count* Command
This function moves point forward *count* words (or backward if *count* is negative). "Moving one word" means moving until point crosses a word-constituent character and then encounters a word-separator character (or the boundary of the accessible part of the buffer).

If it is possible to move *count* words, without being stopped by the buffer boundary (except perhaps after the last word), the value is `t`. Otherwise, the return value is `nil` and point stops at the buffer boundary.

In an interactive call, *count* is set to the numeric prefix argument.

backward-word *count* Command
This function is just like `forward-word`, except that it moves backward until encountering the front of a word, rather than forward.

In an interactive call, *count* is set to the numeric prefix argument.

This function is rarely used in programs, as it is more efficient to call `forward-word` with a negative argument.

words-include-escapes Variable
This variable affects the behavior of `forward-word` and everything that uses it. If it is non-`nil`, then characters in the "escape" and "character quote" syntax classes count as part of words. Otherwise, they do not.

29.2.3 Motion to an End of the Buffer

To move point to the beginning of the buffer, write:

```
(goto-char (point-min))
```

Likewise, to move to the end of the buffer, use:

```
(goto-char (point-max))
```

Here are two commands that users use to do these things. They are documented here to warn you not to use them in Lisp programs, because they set the mark and display messages in the echo area.

beginning-of-buffer &optional *n* Command

This function moves point to the beginning of the buffer (or the limits of the accessible portion, when narrowing is in effect), setting the mark at the previous position. If *n* is non-`nil`, then it puts point *n* tenths of the way from the beginning of the buffer.

In an interactive call, *n* is the numeric prefix argument, if provided; otherwise *n* defaults to `nil`.

Warning: Don't use this function in Lisp programs!

end-of-buffer &optional *n* Command

This function moves point to the end of the buffer (or the limits of the accessible portion, when narrowing is in effect), setting the mark at the previous position. If *n* is non-`nil`, then it puts point *n* tenths of the way from the end of the buffer.

In an interactive call, *n* is the numeric prefix argument, if provided; otherwise *n* defaults to `nil`.

Warning: Don't use this function in Lisp programs!

29.2.4 Motion by Text Lines

Text lines are portions of the buffer delimited by newline characters, which are regarded as part of the previous line. The first text line begins at the beginning of the buffer, and the last text line ends at the end of the buffer whether or not the last character is a newline. The division of the buffer into text lines is not affected by the width of the window, by line continuation in display, or by how tabs and control characters are displayed.

goto-line *line* Command

This function moves point to the front of the *line*th line, counting from line 1 at beginning of the buffer. If *line* is less than 1, it moves point to the beginning of the buffer. If *line* is greater than the number of lines in the buffer, it moves point to the end of the buffer—that is, the *end of the last line* of the buffer. This is the only case in which `goto-line` does not necessarily move to the beginning of a line.

If narrowing is in effect, then *line* still counts from the beginning of the buffer, but point cannot go outside the accessible portion. So `goto-line` moves point to the beginning or end of the accessible portion, if the line number specifies an inaccessible position.

The return value of `goto-line` is the difference between *line* and the line number of the line to which point actually was able to move (in the full buffer, before taking account of narrowing). Thus, the value is positive if the scan encounters the real end of the buffer before finding the specified line. The value is zero if scan encounters the end of the accessible portion but not the real end of the buffer.

In an interactive call, *line* is the numeric prefix argument if one has been provided. Otherwise *line* is read in the minibuffer.

beginning-of-line &optional *count* Command

This function moves point to the beginning of the current line. With an argument *count* not `nil` or 1, it moves forward *count*−1 lines and then to the beginning of the line.

If this function reaches the end of the buffer (or of the accessible portion, if narrowing is in effect), it positions point there. No error is signaled.

end-of-line &optional *count* Command

This function moves point to the end of the current line. With an argument *count* not `nil` or 1, it moves forward *count*−1 lines and then to the end of the line.

If this function reaches the end of the buffer (or of the accessible portion, if narrowing is in effect), it positions point there. No error is signaled.

forward-line &optional *count* Command

This function moves point forward *count* lines, to the beginning of the line. If *count* is negative, it moves point −*count* lines backward, to the beginning of a line. If *count* is zero, it moves point to the beginning of the current line.

If `forward-line` encounters the beginning or end of the buffer (or of the accessible portion) before finding that many lines, it sets point there. No error is signaled.

`forward-line` returns the difference between *count* and the number of lines actually moved. If you attempt to move down five lines from the beginning of a buffer that has only three lines, point stops at the end of the last line, and the value will be 2.

In an interactive call, *count* is the numeric prefix argument.

count-lines *start end* Function

This function returns the number of lines between the positions *start* and *end* in the current buffer. If *start* and *end* are equal, then it returns

0. Otherwise it returns at least 1, even if *start* and *end* are on the same line. This is because the text between them, considered in isolation, must contain at least one line unless it is empty.

Here is an example of using `count-lines`:

```
(defun current-line ()
  "Return the vertical position of point..."
  (+ (count-lines (window-start) (point))
     (if (= (current-column) 0) 1 0)
     -1))
```

Also see the functions `bolp` and `eolp` in Section 31.1 [Near Point], page 575. These functions do not move point, but test whether it is already at the beginning or end of a line.

29.2.5 Motion by Screen Lines

The line functions in the previous section count text lines, delimited only by newline characters. By contrast, these functions count screen lines, which are defined by the way the text appears on the screen. A text line is a single screen line if it is short enough to fit the width of the selected window, but otherwise it may occupy several screen lines.

In some cases, text lines are truncated on the screen rather than continued onto additional screen lines. In these cases, `vertical-motion` moves point much like `forward-line`. See Section 38.2 [Truncation], page 739.

Because the width of a given string depends on the flags that control the appearance of certain characters, `vertical-motion` behaves differently, for a given piece of text, depending on the buffer it is in, and even on the selected window (because the width, the truncation flag, and display table may vary between windows). See Section 38.13 [Usual Display], page 760.

These functions scan text to determine where screen lines break, and thus take time proportional to the distance scanned. If you intend to use them heavily, Emacs provides caches which may improve the performance of your code. See Section 38.2 [Truncation], page 739.

vertical-motion *count* &optional *window* Function

This function moves point to the start of the screen line *count* screen lines down from the screen line containing point. If *count* is negative, it moves up instead.

`vertical-motion` returns the number of screen lines over which it moved point. The value may be less in absolute value than *count* if the beginning or end of the buffer was reached.

The window *window* is used for obtaining parameters such as the width, the horizontal scrolling, and the display table. But `vertical-motion` always operates on the current buffer, even if *window* currently displays some other buffer.

move-to-window-line *count* Command

This function moves point with respect to the text currently displayed in the selected window. It moves point to the beginning of the screen line *count* screen lines from the top of the window. If *count* is negative, that specifies a position $-$*count* lines from the bottom (or the last line of the buffer, if the buffer ends above the specified screen position).

If *count* is `nil`, then point moves to the beginning of the line in the middle of the window. If the absolute value of *count* is greater than the size of the window, then point moves to the place that would appear on that screen line if the window were tall enough. This will probably cause the next redisplay to scroll to bring that location onto the screen.

In an interactive call, *count* is the numeric prefix argument.

The value returned is the window line number point has moved to, with the top line in the window numbered 0.

compute-motion *from frompos to topos width offsets window* Function

This function scans the current buffer, calculating screen positions. It scans the buffer forward from position *from*, assuming that is at screen coordinates *frompos*, to position *to* or coordinates *topos*, whichever comes first. It returns the ending buffer position and screen coordinates.

The coordinate arguments *frompos* and *topos* are cons cells of the form (*hpos* . *vpos*).

The argument *width* is the number of columns available to display text; this affects handling of continuation lines. Use the value returned by `window-width` for the window of your choice; normally, use (`window-width` *window*).

The argument *offsets* is either `nil` or a cons cell of the form (*hscroll* . *tab-offset*). Here *hscroll* is the number of columns not being displayed at the left margin; most callers get this by calling `window-hscroll`. Meanwhile, *tab-offset* is the offset between column numbers on the screen and column numbers in the buffer. This can be nonzero in a continuation line, when the previous screen lines' widths do not add up to a multiple of `tab-width`. It is always zero in a non-continuation line.

The window *window* serves only to specify which display table to use. `compute-motion` always operates on the current buffer, regardless of what buffer is displayed in *window*.

The return value is a list of five elements:

(*pos vpos hpos prevhpos contin*)

Here *pos* is the buffer position where the scan stopped, *vpos* is the vertical screen position, and *hpos* is the horizontal screen position.

The result *prevhpos* is the horizontal position one character back from *pos*. The result *contin* is `t` if the last line was continued after (or within) the previous character.

For example, to find the buffer position of column *col* of screen line *line* of a certain window, pass the window's display start location as *from* and the window's upper-left coordinates as *frompos*. Pass the buffer's `(point-max)` as *to*, to limit the scan to the end of the accessible portion of the buffer, and pass *line* and *col* as *topos*. Here's a function that does this:

```
(defun coordinates-of-position (col line)
  (car (compute-motion (window-start)
                       '(0 . 0)
                       (point-max)
                       (cons col line)
                       (window-width)
                       (cons (window-hscroll) 0)
                       (selected-window))))
```

When you use `compute-motion` for the minibuffer, you need to use `minibuffer-prompt-width` to get the horizontal position of the beginning of the first screen line. See Section 19.9 [Minibuffer Misc], page 316.

29.2.6 Moving over Balanced Expressions

Here are several functions concerned with balanced-parenthesis expressions (also called *sexps* in connection with moving across them in Emacs). The syntax table controls how these functions interpret various characters; see Chapter 34 [Syntax Tables], page 669. See Section 34.6 [Parsing Expressions], page 677, for lower-level primitives for scanning sexps or parts of sexps. For user-level commands, see section "Lists Commands" in *GNU Emacs Manual*.

forward-list *arg* Command

This function moves forward across *arg* balanced groups of parentheses. (Other syntactic entities such as words or paired string quotes are ignored.)

backward-list *arg* Command

This function moves backward across *arg* balanced groups of parentheses. (Other syntactic entities such as words or paired string quotes are ignored.)

up-list *arg* Command

This function moves forward out of *arg* levels of parentheses. A negative argument means move backward but still to a less deep spot.

down-list *arg* Command

This function moves forward into *arg* levels of parentheses. A negative argument means move backward but still go deeper in parentheses ($-$*arg* levels).

forward-sexp *arg* Command

This function moves forward across *arg* balanced expressions. Balanced expressions include both those delimited by parentheses and other kinds, such as words and string constants. For example,

```
---------- Buffer: foo ----------
(concat* "foo " (car x) y z)
---------- Buffer: foo ----------

(forward-sexp 3)
     ⇒ nil

---------- Buffer: foo ----------
(concat "foo " (car x) y* z)
---------- Buffer: foo ----------
```

backward-sexp *arg* Command

This function moves backward across *arg* balanced expressions.

beginning-of-defun *arg* Command

This function moves back to the *arg*th beginning of a defun. If *arg* is negative, this actually moves forward, but it still moves to the beginning of a defun, not to the end of one.

end-of-defun *arg* Command

This function moves forward to the *arg*th end of a defun. If *arg* is negative, this actually moves backward, but it still moves to the end of a defun, not to the beginning of one.

defun-prompt-regexp User Option

If non-`nil`, this variable holds a regular expression that specifies what text can appear before the open-parenthesis that starts a defun. That is to say, a defun begins on a line that starts with a match for this regular expression, followed by a character with open-parenthesis syntax.

29.2.7 Skipping Characters

The following two functions move point over a specified set of characters. For example, they are often used to skip whitespace. For related functions, see Section 34.5 [Motion and Syntax], page 677.

skip-chars-forward *character-set* &optional *limit* Function

This function moves point in the current buffer forward, skipping over a given set of characters. It examines the character following point, then advances point if the character matches *character-set*. This continues until it reaches a character that does not match. The function returns the number of characters moved over.

The argument *character-set* is like the inside of a '`[...]`' in a regular expression except that '`]`' is never special and '`\`' quotes '`^`', '`-`' or '`\`'. Thus, `"a-zA-Z"` skips over all letters, stopping before the first nonletter, and `"^a-zA-Z"` skips nonletters stopping before the first letter. See Section 33.2 [Regular Expressions], page 649.

If *limit* is supplied (it must be a number or a marker), it specifies the maximum position in the buffer that point can be skipped to. Point will stop at or before *limit*.

In the following example, point is initially located directly before the '`T`'. After the form is evaluated, point is located at the end of that line (between the '`t`' of '`hat`' and the newline). The function skips all letters and spaces, but not newlines.

```
---------- Buffer: foo ----------
I read "⋆The cat in the hat
comes back" twice.
---------- Buffer: foo ----------

(skip-chars-forward "a-zA-Z ")
     ⇒ nil

---------- Buffer: foo ----------
I read "The cat in the hat⋆
comes back" twice.
---------- Buffer: foo ----------
```

skip-chars-backward *character-set* &optional *limit* Function

This function moves point backward, skipping characters that match *character-set*, until *limit*. It is just like `skip-chars-forward` except for the direction of motion.

The return value indicates the distance traveled. It is an integer that is zero or less.

29.3 Excursions

It is often useful to move point "temporarily" within a localized portion of the program, or to switch buffers temporarily. This is called an *excursion*, and it is done with the `save-excursion` special form. This construct saves the current buffer and its values of point and the mark so they can be restored after the completion of the excursion.

The forms for saving and restoring the configuration of windows are described elsewhere (see Section 27.16 [Window Configurations], page 521, and see Section 28.12 [Frame Configurations], page 541).

save-excursion *forms...* Special Form

The `save-excursion` special form saves the identity of the current buffer and the values of point and the mark in it, evaluates *forms*, and finally restores the buffer and its saved values of point and the mark. All three saved values are restored even in case of an abnormal exit via `throw` or error (see Section 9.5 [Nonlocal Exits], page 135).

The `save-excursion` special form is the standard way to switch buffers or move point within one part of a program and avoid affecting the rest of the program. It is used more than 4000 times in the Lisp sources of Emacs.

`save-excursion` does not save the values of point and the mark for other buffers, so changes in other buffers remain in effect after `save-excursion` exits.

Likewise, `save-excursion` does not restore window-buffer correspondences altered by functions such as `switch-to-buffer`. One way to restore these correspondences, and the selected window, is to use `save-window-excursion` inside `save-excursion` (see Section 27.16 [Window Configurations], page 521).

The value returned by `save-excursion` is the result of the last of *forms*, or `nil` if no *forms* are given.

```
(save-excursion forms)
≡
(let ((old-buf (current-buffer))
      (old-pnt (point-marker))
      (old-mark (copy-marker (mark-marker))))
  (unwind-protect
      (progn forms)
    (set-buffer old-buf)
    (goto-char old-pnt)
    (set-marker (mark-marker) old-mark)))
```

Warning: Ordinary insertion of text adjacent to the saved point value relocates the saved value, just as it relocates all markers. Therefore, when the saved point value is restored, it normally comes before the inserted text.

Although `save-excursion` saves the location of the mark, it does not prevent functions which modify the buffer from setting `deactivate-mark`, and thus causing the deactivation of the mark after the command finishes. See Section 30.7 [The Mark], page 570.

29.4 Narrowing

Narrowing means limiting the text addressable by Emacs editing commands to a limited range of characters in a buffer. The text that remains addressable is called the *accessible portion* of the buffer.

Narrowing is specified with two buffer positions which become the beginning and end of the accessible portion. For most editing commands and most Emacs primitives, these positions replace the values of the beginning and end of the buffer. While narrowing is in effect, no text outside the accessible portion is displayed, and point cannot move outside the accessible portion.

Values such as positions or line numbers, which usually count from the beginning of the buffer, do so despite narrowing, but the functions which use them refuse to operate on text that is inaccessible.

The commands for saving buffers are unaffected by narrowing; they save the entire buffer regardless of any narrowing.

narrow-to-region *start end* Command

This function sets the accessible portion of the current buffer to start at *start* and end at *end*. Both arguments should be character positions.

In an interactive call, *start* and *end* are set to the bounds of the current region (point and the mark, with the smallest first).

narrow-to-page *move-count* Command

This function sets the accessible portion of the current buffer to include just the current page. An optional first argument *move-count* non-`nil` means to move forward or backward by *move-count* pages and then narrow to one page. The variable `page-delimiter` specifies where pages start and end (see Section 33.8 [Standard Regexps], page 666).

In an interactive call, *move-count* is set to the numeric prefix argument.

widen Command

This function cancels any narrowing in the current buffer, so that the entire contents are accessible. This is called *widening*. It is equivalent to the following expression:

```
(narrow-to-region 1 (1+ (buffer-size)))
```

save-restriction *body...* Special Form

This special form saves the current bounds of the accessible portion, evaluates the *body* forms, and finally restores the saved bounds, thus restoring the same state of narrowing (or absence thereof) formerly in effect. The state of narrowing is restored even in the event of an abnormal exit via `throw` or error (see Section 9.5 [Nonlocal Exits], page 135). Therefore, this construct is a clean way to narrow a buffer temporarily.

The value returned by `save-restriction` is that returned by the last form in *body*, or `nil` if no body forms were given.

Caution: it is easy to make a mistake when using the `save-restriction` construct. Read the entire description here before you try it.

If *body* changes the current buffer, `save-restriction` still restores the restrictions on the original buffer (the buffer whose restrictions it saved from), but it does not restore the identity of the current buffer.

`save-restriction` does *not* restore point and the mark; use `save-excursion` for that. If you use both `save-restriction` and `save-excursion` together, `save-excursion` should come first (on the outside). Otherwise, the old point value would be restored with temporary narrowing still in effect. If the old point value were outside the limits of the temporary narrowing, this would fail to restore it accurately.

The `save-restriction` special form records the values of the beginning and end of the accessible portion as distances from the beginning and end of the buffer. In other words, it records the amount of inaccessible text before and after the accessible portion.

This method yields correct results if *body* does further narrowing. However, `save-restriction` can become confused if the body widens and then makes changes outside the range of the saved narrowing. When this is what you want to do, `save-restriction` is not the right tool for the job. Here is what you must use instead:

```
(let ((beg (point-min-marker))
      (end (point-max-marker)))
  (unwind-protect
      (progn body)
    (save-excursion
      (set-buffer (marker-buffer beg))
      (narrow-to-region beg end))))
```

Here is a simple example of correct use of `save-restriction`:

```
---------- Buffer: foo ----------
This is the contents of foo
This is the contents of foo
This is the contents of foo*
---------- Buffer: foo ----------
```

```
(save-excursion
  (save-restriction
    (goto-char 1)
    (forward-line 2)
    (narrow-to-region 1 (point))
    (goto-char (point-min))
    (replace-string "foo" "bar")))

---------- Buffer: foo ----------
This is the contents of bar
This is the contents of bar
This is the contents of foo*
---------- Buffer: foo ----------
```

30 Markers

A *marker* is a Lisp object used to specify a position in a buffer relative to the surrounding text. A marker changes its offset from the beginning of the buffer automatically whenever text is inserted or deleted, so that it stays with the two characters on either side of it.

30.1 Overview of Markers

A marker specifies a buffer and a position in that buffer. The marker can be used to represent a position in the functions that require one, just as an integer could be used. See Chapter 29 [Positions], page 551, for a complete description of positions.

A marker has two attributes: the marker position, and the marker buffer. The marker position is an integer that is equivalent (at a given time) to the marker as a position in that buffer. But the marker's position value can change often during the life of the marker. Insertion and deletion of text in the buffer relocate the marker. The idea is that a marker positioned between two characters remains between those two characters despite insertion and deletion elsewhere in the buffer. Relocation changes the integer equivalent of the marker.

Deleting text around a marker's position leaves the marker between the characters immediately before and after the deleted text. Inserting text at the position of a marker normally leaves the marker either in front of or after the new text, depending on the marker's *insertion type* (see Section 30.5 [Marker Insertion Types], page 569)—unless the insertion is done with `insert-before-markers` (see Section 31.4 [Insertion], page 578).

Insertion and deletion in a buffer must check all the markers and relocate them if necessary. This slows processing in a buffer with a large number of markers. For this reason, it is a good idea to make a marker point nowhere if you are sure you don't need it any more. Unreferenced markers are garbage collected eventually, but until then will continue to use time if they do point somewhere.

Because it is common to perform arithmetic operations on a marker position, most of the arithmetic operations (including `+` and `-`) accept markers as arguments. In such cases, the marker stands for its current position.

Here are examples of creating markers, setting markers, and moving point to markers:

```
;; Make a new marker that initially does not point anywhere:
(setq m1 (make-marker))
     ⇒ #<marker in no buffer>
```

```
;; Set m1 to point between the 99th and 100th characters
;;    in the current buffer:
(set-marker m1 100)
     ⇒ #<marker at 100 in markers.texi>

;; Now insert one character at the beginning of the buffer:
(goto-char (point-min))
     ⇒ 1
(insert "Q")
     ⇒ nil

;; m1 is updated appropriately.
m1
     ⇒ #<marker at 101 in markers.texi>

;; Two markers that point to the same position
;;    are not eq, but they are equal.
(setq m2 (copy-marker m1))
     ⇒ #<marker at 101 in markers.texi>
(eq m1 m2)
     ⇒ nil
(equal m1 m2)
     ⇒ t

;; When you are finished using a marker, make it point nowhere.
(set-marker m1 nil)
     ⇒ #<marker in no buffer>
```

30.2 Predicates on Markers

You can test an object to see whether it is a marker, or whether it is either an integer or a marker. The latter test is useful in connection with the arithmetic functions that work with both markers and integers.

markerp *object* Function

This function returns `t` if *object* is a marker, `nil` otherwise. Note that integers are not markers, even though many functions will accept either a marker or an integer.

integer-or-marker-p *object* Function

This function returns `t` if *object* is an integer or a marker, `nil` otherwise.

number-or-marker-p *object* Function

This function returns `t` if *object* is a number (either integer or floating point) or a marker, `nil` otherwise.

30.3 Functions That Create Markers

When you create a new marker, you can make it point nowhere, or point to the present position of point, or to the beginning or end of the accessible portion of the buffer, or to the same place as another given marker.

make-marker Function

This function returns a newly created marker that does not point anywhere.

```
(make-marker)
     ⇒ #<marker in no buffer>
```

point-marker Function

This function returns a new marker that points to the present position of point in the current buffer. See Section 29.1 [Point], page 551. For an example, see `copy-marker`, below.

point-min-marker Function

This function returns a new marker that points to the beginning of the accessible portion of the buffer. This will be the beginning of the buffer unless narrowing is in effect. See Section 29.4 [Narrowing], page 561.

point-max-marker Function

This function returns a new marker that points to the end of the accessible portion of the buffer. This will be the end of the buffer unless narrowing is in effect. See Section 29.4 [Narrowing], page 561.

Here are examples of this function and `point-min-marker`, shown in a buffer containing a version of the source file for the text of this chapter.

```
(point-min-marker)
     ⇒ #<marker at 1 in markers.texi>
(point-max-marker)
     ⇒ #<marker at 15573 in markers.texi>

(narrow-to-region 100 200)
     ⇒ nil
(point-min-marker)
     ⇒ #<marker at 100 in markers.texi>
(point-max-marker)
     ⇒ #<marker at 200 in markers.texi>
```

copy-marker *marker-or-integer insertion-type* Function

If passed a marker as its argument, `copy-marker` returns a new marker that points to the same place and the same buffer as does *marker-or-integer*. If passed an integer as its argument, `copy-marker` returns a new marker that points to position *marker-or-integer* in the current buffer.

The new marker's insertion type is specified by the argument *insertion-type*. See Section 30.5 [Marker Insertion Types], page 569.

If passed an integer argument less than 1, `copy-marker` returns a new marker that points to the beginning of the current buffer. If passed an integer argument greater than the length of the buffer, `copy-marker` returns a new marker that points to the end of the buffer.

```
(copy-marker 0)
     ⇒ #<marker at 1 in markers.texi>

(copy-marker 20000)
     ⇒ #<marker at 7572 in markers.texi>
```

An error is signaled if *marker* is neither a marker nor an integer.

Two distinct markers are considered `equal` (even though not `eq`) to each other if they have the same position and buffer, or if they both point nowhere.

```
(setq p (point-marker))
     ⇒ #<marker at 2139 in markers.texi>

(setq q (copy-marker p))
     ⇒ #<marker at 2139 in markers.texi>

(eq p q)
     ⇒ nil

(equal p q)
     ⇒ t
```

30.4 Information from Markers

This section describes the functions for accessing the components of a marker object.

marker-position *marker* Function

This function returns the position that *marker* points to, or `nil` if it points nowhere.

marker-buffer *marker* Function

This function returns the buffer that *marker* points into, or `nil` if it points nowhere.

```
(setq m (make-marker))
     ⇒ #<marker in no buffer>
(marker-position m)
     ⇒ nil
(marker-buffer m)
     ⇒ nil
```

```
(set-marker m 3770 (current-buffer))
     ⇒ #<marker at 3770 in markers.texi>
(marker-buffer m)
     ⇒ #<buffer markers.texi>
(marker-position m)
     ⇒ 3770
```

30.5 Marker Insertion Types

When you insert text directly at the place where a marker points, there are two possible ways to relocate that marker: it can point before the inserted text, or point after it. You can specify which one a given marker should do by setting its *insertion type*. Note that use of `insert-before-markers` ignores markers' insertion types, always relocating a marker to point after the inserted text.

set-marker-insertion-type *marker type* Function

This function sets the insertion type of marker *marker* to *type*. If *type* is `t`, *marker* will advance when text is inserted at its position. If *type* is `nil`, *marker* does not advance when text is inserted there.

marker-insertion-type *marker* Function

This function reports the current insertion type of *marker*.

30.6 Moving Marker Positions

This section describes how to change the position of an existing marker. When you do this, be sure you know whether the marker is used outside of your program, and, if so, what effects will result from moving it—otherwise, confusing things may happen in other parts of Emacs.

set-marker *marker position* &optional *buffer* Function

This function moves *marker* to *position* in *buffer*. If *buffer* is not provided, it defaults to the current buffer.

If *position* is less than 1, `set-marker` moves *marker* to the beginning of the buffer. If *position* is greater than the size of the buffer, `set-marker` moves marker to the end of the buffer. If *position* is `nil` or a marker that points nowhere, then *marker* is set to point nowhere.

The value returned is *marker*.

```
(setq m (point-marker))
     ⇒ #<marker at 4714 in markers.texi>
(set-marker m 55)
     ⇒ #<marker at 55 in markers.texi>
```

```
(setq b (get-buffer "foo"))
     ⇒ #<buffer foo>
(set-marker m 0 b)
     ⇒ #<marker at 1 in foo>
```

move-marker *marker position* &optional *buffer* Function
This is another name for `set-marker`.

30.7 The Mark

One special marker in each buffer is designated *the mark*. It records a position for the user for the sake of commands such as `kill-region` and `indent-rigidly`. Lisp programs should set the mark only to values that have a potential use to the user, and never for their own internal purposes. For example, the `replace-regexp` command sets the mark to the value of point before doing any replacements, because this enables the user to move back there conveniently after the replace is finished.

Many commands are designed so that when called interactively they operate on the text between point and the mark. If you are writing such a command, don't examine the mark directly; instead, use `interactive` with the '`r`' specification. This provides the values of point and the mark as arguments to the command in an interactive call, but permits other Lisp programs to specify arguments explicitly. See Section 20.2.2 [Interactive Codes], page 322.

Each buffer has its own value of the mark that is independent of the value of the mark in other buffers. When a buffer is created, the mark exists but does not point anywhere. We consider this state as "the absence of a mark in that buffer."

Once the mark "exists" in a buffer, it normally never ceases to exist. However, it may become *inactive*, if Transient Mark mode is enabled. The variable `mark-active`, which is always buffer-local in all buffers, indicates whether the mark is active: non-`nil` means yes. A command can request deactivation of the mark upon return to the editor command loop by setting `deactivate-mark` to a non-`nil` value (but this causes deactivation only if Transient Mark mode is enabled).

The main motivation for using Transient Mark mode is that this mode also enables highlighting of the region when the mark is active. See Chapter 38 [Display], page 739.

In addition to the mark, each buffer has a *mark ring* which is a list of markers containing previous values of the mark. When editing commands change the mark, they should normally save the old value of the mark on the mark ring. The variable `mark-ring-max` specifies the maximum number of entries in the mark ring; once the list becomes this long, adding a new element deletes the last element.

mark &optional *force* Function

This function returns the current buffer's mark position as an integer.

If the mark is inactive, `mark` normally signals an error. However, if *force* is non-`nil`, then `mark` returns the mark position anyway—or `nil`, if the mark is not yet set for this buffer.

mark-marker Function

This function returns the current buffer's mark. This is the very marker that records the mark location inside Emacs, not a copy. Therefore, changing this marker's position will directly affect the position of the mark. Don't do it unless that is the effect you want.

```
(setq m (mark-marker))
     ⇒ #<marker at 3420 in markers.texi>
(set-marker m 100)
     ⇒ #<marker at 100 in markers.texi>
(mark-marker)
     ⇒ #<marker at 100 in markers.texi>
```

Like any marker, this marker can be set to point at any buffer you like. We don't recommend that you make it point at any buffer other than the one of which it is the mark. If you do, it will yield perfectly consistent, but rather odd, results.

set-mark *position* Function

This function sets the mark to *position*, and activates the mark. The old value of the mark is *not* pushed onto the mark ring.

Please note: Use this function only if you want the user to see that the mark has moved, and you want the previous mark position to be lost. Normally, when a new mark is set, the old one should go on the `mark-ring`. For this reason, most applications should use `push-mark` and `pop-mark`, not `set-mark`.

Novice Emacs Lisp programmers often try to use the mark for the wrong purposes. The mark saves a location for the user's convenience. An editing command should not alter the mark unless altering the mark is part of the user-level functionality of the command. (And, in that case, this effect should be documented.) To remember a location for internal use in the Lisp program, store it in a Lisp variable. For example:

```
(let ((beg (point)))
  (forward-line 1)
  (delete-region beg (point))).
```

push-mark &optional *position nomsg activate* Function

This function sets the current buffer's mark to *position*, and pushes a copy of the previous mark onto `mark-ring`. If *position* is `nil`, then the value of point is used. `push-mark` returns `nil`.

The function `push-mark` normally *does not* activate the mark. To do that, specify `t` for the argument *activate*.

A '`Mark set`' message is displayed unless *nomsg* is non-`nil`.

pop-mark Function

This function pops off the top element of `mark-ring` and makes that mark become the buffer's actual mark. This does not move point in the buffer, and it does nothing if `mark-ring` is empty. It deactivates the mark.

The return value is not meaningful.

transient-mark-mode User Option

This variable if non-`nil` enables Transient Mark mode, in which every buffer-modifying primitive sets `deactivate-mark`. The consequence of this is that commands that modify the buffer normally make the mark inactive.

mark-even-if-inactive User Option

If this is non-`nil`, Lisp programs and the Emacs user can use the mark even when it is inactive. This option affects the behavior of Transient Mark mode. When the option is non-`nil`, deactivation of the mark turns off region highlighting, but commands that use the mark behave as if the mark were still active.

deactivate-mark Variable

If an editor command sets this variable non-`nil`, then the editor command loop deactivates the mark after the command returns (if Transient Mark mode is enabled). All the primitives that change the buffer set `deactivate-mark`, to deactivate the mark when the command is finished.

deactivate-mark Function

This function deactivates the mark, if Transient Mark mode is enabled. Otherwise it does nothing.

mark-active Variable

The mark is active when this variable is non-`nil`. This variable is always buffer-local in each buffer.

activate-mark-hook Variable
deactivate-mark-hook Variable

These normal hooks are run, respectively, when the mark becomes active and when it becomes inactive. The hook `activate-mark-hook` is also run at the end of a command if the mark is active and it is possible that the region may have changed.

mark-ring Variable

The value of this buffer-local variable is the list of saved former marks of the current buffer, most recent first.

```
mark-ring
⇒ (#<marker at 11050 in markers.texi>
    #<marker at 10832 in markers.texi>
    ...)
```

mark-ring-max User Option

The value of this variable is the maximum size of `mark-ring`. If more marks than this are pushed onto the `mark-ring`, `push-mark` discards an old mark when it adds a new one.

30.8 The Region

The text between point and the mark is known as *the region*. Various functions operate on text delimited by point and the mark, but only those functions specifically related to the region itself are described here.

region-beginning Function

This function returns the position of the beginning of the region (as an integer). This is the position of either point or the mark, whichever is smaller.

If the mark does not point anywhere, an error is signaled.

region-end Function

This function returns the position of the end of the region (as an integer). This is the position of either point or the mark, whichever is larger.

If the mark does not point anywhere, an error is signaled.

Few programs need to use the `region-beginning` and `region-end` functions. A command designed to operate on a region should normally use `interactive` with the '`r`' specification to find the beginning and end of the region. This lets other Lisp programs specify the bounds explicitly as arguments. (See Section 20.2.2 [Interactive Codes], page 322.)

31 Text

This chapter describes the functions that deal with the text in a buffer. Most examine, insert, or delete text in the current buffer, often in the vicinity of point. Many are interactive. All the functions that change the text provide for undoing the changes (see Section 31.9 [Undo], page 590).

Many text-related functions operate on a region of text defined by two buffer positions passed in arguments named *start* and *end*. These arguments should be either markers (see Chapter 30 [Markers], page 565) or numeric character positions (see Chapter 29 [Positions], page 551). The order of these arguments does not matter; it is all right for *start* to be the end of the region and *end* the beginning. For example, `(delete-region 1 10)` and `(delete-region 10 1)` are equivalent. An `args-out-of-range` error is signaled if either *start* or *end* is outside the accessible portion of the buffer. In an interactive call, point and the mark are used for these arguments.

Throughout this chapter, "text" refers to the characters in the buffer, together with their properties (when relevant).

31.1 Examining Text Near Point

Many functions are provided to look at the characters around point. Several simple functions are described here. See also `looking-at` in Section 33.3 [Regexp Search], page 656.

char-after &optional *position* Function

This function returns the character in the current buffer at (i.e., immediately after) position *position*. If *position* is out of range for this purpose, either before the beginning of the buffer, or at or beyond the end, then the value is `nil`. The default for *position* is point.

In the following example, assume that the first character in the buffer is '`@`':

```
(char-to-string (char-after 1))
     ⇒ "@"
```

char-before &optional *position* Function

This function returns the character in the current buffer immediately before position *position*. If *position* is out of range for this purpose, either before the beginning of the buffer, or at or beyond the end, then the value is `nil`. The default for *position* is point.

following-char Function

This function returns the character following point in the current buffer. This is similar to `(char-after (point))`. However, if point is at the end of the buffer, then `following-char` returns 0.

Remember that point is always between characters, and the terminal cursor normally appears over the character following point. Therefore, the character returned by `following-char` is the character the cursor is over.

In this example, point is between the 'a' and the 'c'.

```
---------- Buffer: foo ----------
Gentlemen may cry ''Pea*ce! Peace!,''
but there is no peace.
---------- Buffer: foo ----------

(char-to-string (preceding-char))
     ⇒ "a"
(char-to-string (following-char))
     ⇒ "c"
```

preceding-char Function

This function returns the character preceding point in the current buffer. See above, under `following-char`, for an example. If point is at the beginning of the buffer, `preceding-char` returns 0.

bobp Function

This function returns `t` if point is at the beginning of the buffer. If narrowing is in effect, this means the beginning of the accessible portion of the text. See also `point-min` in Section 29.1 [Point], page 551.

eobp Function

This function returns `t` if point is at the end of the buffer. If narrowing is in effect, this means the end of accessible portion of the text. See also `point-max` in See Section 29.1 [Point], page 551.

bolp Function

This function returns `t` if point is at the beginning of a line. See Section 29.2.4 [Text Lines], page 554. The beginning of the buffer (or of its accessible portion) always counts as the beginning of a line.

eolp Function

This function returns `t` if point is at the end of a line. The end of the buffer (or of its accessible portion) is always considered the end of a line.

31.2 Examining Buffer Contents

This section describes two functions that allow a Lisp program to convert any portion of the text in the buffer into a string.

buffer-substring *start end* Function

This function returns a string containing a copy of the text of the region defined by positions *start* and *end* in the current buffer. If the arguments are not positions in the accessible portion of the buffer, `buffer-substring` signals an `args-out-of-range` error.

It is not necessary for *start* to be less than *end*; the arguments can be given in either order. But most often the smaller argument is written first.

If the text being copied has any text properties, these are copied into the string along with the characters they belong to. See Section 31.19 [Text Properties], page 610. However, overlays (see Section 38.8 [Overlays], page 748) in the buffer and their properties are ignored, not copied.

```
---------- Buffer: foo ----------
This is the contents of buffer foo

---------- Buffer: foo ----------

(buffer-substring 1 10)
⇒ "This is t"
(buffer-substring (point-max) 10)
⇒ "he contents of buffer foo
"
```

buffer-substring-no-properties *start end* Function

This is like `buffer-substring`, except that it does not copy text properties, just the characters themselves. See Section 31.19 [Text Properties], page 610.

buffer-string Function

This function returns the contents of the entire accessible portion of the current buffer as a string. It is equivalent to

```
(buffer-substring (point-min) (point-max))
```

```
---------- Buffer: foo ----------
This is the contents of buffer foo

---------- Buffer: foo ----------

(buffer-string)
     ⇒ "This is the contents of buffer foo
"
```

thing-at-point *thing* Function

Return the *thing* around or next to point, as a string.

The argument *thing* is a symbol which specifies a kind of syntactic entity. Possibilities include `symbol`, `list`, `sexp`, `defun`, `filename`, `url`, `word`, `sentence`, `whitespace`, `line`, `page`, and others.

```
---------- Buffer: foo ----------
Gentlemen may cry ``Pea*ce! Peace!,''
but there is no peace.
---------- Buffer: foo ----------

(thing-at-point 'word)
     ⇒ "Peace"
(thing-at-point 'line)
     ⇒ "Gentlemen may cry ``Peace! Peace!,''\n"
(thing-at-point 'whitespace)
     ⇒ nil
```

31.3 Comparing Text

This function lets you compare portions of the text in a buffer, without copying them into strings first.

compare-buffer-substrings *buffer1 start1 end1 buffer2 start2 end2* Function

This function lets you compare two substrings of the same buffer or two different buffers. The first three arguments specify one substring, giving a buffer and two positions within the buffer. The last three arguments specify the other substring in the same way. You can use `nil` for *buffer1*, *buffer2*, or both to stand for the current buffer.

The value is negative if the first substring is less, positive if the first is greater, and zero if they are equal. The absolute value of the result is one plus the index of the first differing characters within the substrings.

This function ignores case when comparing characters if `case-fold-search` is non-`nil`. It always ignores text properties.

Suppose the current buffer contains the text '`foobarbar haha!rara!`'; then in this example the two substrings are '`rbar `' and '`rara!`'. The value is 2 because the first substring is greater at the second character.

```
(compare-buffer-substring nil 6 11 nil 16 21)
     ⇒ 2
```

31.4 Inserting Text

Insertion means adding new text to a buffer. The inserted text goes at point—between the character before point and the character after point. Some insertion functions leave point before the inserted text, while other

functions leave it after. We call the former insertion *after point* and the latter insertion *before point*.

Insertion relocates markers that point at positions after the insertion point, so that they stay with the surrounding text (see Chapter 30 [Markers], page 565). When a marker points at the place of insertion, insertion may or may not relocate the marker, depending on the marker's insertion type (see Section 30.5 [Marker Insertion Types], page 569). Certain special functions such as `insert-before-markers` relocate all such markers to point after the inserted text, regardless of the markers' insertion type.

Insertion functions signal an error if the current buffer is read-only.

These functions copy text characters from strings and buffers along with their properties. The inserted characters have exactly the same properties as the characters they were copied from. By contrast, characters specified as separate arguments, not part of a string or buffer, inherit their text properties from the neighboring text.

The insertion functions convert text from unibyte to multibyte in order to insert in a multibyte buffer, and vice versa—if the text comes from a string or from a buffer. However, they do not convert unibyte character codes 128 through 255 to multibyte characters, not even if the current buffer is a multibyte buffer. See Section 32.2 [Converting Representations], page 630.

insert &rest *args* Function

This function inserts the strings and/or characters *args* into the current buffer, at point, moving point forward. In other words, it inserts the text before point. An error is signaled unless all *args* are either strings or characters. The value is `nil`.

insert-before-markers &rest *args* Function

This function inserts the strings and/or characters *args* into the current buffer, at point, moving point forward. An error is signaled unless all *args* are either strings or characters. The value is `nil`.

This function is unlike the other insertion functions in that it relocates markers initially pointing at the insertion point, to point after the inserted text. If an overlay begins the insertion point, the inserted text falls outside the overlay; if a nonempty overlay ends at the insertion point, the inserted text falls inside that overlay.

insert-char *character* &optional *count inherit* Function

This function inserts *count* instances of *character* into the current buffer before point. The argument *count* should be a number (`nil` means 1), and *character* must be a character. The value is `nil`.

This function does not convert unibyte character codes 128 through 255 to multibyte characters, not even if the current buffer is a multibyte buffer. See Section 32.2 [Converting Representations], page 630.

If *inherit* is non-`nil`, then the inserted characters inherit sticky text properties from the two characters before and after the insertion point. See Section 31.19.6 [Sticky Properties], page 618.

insert-buffer-substring *from-buffer-or-name* &optional *start end* Function

This function inserts a portion of buffer *from-buffer-or-name* (which must already exist) into the current buffer before point. The text inserted is the region from *start* and *end*. (These arguments default to the beginning and end of the accessible portion of that buffer.) This function returns `nil`.

In this example, the form is executed with buffer '`bar`' as the current buffer. We assume that buffer '`bar`' is initially empty.

```
---------- Buffer: foo ----------
We hold these truths to be self-evident, that all
---------- Buffer: foo ----------

(insert-buffer-substring "foo" 1 20)
     ⇒ nil

---------- Buffer: bar ----------
We hold these truth*
---------- Buffer: bar ----------
```

See Section 31.19.6 [Sticky Properties], page 618, for other insertion functions that inherit text properties from the nearby text in addition to inserting it. Whitespace inserted by indentation functions also inherits text properties.

31.5 User-Level Insertion Commands

This section describes higher-level commands for inserting text, commands intended primarily for the user but useful also in Lisp programs.

insert-buffer *from-buffer-or-name* Command

This command inserts the entire contents of *from-buffer-or-name* (which must exist) into the current buffer after point. It leaves the mark after the inserted text. The value is `nil`.

self-insert-command *count* Command

This command inserts the last character typed; it does so *count* times, before point, and returns `nil`. Most printing characters are bound to this command. In routine use, `self-insert-command` is the most frequently called function in Emacs, but programs rarely use it except to install it on a keymap.

In an interactive call, *count* is the numeric prefix argument.

This command calls `auto-fill-function` whenever that is non-`nil` and the character inserted is a space or a newline (see Section 31.14 [Auto Filling], page 598).

This command performs abbrev expansion if Abbrev mode is enabled and the inserted character does not have word-constituent syntax. (See Chapter 35 [Abbrevs], page 683, and Section 34.2.1 [Syntax Class Table], page 670.)

This is also responsible for calling `blink-paren-function` when the inserted character has close parenthesis syntax (see Section 38.11 [Blinking], page 759).

newline &optional *number-of-newlines* Command

This command inserts newlines into the current buffer before point. If *number-of-newlines* is supplied, that many newline characters are inserted.

This function calls `auto-fill-function` if the current column number is greater than the value of `fill-column` and *number-of-newlines* is `nil`. Typically what `auto-fill-function` does is insert a newline; thus, the overall result in this case is to insert two newlines at different places: one at point, and another earlier in the line. `newline` does not auto-fill if *number-of-newlines* is non-`nil`.

This command indents to the left margin if that is not zero. See Section 31.12 [Margins], page 596.

The value returned is `nil`. In an interactive call, *count* is the numeric prefix argument.

split-line Command

This command splits the current line, moving the portion of the line after point down vertically so that it is on the next line directly below where it was before. Whitespace is inserted as needed at the beginning of the lower line, using the `indent-to` function. `split-line` returns the position of point.

Programs hardly ever use this function.

overwrite-mode Variable

This variable controls whether overwrite mode is in effect. The value should be `overwrite-mode-textual`, `overwrite-mode-binary`, or `nil`. `overwrite-mode-textual` specifies textual overwrite mode (treats newlines and tabs specially), and `overwrite-mode-binary` specifies binary overwrite mode (treats newlines and tabs like any other characters).

31.6 Deleting Text

Deletion means removing part of the text in a buffer, without saving it in the kill ring (see Section 31.8 [The Kill Ring], page 585). Deleted text can't be yanked, but can be reinserted using the undo mechanism (see Section 31.9 [Undo], page 590). Some deletion functions do save text in the kill ring in some special cases.

All of the deletion functions operate on the current buffer, and all return a value of `nil`.

erase-buffer Command

This function deletes the entire text of the current buffer, leaving it empty. If the buffer is read-only, it signals a `buffer-read-only` error. Otherwise, it deletes the text without asking for any confirmation. It returns `nil`.

Normally, deleting a large amount of text from a buffer inhibits further auto-saving of that buffer "because it has shrunk". However, `erase-buffer` does not do this, the idea being that the future text is not really related to the former text, and its size should not be compared with that of the former text.

delete-region *start end* Command

This command deletes the text in the current buffer in the region defined by *start* and *end*. The value is `nil`. If point was inside the deleted region, its value afterward is *start*. Otherwise, point relocates with the surrounding text, as markers do.

delete-char *count* &optional *killp* Command

This command deletes *count* characters directly after point, or before point if *count* is negative. If *killp* is non-`nil`, then it saves the deleted characters in the kill ring.

In an interactive call, *count* is the numeric prefix argument, and *killp* is the unprocessed prefix argument. Therefore, if a prefix argument is supplied, the text is saved in the kill ring. If no prefix argument is supplied, then one character is deleted, but not saved in the kill ring.

The value returned is always `nil`.

delete-backward-char *count* &optional *killp* Command

This command deletes *count* characters directly before point, or after point if *count* is negative. If *killp* is non-`nil`, then it saves the deleted characters in the kill ring.

In an interactive call, *count* is the numeric prefix argument, and *killp* is the unprocessed prefix argument. Therefore, if a prefix argument is supplied, the text is saved in the kill ring. If no prefix argument is supplied, then one character is deleted, but not saved in the kill ring.

The value returned is always `nil`.

backward-delete-char-untabify *count* &optional *killp* Command
This command deletes *count* characters backward, changing tabs into spaces. When the next character to be deleted is a tab, it is first replaced with the proper number of spaces to preserve alignment and then one of those spaces is deleted instead of the tab. If *killp* is non-`nil`, then the command saves the deleted characters in the kill ring.

Conversion of tabs to spaces happens only if *count* is positive. If it is negative, exactly $-$*count* characters after point are deleted.

In an interactive call, *count* is the numeric prefix argument, and *killp* is the unprocessed prefix argument. Therefore, if a prefix argument is supplied, the text is saved in the kill ring. If no prefix argument is supplied, then one character is deleted, but not saved in the kill ring.

The value returned is always `nil`.

backward-delete-char-untabify-method User Option
This option specifies how `backward-delete-char-untabify` should deal with whitespace. Possible values include `untabify`, the default, meaning convert a tab to many spaces and delete one; `hungry`, meaning delete all the whitespace characters before point with one command, and `nil`, meaning do nothing special for whitespace characters.

31.7 User-Level Deletion Commands

This section describes higher-level commands for deleting text, commands intended primarily for the user but useful also in Lisp programs.

delete-horizontal-space Command
This function deletes all spaces and tabs around point. It returns `nil`.

In the following examples, we call `delete-horizontal-space` four times, once on each line, with point between the second and third characters on the line each time.

```
---------- Buffer: foo ----------
I ⋆thought
I ⋆     thought
We⋆ thought
Yo⋆u thought
---------- Buffer: foo ----------
```

```
(delete-horizontal-space)   ; Four times.
     ⇒ nil

---------- Buffer: foo ----------
Ithought
Ithought
Wethought
You thought
---------- Buffer: foo ----------
```

delete-indentation &optional *join-following-p* Command

This function joins the line point is on to the previous line, deleting any whitespace at the join and in some cases replacing it with one space. If *join-following-p* is non-`nil`, `delete-indentation` joins this line to the following line instead. The function returns `nil`.

If there is a fill prefix, and the second of the lines being joined starts with the prefix, then `delete-indentation` deletes the fill prefix before joining the lines. See Section 31.12 [Margins], page 596.

In the example below, point is located on the line starting '`events`', and it makes no difference if there are trailing spaces in the preceding line.

```
---------- Buffer: foo ----------
When in the course of human
*    events, it becomes necessary
---------- Buffer: foo ----------

(delete-indentation)
       nil

---------- Buffer: foo ----------
When in the course of human* events, it becomes necessary
---------- Buffer: foo ----------
```

After the lines are joined, the function `fixup-whitespace` is responsible for deciding whether to leave a space at the junction.

fixup-whitespace Function

This function replaces all the whitespace surrounding point with either one space or no space, according to the context. It returns `nil`.

At the beginning or end of a line, the appropriate amount of space is none. Before a character with close parenthesis syntax, or after a character with open parenthesis or expression-prefix syntax, no space is also appropriate. Otherwise, one space is appropriate. See Section 34.2.1 [Syntax Class Table], page 670.

In the example below, `fixup-whitespace` is called the first time with point before the word '`spaces`' in the first line. For the second invocation, point is directly after the '`(`'.

```
---------- Buffer: foo ----------
This has too many     *spaces
This has too many spaces at the start of (*   this list)
---------- Buffer: foo ----------

(fixup-whitespace)
        nil
(fixup-whitespace)
        nil

---------- Buffer: foo ----------
This has too many spaces
This has too many spaces at the start of (this list)
---------- Buffer: foo ----------
```

just-one-space Command

This command replaces any spaces and tabs around point with a single space. It returns `nil`.

delete-blank-lines Command

This function deletes blank lines surrounding point. If point is on a blank line with one or more blank lines before or after it, then all but one of them are deleted. If point is on an isolated blank line, then it is deleted. If point is on a nonblank line, the command deletes all blank lines following it.

A blank line is defined as a line containing only tabs and spaces.

`delete-blank-lines` returns `nil`.

31.8 The Kill Ring

Kill functions delete text like the deletion functions, but save it so that the user can reinsert it by *yanking*. Most of these functions have '`kill-`' in their name. By contrast, the functions whose names start with '`delete-`' normally do not save text for yanking (though they can still be undone); these are "deletion" functions.

Most of the kill commands are primarily for interactive use, and are not described here. What we do describe are the functions provided for use in writing such commands. You can use these functions to write commands for killing text. When you need to delete text for internal purposes within a Lisp function, you should normally use deletion functions, so as not to disturb the kill ring contents. See Section 31.6 [Deletion], page 582.

Killed text is saved for later yanking in the *kill ring*. This is a list that holds a number of recent kills, not just the last text kill. We call this a "ring" because yanking treats it as having elements in a cyclic order. The list is kept in the variable `kill-ring`, and can be operated on with the usual functions

for lists; there are also specialized functions, described in this section, that treat it as a ring.

Some people think this use of the word "kill" is unfortunate, since it refers to operations that specifically *do not* destroy the entities "killed". This is in sharp contrast to ordinary life, in which death is permanent and "killed" entities do not come back to life. Therefore, other metaphors have been proposed. For example, the term "cut ring" makes sense to people who, in pre-computer days, used scissors and paste to cut up and rearrange manuscripts. However, it would be difficult to change the terminology now.

31.8.1 Kill Ring Concepts

The kill ring records killed text as strings in a list, most recent first. A short kill ring, for example, might look like this:

```
("some text" "a different piece of text" "even older text")
```

When the list reaches `kill-ring-max` entries in length, adding a new entry automatically deletes the last entry.

When kill commands are interwoven with other commands, each kill command makes a new entry in the kill ring. Multiple kill commands in succession build up a single kill-ring entry, which would be yanked as a unit; the second and subsequent consecutive kill commands add text to the entry made by the first one.

For yanking, one entry in the kill ring is designated the "front" of the ring. Some yank commands "rotate" the ring by designating a different element as the "front." But this virtual rotation doesn't change the list itself—the most recent entry always comes first in the list.

31.8.2 Functions for Killing

`kill-region` is the usual subroutine for killing text. Any command that calls this function is a "kill command" (and should probably have '`kill`' in its name). `kill-region` puts the newly killed text in a new element at the beginning of the kill ring or adds it to the most recent element. It determines automatically (using `last-command`) whether the previous command was a kill command, and if so appends the killed text to the most recent entry.

kill-region *start end* Command

This function kills the text in the region defined by *start* and *end*. The text is deleted but saved in the kill ring, along with its text properties. The value is always `nil`.

In an interactive call, *start* and *end* are point and the mark.

If the buffer is read-only, `kill-region` modifies the kill ring just the same, then signals an error without modifying the buffer. This is convenient because it lets the user use all the kill commands to copy text into the kill ring from a read-only buffer.

kill-read-only-ok User Option
If this option is non-`nil`, `kill-region` does not get an error if the buffer is read-only. Instead, it simply returns, updating the kill ring but not changing the buffer.

copy-region-as-kill *start end* Command
This command saves the region defined by *start* and *end* on the kill ring (including text properties), but does not delete the text from the buffer. It returns `nil`. It also indicates the extent of the text copied by moving the cursor momentarily, or by displaying a message in the echo area.

The command does not set `this-command` to `kill-region`, so a subsequent kill command does not append to the same kill ring entry.

Don't call `copy-region-as-kill` in Lisp programs unless you aim to support Emacs 18. For newer Emacs versions, it is better to use `kill-new` or `kill-append` instead. See Section 31.8.4 [Low-Level Kill Ring], page 588.

31.8.3 Functions for Yanking

Yanking means reinserting an entry of previously killed text from the kill ring. The text properties are copied too.

yank &optional *arg* Command
This command inserts before point the text in the first entry in the kill ring. It positions the mark at the beginning of that text, and point at the end.

If *arg* is a list (which occurs interactively when the user types `C-u` with no digits), then `yank` inserts the text as described above, but puts point before the yanked text and puts the mark after it.

If *arg* is a number, then `yank` inserts the *arg*th most recently killed text—the *arg*th element of the kill ring list.

`yank` does not alter the contents of the kill ring or rotate it. It returns `nil`.

yank-pop *arg* Command
This command replaces the just-yanked entry from the kill ring with a different entry from the kill ring.

This is allowed only immediately after a `yank` or another `yank-pop`. At such a time, the region contains text that was just inserted by yanking. `yank-pop` deletes that text and inserts in its place a different piece of killed text. It does not add the deleted text to the kill ring, since it is already in the kill ring somewhere.

If *arg* is `nil`, then the replacement text is the previous element of the kill ring. If *arg* is numeric, the replacement is the *arg*th previous kill. If *arg* is negative, a more recent kill is the replacement.

The sequence of kills in the kill ring wraps around, so that after the oldest one comes the newest one, and before the newest one goes the oldest.

The return value is always `nil`.

31.8.4 Low-Level Kill Ring

These functions and variables provide access to the kill ring at a lower level, but still convenient for use in Lisp programs, because they take care of interaction with window system selections (see Section 28.18 [Window System Selections], page 544).

current-kill *n* &optional *do-not-move* Function

The function `current-kill` rotates the yanking pointer, which designates the "front" of the kill ring, by *n* places (from newer kills to older ones), and returns the text at that place in the ring.

If the optional second argument *do-not-move* is non-`nil`, then `current-kill` doesn't alter the yanking pointer; it just returns the *n*th kill, counting from the current yanking pointer.

If *n* is zero, indicating a request for the latest kill, `current-kill` calls the value of `interprogram-paste-function` (documented below) before consulting the kill ring.

kill-new *string* Function

This function puts the text *string* into the kill ring as a new entry at the front of the ring. It discards the oldest entry if appropriate. It also invokes the value of `interprogram-cut-function` (see below).

kill-append *string before-p* Function

This function appends the text *string* to the first entry in the kill ring. Normally *string* goes at the end of the entry, but if *before-p* is non-`nil`, it goes at the beginning. This function also invokes the value of `interprogram-cut-function` (see below).

interprogram-paste-function Variable

This variable provides a way of transferring killed text from other programs, when you are using a window system. Its value should be `nil` or a function of no arguments.

If the value is a function, `current-kill` calls it to get the "most recent kill". If the function returns a non-`nil` value, then that value is used as the "most recent kill". If it returns `nil`, then the first element of `kill-ring` is used.

The normal use of this hook is to get the window system's primary selection as the most recent kill, even if the selection belongs to another application. See Section 28.18 [Window System Selections], page 544.

interprogram-cut-function Variable

This variable provides a way of communicating killed text to other programs, when you are using a window system. Its value should be `nil` or a function of one argument.

If the value is a function, `kill-new` and `kill-append` call it with the new first element of the kill ring as an argument.

The normal use of this hook is to set the window system's primary selection from the newly killed text. See Section 28.18 [Window System Selections], page 544.

31.8.5 Internals of the Kill Ring

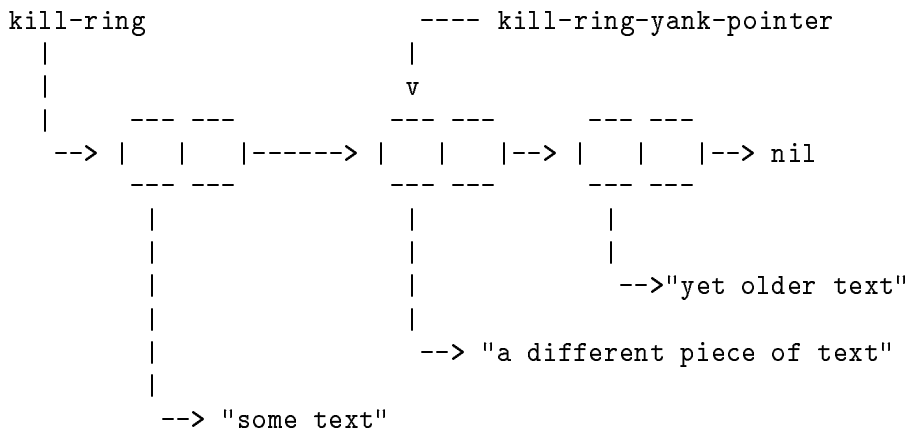
The variable `kill-ring` holds the kill ring contents, in the form of a list of strings. The most recent kill is always at the front of the list.

The `kill-ring-yank-pointer` variable points to a link in the kill ring list, whose CAR is the text to yank next. We say it identifies the "front" of the ring. Moving `kill-ring-yank-pointer` to a different link is called *rotating the kill ring*. We call the kill ring a "ring" because the functions that move the yank pointer wrap around from the end of the list to the beginning, or vice-versa. Rotation of the kill ring is virtual; it does not change the value of `kill-ring`.

Both `kill-ring` and `kill-ring-yank-pointer` are Lisp variables whose values are normally lists. The word "pointer" in the name of the `kill-ring-yank-pointer` indicates that the variable's purpose is to identify one element of the list for use by the next yank command.

The value of `kill-ring-yank-pointer` is always `eq` to one of the links in the kill ring list. The element it identifies is the CAR of that link. Kill commands, which change the kill ring, also set this variable to the value of `kill-ring`. The effect is to rotate the ring so that the newly killed text is at the front.

Here is a diagram that shows the variable `kill-ring-yank-pointer` pointing to the second entry in the kill ring `("some text" "a different piece of text" "yet older text")`.



This state of affairs might occur after `C-y` (`yank`) immediately followed by `M-y` (`yank-pop`).

kill-ring Variable

This variable holds the list of killed text sequences, most recently killed first.

kill-ring-yank-pointer Variable

This variable's value indicates which element of the kill ring is at the "front" of the ring for yanking. More precisely, the value is a tail of the value of `kill-ring`, and its CAR is the kill string that `C-y` should yank.

kill-ring-max User Option

The value of this variable is the maximum length to which the kill ring can grow, before elements are thrown away at the end. The default value for `kill-ring-max` is 30.

31.9 Undo

Most buffers have an *undo list*, which records all changes made to the buffer's text so that they can be undone. (The buffers that don't have one are usually special-purpose buffers for which Emacs assumes that undoing is not useful.) All the primitives that modify the text in the buffer automatically add elements to the front of the undo list, which is in the variable `buffer-undo-list`.

buffer-undo-list Variable

This variable's value is the undo list of the current buffer. A value of `t` disables the recording of undo information.

Here are the kinds of elements an undo list can have:

position This kind of element records a previous value of point; undoing this element moves point to *position*. Ordinary cursor motion does not make any sort of undo record, but deletion operations use these entries to record where point was before the command.

(*beg* . *end*)
This kind of element indicates how to delete text that was inserted. Upon insertion, the text occupied the range *beg*–*end* in the buffer.

(*text* . *position*)
This kind of element indicates how to reinsert text that was deleted. The deleted text itself is the string *text*. The place to reinsert it is `(abs` *position*`)`.

(`t` *high* . *low*)
This kind of element indicates that an unmodified buffer became modified. The elements *high* and *low* are two integers, each recording 16 bits of the visited file's modification time as of when it was previously visited or saved. `primitive-undo` uses those values to determine whether to mark the buffer as unmodified once again; it does so only if the file's modification time matches those numbers.

(`nil` *property value beg* . *end*)
This kind of element records a change in a text property. Here's how you might undo the change:

```
(put-text-property beg end property value)
```

(*marker* . *adjustment*)
This kind of element records the fact that the marker *marker* was relocated due to deletion of surrounding text, and that it moved *adjustment* character positions. Undoing this element moves *marker* − *adjustment* characters.

`nil` This element is a boundary. The elements between two boundaries are called a *change group*; normally, each change group corresponds to one keyboard command, and undo commands normally undo an entire group as a unit.

undo-boundary Function

This function places a boundary element in the undo list. The undo command stops at such a boundary, and successive undo commands undo to earlier and earlier boundaries. This function returns `nil`.

The editor command loop automatically creates an undo boundary before each key sequence is executed. Thus, each undo normally undoes the effects of one command. Self-inserting input characters are an exception. The command loop makes a boundary for the first such character; the

next 19 consecutive self-inserting input characters do not make boundaries, and then the 20th does, and so on as long as self-inserting characters continue.

All buffer modifications add a boundary whenever the previous undoable change was made in some other buffer. This is to ensure that each command makes a boundary in each buffer where it makes changes.

Calling this function explicitly is useful for splitting the effects of a command into more than one unit. For example, `query-replace` calls `undo-boundary` after each replacement, so that the user can undo individual replacements one by one.

primitive-undo *count list* Function

This is the basic function for undoing elements of an undo list. It undoes the first *count* elements of *list*, returning the rest of *list*. You could write this function in Lisp, but it is convenient to have it in C.

`primitive-undo` adds elements to the buffer's undo list when it changes the buffer. Undo commands avoid confusion by saving the undo list value at the beginning of a sequence of undo operations. Then the undo operations use and update the saved value. The new elements added by undoing are not part of this saved value, so they don't interfere with continuing to undo.

31.10 Maintaining Undo Lists

This section describes how to enable and disable undo information for a given buffer. It also explains how the undo list is truncated automatically so it doesn't get too big.

Recording of undo information in a newly created buffer is normally enabled to start with; but if the buffer name starts with a space, the undo recording is initially disabled. You can explicitly enable or disable undo recording with the following two functions, or by setting `buffer-undo-list` yourself.

buffer-enable-undo &optional *buffer-or-name* Command

This command enables recording undo information for buffer *buffer-or-name*, so that subsequent changes can be undone. If no argument is supplied, then the current buffer is used. This function does nothing if undo recording is already enabled in the buffer. It returns `nil`.

In an interactive call, *buffer-or-name* is the current buffer. You cannot specify any other buffer.

buffer-disable-undo &optional *buffer* Command
buffer-flush-undo &optional *buffer* Command

This function discards the undo list of *buffer*, and disables further recording of undo information. As a result, it is no longer possible to undo either

previous changes or any subsequent changes. If the undo list of *buffer* is already disabled, this function has no effect.

This function returns `nil`.

The name `buffer-flush-undo` is not considered obsolete, but the preferred name is `buffer-disable-undo`.

As editing continues, undo lists get longer and longer. To prevent them from using up all available memory space, garbage collection trims them back to size limits you can set. (For this purpose, the "size" of an undo list measures the cons cells that make up the list, plus the strings of deleted text.) Two variables control the range of acceptable sizes: `undo-limit` and `undo-strong-limit`.

undo-limit Variable

This is the soft limit for the acceptable size of an undo list. The change group at which this size is exceeded is the last one kept.

undo-strong-limit Variable

This is the upper limit for the acceptable size of an undo list. The change group at which this size is exceeded is discarded itself (along with all older change groups). There is one exception: the very latest change group is never discarded no matter how big it is.

31.11 Filling

Filling means adjusting the lengths of lines (by moving the line breaks) so that they are nearly (but no greater than) a specified maximum width. Additionally, lines can be *justified*, which means inserting spaces to make the left and/or right margins line up precisely. The width is controlled by the variable `fill-column`. For ease of reading, lines should be no longer than 70 or so columns.

You can use Auto Fill mode (see Section 31.14 [Auto Filling], page 598) to fill text automatically as you insert it, but changes to existing text may leave it improperly filled. Then you must fill the text explicitly.

Most of the commands in this section return values that are not meaningful. All the functions that do filling take note of the current left margin, current right margin, and current justification style (see Section 31.12 [Margins], page 596). If the current justification style is `none`, the filling functions don't actually do anything.

Several of the filling functions have an argument *justify*. If it is non-`nil`, that requests some kind of justification. It can be `left`, `right`, `full`, or `center`, to request a specific style of justification. If it is `t`, that means to use the current justification style for this part of the text (see `current-justification`, below). Any other value is treated as `full`.

When you call the filling functions interactively, using a prefix argument implies the value `full` for *justify*.

fill-paragraph *justify* Command

This command fills the paragraph at or after point. If *justify* is non-`nil`, each line is justified as well. It uses the ordinary paragraph motion commands to find paragraph boundaries. See section "Paragraphs" in *The Emacs Manual*.

fill-region *start end* &optional *justify nosqueeze* Command

This command fills each of the paragraphs in the region from *start* to *end*. It justifies as well if *justify* is non-`nil`.

If *nosqueeze* is non-`nil`, that means to leave whitespace other than line breaks untouched. If *to-eop* is non-`nil`, that means to keep filling to the end of the paragraph—or the next hard newline, if `use-hard-newlines` is enabled (see below).

The variable `paragraph-separate` controls how to distinguish paragraphs. See Section 33.8 [Standard Regexps], page 666.

fill-individual-paragraphs *start end* &optional *justify* *mail-flag* Command

This command fills each paragraph in the region according to its individual fill prefix. Thus, if the lines of a paragraph were indented with spaces, the filled paragraph will remain indented in the same fashion.

The first two arguments, *start* and *end*, are the beginning and end of the region to be filled. The third and fourth arguments, *justify* and *mail-flag*, are optional. If *justify* is non-`nil`, the paragraphs are justified as well as filled. If *mail-flag* is non-`nil`, it means the function is operating on a mail message and therefore should not fill the header lines.

Ordinarily, `fill-individual-paragraphs` regards each change in indentation as starting a new paragraph. If `fill-individual-varying-indent` is non-`nil`, then only separator lines separate paragraphs. That mode can handle indented paragraphs with additional indentation on the first line.

fill-individual-varying-indent User Option

This variable alters the action of `fill-individual-paragraphs` as described above.

fill-region-as-paragraph *start end* &optional *justify* *nosqueeze squeeze-after* Command

This command considers a region of text as a single paragraph and fills it. If the region was made up of many paragraphs, the blank lines between paragraphs are removed. This function justifies as well as filling when *justify* is non-`nil`.

In an interactive call, any prefix argument requests justification.

If *nosqueeze* is non-`nil`, that means to leave whitespace other than line breaks untouched. If *squeeze-after* is non-`nil`, it specifies a position in the region, and means don't canonicalize spaces before that position.

In Adaptive Fill mode, this command calls `fill-context-prefix` to choose a fill prefix by default. See Section 31.13 [Adaptive Fill], page 598.

justify-current-line *how eop nosqueeze* Command

This command inserts spaces between the words of the current line so that the line ends exactly at `fill-column`. It returns `nil`.

The argument *how*, if non-`nil` specifies explicitly the style of justification. It can be `left`, `right`, `full`, `center`, or `none`. If it is `t`, that means to do follow specified justification style (see `current-justification`, below). `nil` means to do full justification.

If *eop* is non-`nil`, that means do left-justification if `current-justification` specifies full justification. This is used for the last line of a paragraph; even if the paragraph as a whole is fully justified, the last line should not be.

If *nosqueeze* is non-`nil`, that means do not change interior whitespace.

default-justification User Option

This variable's value specifies the style of justification to use for text that doesn't specify a style with a text property. The possible values are `left`, `right`, `full`, `center`, or `none`. The default value is `left`.

current-justification Function

This function returns the proper justification style to use for filling the text around point.

sentence-end-double-space User Option

If this variable is non-`nil`, a period followed by just one space does not count as the end of a sentence, and the filling functions avoid breaking the line at such a place.

fill-paragraph-function Variable

This variable provides a way for major modes to override the filling of paragraphs. If the value is non-`nil`, `fill-paragraph` calls this function to do the work. If the function returns a non-`nil` value, `fill-paragraph` assumes the job is done, and immediately returns that value.

The usual use of this feature is to fill comments in programming language modes. If the function needs to fill a paragraph in the usual way, it can do so as follows:

```
(let ((fill-paragraph-function nil))
  (fill-paragraph arg))
```

use-hard-newlines Variable

If this variable is non-`nil`, the filling functions do not delete newlines that have the `hard` text property. These "hard newlines" act as paragraph separators.

31.12 Margins for Filling

fill-prefix User Option

This buffer-local variable specifies a string of text that appears at the beginning of normal text lines and should be disregarded when filling them. Any line that fails to start with the fill prefix is considered the start of a paragraph; so is any line that starts with the fill prefix followed by additional whitespace. Lines that start with the fill prefix but no additional whitespace are ordinary text lines that can be filled together. The resulting filled lines also start with the fill prefix.

The fill prefix follows the left margin whitespace, if any.

fill-column User Option

This buffer-local variable specifies the maximum width of filled lines. Its value should be an integer, which is a number of columns. All the filling, justification, and centering commands are affected by this variable, including Auto Fill mode (see Section 31.14 [Auto Filling], page 598).

As a practical matter, if you are writing text for other people to read, you should set `fill-column` to no more than 70. Otherwise the line will be too long for people to read comfortably, and this can make the text seem clumsy.

default-fill-column Variable

The value of this variable is the default value for `fill-column` in buffers that do not override it. This is the same as `(default-value 'fill-column)`.

The default value for `default-fill-column` is 70.

set-left-margin *from to margin* Command

This sets the `left-margin` property on the text from *from* to *to* to the value *margin*. If Auto Fill mode is enabled, this command also refills the region to fit the new margin.

set-right-margin *from to margin* Command

This sets the `right-margin` property on the text from *from* to *to* to the value *margin*. If Auto Fill mode is enabled, this command also refills the region to fit the new margin.

current-left-margin Function

This function returns the proper left margin value to use for filling the text around point. The value is the sum of the `left-margin` property of the character at the start of the current line (or zero if none), and the value of the variable `left-margin`.

current-fill-column Function

This function returns the proper fill column value to use for filling the text around point. The value is the value of the `fill-column` variable, minus the value of the `right-margin` property of the character after point.

move-to-left-margin &optional *n force* Command

This function moves point to the left margin of the current line. The column moved to is determined by calling the function `current-left-margin`. If the argument *n* is non-`nil`, `move-to-left-margin` moves forward $n-1$ lines first.

If *force* is non-`nil`, that says to fix the line's indentation if that doesn't match the left margin value.

delete-to-left-margin *from to* Function

This function removes left margin indentation from the text between *from* and *to*. The amount of indentation to delete is determined by calling `current-left-margin`. In no case does this function delete non-whitespace.

indent-to-left-margin Function

This is the default `indent-line-function`, used in Fundamental mode, Text mode, etc. Its effect is to adjust the indentation at the beginning of the current line to the value specified by the variable `left-margin`. This may involve either inserting or deleting whitespace.

left-margin Variable

This variable specifies the base left margin column. In Fundamental mode, `C-j` indents to this column. This variable automatically becomes buffer-local when set in any fashion.

fill-nobreak-predicate Variable

This variable gives major modes a way to specify not to break a line at certain places. Its value should be a function. This function is called during filling, with no arguments and with point located at the place where a break is being considered. If the function returns non-`nil`, then the line won't be broken there.

31.13 Adaptive Fill Mode

Adaptive Fill mode chooses a fill prefix automatically from the text in each paragraph being filled.

adaptive-fill-mode User Option
Adaptive Fill mode is enabled when this variable is non-`nil`. It is `t` by default.

fill-context-prefix *from to* Function
This function implements the heart of Adaptive Fill mode; it chooses a fill prefix based on the text between *from* and *to*. It does this by looking at the first two lines of the paragraph, based on the variables described below.

adaptive-fill-regexp User Option
This variable holds a regular expression to control Adaptive Fill mode. Adaptive Fill mode matches this regular expression against the text starting after the left margin whitespace (if any) on a line; the characters it matches are that line's candidate for the fill prefix.

adaptive-fill-first-line-regexp User Option
In a one-line paragraph, if the candidate fill prefix matches this regular expression, or if it matches `comment-start-skip`, then it is used—otherwise, spaces amounting to the same width are used instead.

However, the fill prefix is never taken from a one-line paragraph if it would act as a paragraph starter on subsequent lines.

adaptive-fill-function User Option
You can specify more complex ways of choosing a fill prefix automatically by setting this variable to a function. The function is called when `adaptive-fill-regexp` does not match, with point after the left margin of a line, and it should return the appropriate fill prefix based on that line. If it returns `nil`, that means it sees no fill prefix in that line.

31.14 Auto Filling

Auto Fill mode is a minor mode that fills lines automatically as text is inserted. This section describes the hook used by Auto Fill mode. For a description of functions that you can call explicitly to fill and justify existing text, see Section 31.11 [Filling], page 593.

Auto Fill mode also enables the functions that change the margins and justification style to refill portions of the text. See Section 31.12 [Margins], page 596.

auto-fill-function Variable

The value of this variable should be a function (of no arguments) to be called after self-inserting a space or a newline. It may be `nil`, in which case nothing special is done in that case.

The value of `auto-fill-function` is `do-auto-fill` when Auto-Fill mode is enabled. That is a function whose sole purpose is to implement the usual strategy for breaking a line.

> In older Emacs versions, this variable was named `auto-fill-hook`, but since it is not called with the standard convention for hooks, it was renamed to `auto-fill-function` in version 19.

normal-auto-fill-function Variable

This variable specifies the function to use for `auto-fill-function`, if and when Auto Fill is turned on. Major modes can set buffer-local values for this variable to alter how Auto Fill works.

31.15 Sorting Text

The sorting functions described in this section all rearrange text in a buffer. This is in contrast to the function `sort`, which rearranges the order of the elements of a list (see Section 5.6.3 [Rearrangement], page 86). The values returned by these functions are not meaningful.

sort-subr *reverse nextrecfun endrecfun* &optional *startkeyfun endkeyfun* Function

This function is the general text-sorting routine that subdivides a buffer into records and then sorts them. Most of the commands in this section use this function.

To understand how `sort-subr` works, consider the whole accessible portion of the buffer as being divided into disjoint pieces called *sort records*. The records may or may not be contiguous, but they must not overlap. A portion of each sort record (perhaps all of it) is designated as the sort key. Sorting rearranges the records in order by their sort keys.

Usually, the records are rearranged in order of ascending sort key. If the first argument to the `sort-subr` function, *reverse*, is non-`nil`, the sort records are rearranged in order of descending sort key.

The next four arguments to `sort-subr` are functions that are called to move point across a sort record. They are called many times from within `sort-subr`.

1. *nextrecfun* is called with point at the end of a record. This function moves point to the start of the next record. The first record is assumed to start at the position of point when `sort-subr` is called. Therefore, you should usually move point to the beginning of the buffer before calling `sort-subr`.

This function can indicate there are no more sort records by leaving point at the end of the buffer.

2. *endrecfun* is called with point within a record. It moves point to the end of the record.

3. *startkeyfun* is called to move point from the start of a record to the start of the sort key. This argument is optional; if it is omitted, the whole record is the sort key. If supplied, the function should either return a non-`nil` value to be used as the sort key, or return `nil` to indicate that the sort key is in the buffer starting at point. In the latter case, *endkeyfun* is called to find the end of the sort key.

4. *endkeyfun* is called to move point from the start of the sort key to the end of the sort key. This argument is optional. If *startkeyfun* returns `nil` and this argument is omitted (or `nil`), then the sort key extends to the end of the record. There is no need for *endkeyfun* if *startkeyfun* returns a non-`nil` value.

As an example of `sort-subr`, here is the complete function definition for `sort-lines`:

```
;; Note that the first two lines of doc string
;; are effectively one line when viewed by a user.
(defun sort-lines (reverse beg end)
  "Sort lines in region alphabetically;\
 argument means descending order.
Called from a program, there are three arguments:
REVERSE (non-nil means reverse order),\
 BEG and END (region to sort).
The variable `sort-fold-case' determines\
 whether alphabetic case affects
the sort order.
  (interactive "P\nr")
  (save-excursion
    (save-restriction
      (narrow-to-region beg end)
      (goto-char (point-min))
      (sort-subr reverse 'forward-line 'end-of-line))))
```

Here `forward-line` moves point to the start of the next record, and `end-of-line` moves point to the end of record. We do not pass the arguments *startkeyfun* and *endkeyfun*, because the entire record is used as the sort key.

The `sort-paragraphs` function is very much the same, except that its `sort-subr` call looks like this:

```
(sort-subr reverse
           (function
             (lambda ()
               (while (and (not (eobp))
                           (looking-at paragraph-separate))
                 (forward-line 1))))
           'forward-paragraph)
```

Markers pointing into any sort records are left with no useful position after `sort-subr` returns.

sort-fold-case User Option

If this variable is non-`nil`, `sort-subr` and the other buffer sorting functions ignore case when comparing strings.

sort-regexp-fields *reverse record-regexp key-regexp start end* Command

This command sorts the region between *start* and *end* alphabetically as specified by *record-regexp* and *key-regexp*. If *reverse* is a negative integer, then sorting is in reverse order.

Alphabetical sorting means that two sort keys are compared by comparing the first characters of each, the second characters of each, and so on. If a mismatch is found, it means that the sort keys are unequal; the sort key whose character is less at the point of first mismatch is the lesser sort key. The individual characters are compared according to their numerical character codes in the Emacs character set.

The value of the *record-regexp* argument specifies how to divide the buffer into sort records. At the end of each record, a search is done for this regular expression, and the text that matches it is taken as the next record. For example, the regular expression '`^.+$`', which matches lines with at least one character besides a newline, would make each such line into a sort record. See Section 33.2 [Regular Expressions], page 649, for a description of the syntax and meaning of regular expressions.

The value of the *key-regexp* argument specifies what part of each record is the sort key. The *key-regexp* could match the whole record, or only a part. In the latter case, the rest of the record has no effect on the sorted order of records, but it is carried along when the record moves to its new position.

The *key-regexp* argument can refer to the text matched by a subexpression of *record-regexp*, or it can be a regular expression on its own.

If *key-regexp* is:

'`\`*digit*' then the text matched by the *digit*th '`\(...\)`' parenthesis grouping in *record-regexp* is the sort key.

'`\&`' then the whole record is the sort key.

a regular expression
: then `sort-regexp-fields` searches for a match for the regular expression within the record. If such a match is found, it is the sort key. If there is no match for *key-regexp* within a record then that record is ignored, which means its position in the buffer is not changed. (The other records may move around it.)

For example, if you plan to sort all the lines in the region by the first word on each line starting with the letter '`f`', you should set *record-regexp* to '`^.*$`' and set *key-regexp* to '`\<f\w*\>`'. The resulting expression looks like this:

```
(sort-regexp-fields nil "^.*$" "\\<f\\w*\\>"
                    (region-beginning)
                    (region-end))
```

If you call `sort-regexp-fields` interactively, it prompts for *record-regexp* and *key-regexp* in the minibuffer.

sort-lines *reverse start end* Command
This command alphabetically sorts lines in the region between *start* and *end*. If *reverse* is non-`nil`, the sort is in reverse order.

sort-paragraphs *reverse start end* Command
This command alphabetically sorts paragraphs in the region between *start* and *end*. If *reverse* is non-`nil`, the sort is in reverse order.

sort-pages *reverse start end* Command
This command alphabetically sorts pages in the region between *start* and *end*. If *reverse* is non-`nil`, the sort is in reverse order.

sort-fields *field start end* Command
This command sorts lines in the region between *start* and *end*, comparing them alphabetically by the *field*th field of each line. Fields are separated by whitespace and numbered starting from 1. If *field* is negative, sorting is by the $-$*field*th field from the end of the line. This command is useful for sorting tables.

sort-numeric-fields *field start end* Command
This command sorts lines in the region between *start* and *end*, comparing them numerically by the *field*th field of each line. The specified field must contain a number in each line of the region. Fields are separated by whitespace and numbered starting from 1. If *field* is negative, sorting is by the $-$*field*th field from the end of the line. This command is useful for sorting tables.

sort-columns *reverse* &optional *beg end* Command

This command sorts the lines in the region between *beg* and *end*, comparing them alphabetically by a certain range of columns. The column positions of *beg* and *end* bound the range of columns to sort on.

If *reverse* is non-`nil`, the sort is in reverse order.

One unusual thing about this command is that the entire line containing position *beg*, and the entire line containing position *end*, are included in the region sorted.

Note that `sort-columns` uses the `sort` utility program, and so cannot work properly on text containing tab characters. Use *`M-x untabify`* to convert tabs to spaces before sorting.

31.16 Counting Columns

The column functions convert between a character position (counting characters from the beginning of the buffer) and a column position (counting screen characters from the beginning of a line).

These functions count each character according to the number of columns it occupies on the screen. This means control characters count as occupying 2 or 4 columns, depending upon the value of `ctl-arrow`, and tabs count as occupying a number of columns that depends on the value of `tab-width` and on the column where the tab begins. See Section 38.13 [Usual Display], page 760.

Column number computations ignore the width of the window and the amount of horizontal scrolling. Consequently, a column value can be arbitrarily high. The first (or leftmost) column is numbered 0.

current-column Function

This function returns the horizontal position of point, measured in columns, counting from 0 at the left margin. The column position is the sum of the widths of all the displayed representations of the characters between the start of the current line and point.

For an example of using `current-column`, see the description of `count-lines` in Section 29.2.4 [Text Lines], page 554.

move-to-column *column* &optional *force* Function

This function moves point to *column* in the current line. The calculation of *column* takes into account the widths of the displayed representations of the characters between the start of the line and point.

If column *column* is beyond the end of the line, point moves to the end of the line. If *column* is negative, point moves to the beginning of the line.

If it is impossible to move to column *column* because that is in the middle of a multicolumn character such as a tab, point moves to the end of that character. However, if *force* is non-`nil`, and *column* is in the middle

of a tab, then `move-to-column` converts the tab into spaces so that it can move precisely to column *column*. Other multicolumn characters can cause anomalies despite *force*, since there is no way to split them.

The argument *force* also has an effect if the line isn't long enough to reach column *column*; in that case, it says to add whitespace at the end of the line to reach that column.

If *column* is not an integer, an error is signaled.

The return value is the column number actually moved to.

31.17 Indentation

The indentation functions are used to examine, move to, and change whitespace that is at the beginning of a line. Some of the functions can also change whitespace elsewhere on a line. Columns and indentation count from zero at the left margin.

31.17.1 Indentation Primitives

This section describes the primitive functions used to count and insert indentation. The functions in the following sections use these primitives. See Section 38.9 [Width], page 753, for related functions.

current-indentation Function

This function returns the indentation of the current line, which is the horizontal position of the first nonblank character. If the contents are entirely blank, then this is the horizontal position of the end of the line.

indent-to *column* &optional *minimum* Command

This function indents from point with tabs and spaces until *column* is reached. If *minimum* is specified and non-`nil`, then at least that many spaces are inserted even if this requires going beyond *column*. Otherwise the function does nothing if point is already beyond *column*. The value is the column at which the inserted indentation ends.

The inserted whitespace characters inherit text properties from the surrounding text (usually, from the preceding text only). See Section 31.19.6 [Sticky Properties], page 618.

indent-tabs-mode User Option

If this variable is non-`nil`, indentation functions can insert tabs as well as spaces. Otherwise, they insert only spaces. Setting this variable automatically makes it buffer-local in the current buffer.

31.17.2 Indentation Controlled by Major Mode

An important function of each major mode is to customize the TAB key to indent properly for the language being edited. This section describes the mechanism of the TAB key and how to control it. The functions in this section return unpredictable values.

indent-line-function Variable

This variable's value is the function to be used by TAB (and various commands) to indent the current line. The command `indent-according-to-mode` does no more than call this function.

In Lisp mode, the value is the symbol `lisp-indent-line`; in C mode, `c-indent-line`; in Fortran mode, `fortran-indent-line`. In Fundamental mode, Text mode, and many other modes with no standard for indentation, the value is `indent-to-left-margin` (which is the default value).

indent-according-to-mode Command

This command calls the function in `indent-line-function` to indent the current line in a way appropriate for the current major mode.

indent-for-tab-command Command

This command calls the function in `indent-line-function` to indent the current line; however, if that function is `indent-to-left-margin`, `insert-tab` is called instead. (That is a trivial command that inserts a tab character.)

newline-and-indent Command

This function inserts a newline, then indents the new line (the one following the newline just inserted) according to the major mode.

It does indentation by calling the current `indent-line-function`. In programming language modes, this is the same thing TAB does, but in some text modes, where TAB inserts a tab, `newline-and-indent` indents to the column specified by `left-margin`.

reindent-then-newline-and-indent Command

This command reindents the current line, inserts a newline at point, and then indents the new line (the one following the newline just inserted).

This command does indentation on both lines according to the current major mode, by calling the current value of `indent-line-function`. In programming language modes, this is the same thing TAB does, but in some text modes, where TAB inserts a tab, `reindent-then-newline-and-indent` indents to the column specified by `left-margin`.

31.17.3 Indenting an Entire Region

This section describes commands that indent all the lines in the region. They return unpredictable values.

indent-region *start end to-column* Command

This command indents each nonblank line starting between *start* (inclusive) and *end* (exclusive). If *to-column* is `nil`, `indent-region` indents each nonblank line by calling the current mode's indentation function, the value of `indent-line-function`.

If *to-column* is non-`nil`, it should be an integer specifying the number of columns of indentation; then this function gives each line exactly that much indentation, by either adding or deleting whitespace.

If there is a fill prefix, `indent-region` indents each line by making it start with the fill prefix.

indent-region-function Variable

The value of this variable is a function that can be used by `indent-region` as a short cut. It should take two arguments, the start and end of the region. You should design the function so that it will produce the same results as indenting the lines of the region one by one, but presumably faster.

If the value is `nil`, there is no short cut, and `indent-region` actually works line by line.

A short-cut function is useful in modes such as C mode and Lisp mode, where the `indent-line-function` must scan from the beginning of the function definition: applying it to each line would be quadratic in time. The short cut can update the scan information as it moves through the lines indenting them; this takes linear time. In a mode where indenting a line individually is fast, there is no need for a short cut.

`indent-region` with a non-`nil` argument *to-column* has a different meaning and does not use this variable.

indent-rigidly *start end count* Command

This command indents all lines starting between *start* (inclusive) and *end* (exclusive) sideways by *count* columns. This "preserves the shape" of the affected region, moving it as a rigid unit. Consequently, this command is useful not only for indenting regions of unindented text, but also for indenting regions of formatted code.

For example, if *count* is 3, this command adds 3 columns of indentation to each of the lines beginning in the region specified.

In Mail mode, `C-c C-y` (`mail-yank-original`) uses `indent-rigidly` to indent the text copied from the message being replied to.

indent-code-rigidly *start end columns* &optional *nochange-regexp* Function

This is like `indent-rigidly`, except that it doesn't alter lines that start within strings or comments.

In addition, it doesn't alter a line if *nochange-regexp* matches at the beginning of the line (if *nochange-regexp* is non-`nil`).

31.17.4 Indentation Relative to Previous Lines

This section describes two commands that indent the current line based on the contents of previous lines.

indent-relative &optional *unindented-ok* Command

This command inserts whitespace at point, extending to the same column as the next *indent point* of the previous nonblank line. An indent point is a non-whitespace character following whitespace. The next indent point is the first one at a column greater than the current column of point. For example, if point is underneath and to the left of the first non-blank character of a line of text, it moves to that column by inserting whitespace.

If the previous nonblank line has no next indent point (i.e., none at a great enough column position), `indent-relative` either does nothing (if *unindented-ok* is non-`nil`) or calls `tab-to-tab-stop`. Thus, if point is underneath and to the right of the last column of a short line of text, this command ordinarily moves point to the next tab stop by inserting whitespace.

The return value of `indent-relative` is unpredictable.

In the following example, point is at the beginning of the second line:

```
            This line is indented twelve spaces.
    *The quick brown fox jumped.
```

Evaluation of the expression `(indent-relative nil)` produces the following:

```
            This line is indented twelve spaces.
            *The quick brown fox jumped.
```

In this next example, point is between the '`m`' and '`p`' of '`jumped`':

```
            This line is indented twelve spaces.
    The quick brown fox jum*ped.
```

Evaluation of the expression `(indent-relative nil)` produces the following:

```
            This line is indented twelve spaces.
    The quick brown fox jum  *ped.
```

indent-relative-maybe Command

This command indents the current line like the previous nonblank line, by calling `indent-relative` with `t` as the *unindented-ok* argument. The return value is unpredictable.

If the previous nonblank line has no indent points beyond the current column, this command does nothing.

31.17.5 Adjustable "Tab Stops"

This section explains the mechanism for user-specified "tab stops" and the mechanisms that use and set them. The name "tab stops" is used because the feature is similar to that of the tab stops on a typewriter. The feature works by inserting an appropriate number of spaces and tab characters to reach the next tab stop column; it does not affect the display of tab characters in the buffer (see Section 38.13 [Usual Display], page 760). Note that the TAB character as input uses this tab stop feature only in a few major modes, such as Text mode.

tab-to-tab-stop Command

This command inserts spaces or tabs before point, up to the next tab stop column defined by `tab-stop-list`. It searches the list for an element greater than the current column number, and uses that element as the column to indent to. It does nothing if no such element is found.

tab-stop-list User Option

This variable is the list of tab stop columns used by `tab-to-tab-stops`. The elements should be integers in increasing order. The tab stop columns need not be evenly spaced.

Use *M-x edit-tab-stops* to edit the location of tab stops interactively.

31.17.6 Indentation-Based Motion Commands

These commands, primarily for interactive use, act based on the indentation in the text.

back-to-indentation Command

This command moves point to the first non-whitespace character in the current line (which is the line in which point is located). It returns `nil`.

backward-to-indentation *arg* Command

This command moves point backward *arg* lines and then to the first nonblank character on that line. It returns `nil`.

forward-to-indentation *arg* Command

This command moves point forward *arg* lines and then to the first nonblank character on that line. It returns `nil`.

31.18 Case Changes

The case change commands described here work on text in the current buffer. See Section 4.8 [Case Conversion], page 70, for case conversion functions that work on strings and characters. See Section 4.9 [Case Tables], page 72, for how to customize which characters are upper or lower case and how to convert them.

capitalize-region *start end* Command

This function capitalizes all words in the region defined by *start* and *end*. To capitalize means to convert each word's first character to upper case and convert the rest of each word to lower case. The function returns `nil`.

If one end of the region is in the middle of a word, the part of the word within the region is treated as an entire word.

When `capitalize-region` is called interactively, *start* and *end* are point and the mark, with the smallest first.

```
---------- Buffer: foo ----------
This is the contents of the 5th foo.
---------- Buffer: foo ----------

(capitalize-region 1 44)
⇒ nil

---------- Buffer: foo ----------
This Is The Contents Of The 5th Foo.
---------- Buffer: foo ----------
```

downcase-region *start end* Command

This function converts all of the letters in the region defined by *start* and *end* to lower case. The function returns `nil`.

When `downcase-region` is called interactively, *start* and *end* are point and the mark, with the smallest first.

upcase-region *start end* Command

This function converts all of the letters in the region defined by *start* and *end* to upper case. The function returns `nil`.

When `upcase-region` is called interactively, *start* and *end* are point and the mark, with the smallest first.

capitalize-word *count* Command

This function capitalizes *count* words after point, moving point over as it does. To capitalize means to convert each word's first character to upper case and convert the rest of each word to lower case. If *count* is negative,

the function capitalizes the *−count* previous words but does not move point. The value is `nil`.

If point is in the middle of a word, the part of the word before point is ignored when moving forward. The rest is treated as an entire word.

When `capitalize-word` is called interactively, *count* is set to the numeric prefix argument.

downcase-word *count* Command

This function converts the *count* words after point to all lower case, moving point over as it does. If *count* is negative, it converts the *−count* previous words but does not move point. The value is `nil`.

When `downcase-word` is called interactively, *count* is set to the numeric prefix argument.

upcase-word *count* Command

This function converts the *count* words after point to all upper case, moving point over as it does. If *count* is negative, it converts the *−count* previous words but does not move point. The value is `nil`.

When `upcase-word` is called interactively, *count* is set to the numeric prefix argument.

31.19 Text Properties

Each character position in a buffer or a string can have a *text property list*, much like the property list of a symbol (see Section 7.4 [Property Lists], page 114). The properties belong to a particular character at a particular place, such as, the letter '`T`' at the beginning of this sentence or the first '`o`' in '`foo`'—if the same character occurs in two different places, the two occurrences generally have different properties.

Each property has a name and a value. Both of these can be any Lisp object, but the name is normally a symbol. The usual way to access the property list is to specify a name and ask what value corresponds to it.

If a character has a `category` property, we call it the *category* of the character. It should be a symbol. The properties of the symbol serve as defaults for the properties of the character.

Copying text between strings and buffers preserves the properties along with the characters; this includes such diverse functions as `substring`, `insert`, and `buffer-substring`.

31.19.1 Examining Text Properties

The simplest way to examine text properties is to ask for the value of a particular property of a particular character. For that, use `get-text-property`. Use `text-properties-at` to get the entire property list of a

character. See Section 31.19.3 [Property Search], page 613, for functions to examine the properties of a number of characters at once.

These functions handle both strings and buffers. Keep in mind that positions in a string start from 0, whereas positions in a buffer start from 1.

get-text-property *pos prop* &optional *object* Function

This function returns the value of the *prop* property of the character after position *pos* in *object* (a buffer or string). The argument *object* is optional and defaults to the current buffer.

If there is no *prop* property strictly speaking, but the character has a category that is a symbol, then `get-text-property` returns the *prop* property of that symbol.

get-char-property *pos prop* &optional *object* Function

This function is like `get-text-property`, except that it checks overlays first and then text properties. See Section 38.8 [Overlays], page 748.

The argument *object* may be a string, a buffer, or a window. If it is a window, then the buffer displayed in that window is used for text properties and overlays, but only the overlays active for that window are considered. If *object* is a buffer, then all overlays in that buffer are considered, as well as text properties. If *object* is a string, only text properties are considered, since strings never have overlays.

text-properties-at *position* &optional *object* Function

This function returns the entire property list of the character at *position* in the string or buffer *object*. If *object* is `nil`, it defaults to the current buffer.

default-text-properties Variable

This variable holds a property list giving default values for text properties. Whenever a character does not specify a value for a property, neither directly nor through a category symbol, the value stored in this list is used instead. Here is an example:

```
(setq default-text-properties '(foo 69))
;; Make sure character 1 has no properties of its own.
(set-text-properties 1 2 nil)
;; What we get, when we ask, is the default value.
(get-text-property 1 'foo)
     ⇒ 69
```

31.19.2 Changing Text Properties

The primitives for changing properties apply to a specified range of text in a buffer or string. The function `set-text-properties` (see end of section)

sets the entire property list of the text in that range; more often, it is useful to add, change, or delete just certain properties specified by name.

Since text properties are considered part of the contents of the buffer (or string), and can affect how a buffer looks on the screen, any change in buffer text properties mark the buffer as modified. Buffer text property changes are undoable also (see Section 31.9 [Undo], page 590).

put-text-property *start end prop value* &optional *object* Function

This function sets the *prop* property to *value* for the text between *start* and *end* in the string or buffer *object*. If *object* is `nil`, it defaults to the current buffer.

add-text-properties *start end props* &optional *object* Function

This function adds or overrides text properties for the text between *start* and *end* in the string or buffer *object*. If *object* is `nil`, it defaults to the current buffer.

The argument *props* specifies which properties to add. It should have the form of a property list (see Section 7.4 [Property Lists], page 114): a list whose elements include the property names followed alternately by the corresponding values.

The return value is `t` if the function actually changed some property's value; `nil` otherwise (if *props* is `nil` or its values agree with those in the text).

For example, here is how to set the `comment` and `face` properties of a range of text:

```
(add-text-properties start end
                     '(comment t face highlight))
```

remove-text-properties *start end props* &optional *object* Function

This function deletes specified text properties from the text between *start* and *end* in the string or buffer *object*. If *object* is `nil`, it defaults to the current buffer.

The argument *props* specifies which properties to delete. It should have the form of a property list (see Section 7.4 [Property Lists], page 114): a list whose elements are property names alternating with corresponding values. But only the names matter—the values that accompany them are ignored. For example, here's how to remove the `face` property.

```
(remove-text-properties start end '(face nil))
```

The return value is `t` if the function actually changed some property's value; `nil` otherwise (if *props* is `nil` or if no character in the specified text had any of those properties).

To remove all text properties from certain text, use `set-text-properties` and specify `nil` for the new property list.

set-text-properties *start end props* &optional *object* Function

This function completely replaces the text property list for the text between *start* and *end* in the string or buffer *object*. If *object* is `nil`, it defaults to the current buffer.

The argument *props* is the new property list. It should be a list whose elements are property names alternating with corresponding values.

After `set-text-properties` returns, all the characters in the specified range have identical properties.

If *props* is `nil`, the effect is to get rid of all properties from the specified range of text. Here's an example:

```
(set-text-properties start end nil)
```

See also the function `buffer-substring-no-properties` (see Section 31.2 [Buffer Contents], page 576) which copies text from the buffer but does not copy its properties.

31.19.3 Text Property Search Functions

In typical use of text properties, most of the time several or many consecutive characters have the same value for a property. Rather than writing your programs to examine characters one by one, it is much faster to process chunks of text that have the same property value.

Here are functions you can use to do this. They use `eq` for comparing property values. In all cases, *object* defaults to the current buffer.

For high performance, it's very important to use the *limit* argument to these functions, especially the ones that search for a single property—otherwise, they may spend a long time scanning to the end of the buffer, if the property you are interested in does not change.

These functions do not move point; instead, they return a position (or `nil`). Remember that a position is always between two characters; the position returned by these functions is between two characters with different properties.

next-property-change *pos* &optional *object limit* Function

The function scans the text forward from position *pos* in the string or buffer *object* till it finds a change in some text property, then returns the position of the change. In other words, it returns the position of the first character beyond *pos* whose properties are not identical to those of the character just after *pos*.

If *limit* is non-`nil`, then the scan ends at position *limit*. If there is no property change before that point, `next-property-change` returns *limit*.

The value is `nil` if the properties remain unchanged all the way to the end of *object* and *limit* is `nil`. If the value is non-`nil`, it is a position greater than or equal to *pos*. The value equals *pos* only when *limit* equals *pos*.

Here is an example of how to scan the buffer by chunks of text within which all properties are constant:

```
(while (not (eobp))
  (let ((plist (text-properties-at (point)))
        (next-change
         (or (next-property-change (point) (current-buffer))
             (point-max))))
    Process text from point to next-change...
    (goto-char next-change)))
```

next-single-property-change *pos prop* &optional *object limit* Function

The function scans the text forward from position *pos* in the string or buffer *object* till it finds a change in the *prop* property, then returns the position of the change. In other words, it returns the position of the first character beyond *pos* whose *prop* property differs from that of the character just after *pos*.

If *limit* is non-`nil`, then the scan ends at position *limit*. If there is no property change before that point, `next-single-property-change` returns *limit*.

The value is `nil` if the property remains unchanged all the way to the end of *object* and *limit* is `nil`. If the value is non-`nil`, it is a position greater than or equal to *pos*; it equals *pos* only if *limit* equals *pos*.

previous-property-change *pos* &optional *object limit* Function

This is like `next-property-change`, but scans back from *pos* instead of forward. If the value is non-`nil`, it is a position less than or equal to *pos*; it equals *pos* only if *limit* equals *pos*.

previous-single-property-change *pos prop* &optional *object limit* Function

This is like `next-single-property-change`, but scans back from *pos* instead of forward. If the value is non-`nil`, it is a position less than or equal to *pos*; it equals *pos* only if *limit* equals *pos*.

next-char-property-change *position* &optional *limit* Function

This is like `next-property-change` except that it considers overlay properties as well as text properties. There is no *object* operand because this function operates only on the current buffer. It returns the next address at which either kind of property changes.

previous-char-property-change *position* &optional *limit* Function

This is like `next-char-property-change`, but scans back from *position* instead of forward.

text-property-any *start end prop value* &optional *object* Function

This function returns non-`nil` if at least one character between *start* and *end* has a property *prop* whose value is *value*. More precisely, it returns the position of the first such character. Otherwise, it returns `nil`.

The optional fifth argument, *object*, specifies the string or buffer to scan. Positions are relative to *object*. The default for *object* is the current buffer.

text-property-not-all *start end prop value* &optional *object* Function

This function returns non-`nil` if at least one character between *start* and *end* does not have a property *prop* with value *value*. More precisely, it returns the position of the first such character. Otherwise, it returns `nil`.

The optional fifth argument, *object*, specifies the string or buffer to scan. Positions are relative to *object*. The default for *object* is the current buffer.

31.19.4 Properties with Special Meanings

Here is a table of text property names that have special built-in meanings. The following sections list a few additional special property names that control filling and property inheritance. All other names have no standard meaning, and you can use them as you like.

`category` If a character has a `category` property, we call it the *category* of the character. It should be a symbol. The properties of the symbol serve as defaults for the properties of the character.

`face` You can use the property `face` to control the font and color of text. Its value is a face name or a list of face names. See Section 38.10 [Faces], page 753, for more information.

If the property value is a list, elements may also have the form `(foreground-color .` *color-name*`)` or `(background-color .` *color-name*`)`. These elements specify just the foreground color or just the background color; therefore, there is no need to create a face for each color that you want to use.

See Section 22.5 [Font Lock Mode], page 414, for information on how to update `face` properties automatically based on the contents of the text.

`mouse-face`

The property `mouse-face` is used instead of `face` when the mouse is on or near the character. For this purpose, "near" means that all text between the character and where the mouse is have the same `mouse-face` property value.

`local-map`
: You can specify a different keymap for some of the text in a buffer by means of the `local-map` property. The property's value for the character after point, if non-`nil`, is used for key lookup instead of the buffer's local map. If the property value is a symbol, the symbol's function definition is used as the keymap. See Section 21.6 [Active Keymaps], page 367.

`syntax-table`
: The `syntax-table` property overrides what the syntax table says about this particular character. See Section 34.4 [Syntax Properties], page 676.

`read-only`
: If a character has the property `read-only`, then modifying that character is not allowed. Any command that would do so gets an error.

  Insertion next to a read-only character is an error if inserting ordinary text there would inherit the `read-only` property due to stickiness. Thus, you can control permission to insert next to read-only text by controlling the stickiness. See Section 31.19.6 [Sticky Properties], page 618.

  Since changing properties counts as modifying the buffer, it is not possible to remove a `read-only` property unless you know the special trick: bind `inhibit-read-only` to a non-`nil` value and then remove the property. See Section 26.7 [Read Only Buffers], page 487.

`invisible`
: A non-`nil` `invisible` property can make a character invisible on the screen. See Section 38.4 [Invisible Text], page 742, for details.

`intangible`
: If a group of consecutive characters have equal and non-`nil` `intangible` properties, then you cannot place point between them. If you try to move point forward into the group, point actually moves to the end of the group. If you try to move point backward into the group, point actually moves to the start of the group.

  When the variable `inhibit-point-motion-hooks` is non-`nil`, the `intangible` property is ignored.

`modification-hooks`
: If a character has the property `modification-hooks`, then its value should be a list of functions; modifying that character calls all of those functions. Each function receives two arguments: the beginning and end of the part of the buffer being modified.

Note that if a particular modification hook function appears on several characters being modified by a single primitive, you can't predict how many times the function will be called.

`insert-in-front-hooks`
`insert-behind-hooks`
: The operation of inserting text in a buffer also calls the functions listed in the `insert-in-front-hooks` property of the following character and in the `insert-behind-hooks` property of the preceding character. These functions receive two arguments, the beginning and end of the inserted text. The functions are called *after* the actual insertion takes place.

  See also Section 31.23 [Change Hooks], page 626, for other hooks that are called when you change text in a buffer.

`point-entered`
`point-left`
: The special properties `point-entered` and `point-left` record hook functions that report motion of point. Each time point moves, Emacs compares these two property values:

  - the `point-left` property of the character after the old location, and
  - the `point-entered` property of the character after the new location.

  If these two values differ, each of them is called (if not `nil`) with two arguments: the old value of point, and the new one.

  The same comparison is made for the characters before the old and new locations. The result may be to execute two `point-left` functions (which may be the same function) and/or two `point-entered` functions (which may be the same function). In any case, all the `point-left` functions are called first, followed by all the `point-entered` functions.

  It is possible using `char-after` to examine characters at various positions without moving point to those positions. Only an actual change in the value of point runs these hook functions.

inhibit-point-motion-hooks Variable

When this variable is non-`nil`, `point-left` and `point-entered` hooks are not run, and the `intangible` property has no effect. Do not set this variable globally; bind it with `let`.

31.19.5 Formatted Text Properties

These text properties affect the behavior of the fill commands. They are used for representing formatted text. See Section 31.11 [Filling], page 593, and Section 31.12 [Margins], page 596.

`hard` If a newline character has this property, it is a "hard" newline. The fill commands do not alter hard newlines and do not move words across them. However, this property takes effect only if the variable `use-hard-newlines` is non-`nil`.

`right-margin`
This property specifies an extra right margin for filling this part of the text.

`left-margin`
This property specifies an extra left margin for filling this part of the text.

`justification`
This property specifies the style of justification for filling this part of the text.

31.19.6 Stickiness of Text Properties

Self-inserting characters normally take on the same properties as the preceding character. This is called *inheritance* of properties.

In a Lisp program, you can do insertion with inheritance or without, depending on your choice of insertion primitive. The ordinary text insertion functions such as `insert` do not inherit any properties. They insert text with precisely the properties of the string being inserted, and no others. This is correct for programs that copy text from one context to another—for example, into or out of the kill ring. To insert with inheritance, use the special primitives described in this section. Self-inserting characters inherit properties because they work using these primitives.

When you do insertion with inheritance, *which* properties are inherited depends on two specific properties: `front-sticky` and `rear-nonsticky`.

Insertion after a character inherits those of its properties that are *rear-sticky*. Insertion before a character inherits those of its properties that are *front-sticky*. By default, a text property is rear-sticky but not front-sticky. Thus, the default is to inherit all the properties of the preceding character, and nothing from the following character. You can request different behavior by specifying the stickiness of certain properties.

If a character's `front-sticky` property is `t`, then all its properties are front-sticky. If the `front-sticky` property is a list, then the sticky properties of the character are those whose names are in the list. For example, if a character has a `front-sticky` property whose value is `(face read-only)`, then insertion before the character can inherit its `face` property and its `read-only` property, but no others.

The `rear-nonsticky` works the opposite way. Every property is rear-sticky by default, so the `rear-nonsticky` property says which properties are *not* rear-sticky. If a character's `rear-nonsticky` property is `t`, then

none of its properties are rear-sticky. If the `rear-nonsticky` property is a list, properties are rear-sticky *unless* their names are in the list.

When you insert text with inheritance, it inherits all the rear-sticky properties of the preceding character, and all the front-sticky properties of the following character. The previous character's properties take precedence when both sides offer different sticky values for the same property.

Here are the functions that insert text with inheritance of properties:

insert-and-inherit &rest *strings* Function

Insert the strings *strings*, just like the function `insert`, but inherit any sticky properties from the adjoining text.

insert-before-markers-and-inherit &rest *strings* Function

Insert the strings *strings*, just like the function `insert-before-markers`, but inherit any sticky properties from the adjoining text.

See Section 31.4 [Insertion], page 578, for the ordinary insertion functions which do not inherit.

31.19.7 Saving Text Properties in Files

You can save text properties in files (along with the text itself), and restore the same text properties when visiting or inserting the files, using these two hooks:

write-region-annotate-functions Variable

This variable's value is a list of functions for `write-region` to run to encode text properties in some fashion as annotations to the text being written in the file. See Section 24.4 [Writing to Files], page 440.

Each function in the list is called with two arguments: the start and end of the region to be written. These functions should not alter the contents of the buffer. Instead, they should return lists indicating annotations to write in the file in addition to the text in the buffer.

Each function should return a list of elements of the form (*position* . *string*), where *position* is an integer specifying the relative position within the text to be written, and *string* is the annotation to add there.

Each list returned by one of these functions must be already sorted in increasing order by *position*. If there is more than one function, `write-region` merges the lists destructively into one sorted list.

When `write-region` actually writes the text from the buffer to the file, it intermixes the specified annotations at the corresponding positions. All this takes place without modifying the buffer.

after-insert-file-functions Variable

This variable holds a list of functions for `insert-file-contents` to call after inserting a file's contents. These functions should scan the inserted text for annotations, and convert them to the text properties they stand for.

Each function receives one argument, the length of the inserted text; point indicates the start of that text. The function should scan that text for annotations, delete them, and create the text properties that the annotations specify. The function should return the updated length of the inserted text, as it stands after those changes. The value returned by one function becomes the argument to the next function.

These functions should always return with point at the beginning of the inserted text.

The intended use of `after-insert-file-functions` is for converting some sort of textual annotations into actual text properties. But other uses may be possible.

We invite users to write Lisp programs to store and retrieve text properties in files, using these hooks, and thus to experiment with various data formats and find good ones. Eventually we hope users will produce good, general extensions we can install in Emacs.

We suggest not trying to handle arbitrary Lisp objects as text property names or values—because a program that general is probably difficult to write, and slow. Instead, choose a set of possible data types that are reasonably flexible, and not too hard to encode.

See Section 24.12 [Format Conversion], page 463, for a related feature.

31.19.8 Lazy Computation of Text Properties

Instead of computing text properties for all the text in the buffer, you can arrange to compute the text properties for parts of the text when and if something depends on them.

The primitive that extracts text from the buffer along with its properties is `buffer-substring`. Before examining the properties, this function runs the abnormal hook `buffer-access-fontify-functions`.

buffer-access-fontify-functions Variable

This variable holds a list of functions for computing text properties. Before `buffer-substring` copies the text and text properties for a portion of the buffer, it calls all the functions in this list. Each of the functions receives two arguments that specify the range of the buffer being accessed. (The buffer itself is always the current buffer.)

The function `buffer-substring-no-properties` does not call these functions, since it ignores text properties anyway.

In order to prevent the hook functions from being called more than once for the same part of the buffer, you can use the variable `buffer-access-fontified-property`.

buffer-access-fontified-property Variable

If this value's variable is non-`nil`, it is a symbol which is used as a text property name. A non-`nil` value for that text property means, "the other text properties for this character have already been computed."

If all the characters in the range specified for `buffer-substring` have a non-`nil` value for this property, `buffer-substring` does not call the `buffer-access-fontify-functions` functions. It assumes these characters already have the right text properties, and just copies the properties they already have.

The normal way to use this feature is that the `buffer-access-fontify-functions` functions add this property, as well as others, to the characters they operate on. That way, they avoid being called over and over for the same text.

31.19.9 Defining Clickable Text

There are two ways to set up *clickable text* in a buffer. There are typically two parts of this: to make the text highlight when the mouse is over it, and to make a mouse button do something when you click it on that part of the text.

Highlighting is done with the `mouse-face` text property. Here is an example of how Dired does it:

```
(condition-case nil
    (if (dired-move-to-filename)
        (put-text-property (point)
                           (save-excursion
                             (dired-move-to-end-of-filename)
                             (point))
                           'mouse-face 'highlight))
  (error nil))
```

The first two arguments to `put-text-property` specify the beginning and end of the text.

The usual way to make the mouse do something when you click it on this text is to define `mouse-2` in the major mode's keymap. The job of checking whether the click was on clickable text is done by the command definition. Here is how Dired does it:

```
(defun dired-mouse-find-file-other-window (event)
  "In dired, visit the file or directory name you click on."
  (interactive "e")
  (let (file)
```

```
(save-excursion
  (set-buffer (window-buffer (posn-window (event-end event))))
  (save-excursion
    (goto-char (posn-point (event-end event)))
    (setq file (dired-get-filename))))
(select-window (posn-window (event-end event)))
(find-file-other-window (file-name-sans-versions file t))))
```

The reason for the outer `save-excursion` construct is to avoid changing the current buffer; the reason for the inner one is to avoid permanently altering point in the buffer you click on. In this case, Dired uses the function `dired-get-filename` to determine which file to visit, based on the position found in the event.

Instead of defining a mouse command for the major mode, you can define a key binding for the clickable text itself, using the `local-map` text property:

```
(let ((map (make-sparse-keymap)))
  (define-key-binding map [mouse-2] 'operate-this-button)
  (put-text-property (point)
                     (save-excursion
                       (dired-move-to-end-of-filename)
                       (point))
                     'local-map map))
```

This method makes it possible to define different commands for various clickable pieces of text. Also, the major mode definition (or the global definition) remains available for the rest of the text in the buffer.

31.19.10 Why Text Properties are not Intervals

Some editors that support adding attributes to text in the buffer do so by letting the user specify "intervals" within the text, and adding the properties to the intervals. Those editors permit the user or the programmer to determine where individual intervals start and end. We deliberately provided a different sort of interface in Emacs Lisp to avoid certain paradoxical behavior associated with text modification.

If the actual subdivision into intervals is meaningful, that means you can distinguish between a buffer that is just one interval with a certain property, and a buffer containing the same text subdivided into two intervals, both of which have that property.

Suppose you take the buffer with just one interval and kill part of the text. The text remaining in the buffer is one interval, and the copy in the kill ring (and the undo list) becomes a separate interval. Then if you yank back the killed text, you get two intervals with the same properties. Thus, editing does not preserve the distinction between one interval and two.

Suppose we "fix" this problem by coalescing the two intervals when the text is inserted. That works fine if the buffer originally was a single interval.

But suppose instead that we have two adjacent intervals with the same properties, and we kill the text of one interval and yank it back. The same interval-coalescence feature that rescues the other case causes trouble in this one: after yanking, we have just one interval. One again, editing does not preserve the distinction between one interval and two.

Insertion of text at the border between intervals also raises questions that have no satisfactory answer.

However, it is easy to arrange for editing to behave consistently for questions of the form, "What are the properties of this character?" So we have decided these are the only questions that make sense; we have not implemented asking questions about where intervals start or end.

In practice, you can usually use the text property search functions in place of explicit interval boundaries. You can think of them as finding the boundaries of intervals, assuming that intervals are always coalesced whenever possible. See Section 31.19.3 [Property Search], page 613.

Emacs also provides explicit intervals as a presentation feature; see Section 38.8 [Overlays], page 748.

31.20 Substituting for a Character Code

The following functions replace characters within a specified region based on their character codes.

subst-char-in-region *start end old-char new-char* &optional *noundo* Function

This function replaces all occurrences of the character *old-char* with the character *new-char* in the region of the current buffer defined by *start* and *end*.

If *noundo* is non-`nil`, then `subst-char-in-region` does not record the change for undo and does not mark the buffer as modified. This feature is used for controlling selective display (see Section 38.5 [Selective Display], page 744).

`subst-char-in-region` does not move point and returns `nil`.

```
---------- Buffer: foo ----------
This is the contents of the buffer before.
---------- Buffer: foo ----------

(subst-char-in-region 1 20 ?i ?X)
     ⇒ nil

---------- Buffer: foo ----------
ThXs Xs the contents of the buffer before.
---------- Buffer: foo ----------
```

translate-region *start end table* Function

This function applies a translation table to the characters in the buffer between positions *start* and *end*.

The translation table *table* is a string; `(aref` *table ochar*`)` gives the translated character corresponding to *ochar*. If the length of *table* is less than 256, any characters with codes larger than the length of *table* are not altered by the translation.

The return value of `translate-region` is the number of characters that were actually changed by the translation. This does not count characters that were mapped into themselves in the translation table.

31.21 Registers

A register is a sort of variable used in Emacs editing that can hold a variety of different kinds of values. Each register is named by a single character. All ASCII characters and their meta variants (but with the exception of `C-g`) can be used to name registers. Thus, there are 255 possible registers. A register is designated in Emacs Lisp by the character that is its name.

register-alist Variable

This variable is an alist of elements of the form `(`*name* `.` *contents*`)`. Normally, there is one element for each Emacs register that has been used.

The object *name* is a character (an integer) identifying the register.

The *contents* of a register can have several possible types:

a number
: A number stands for itself. If `insert-register` finds a number in the register, it converts the number to decimal.

a marker
: A marker represents a buffer position to jump to.

a string
: A string is text saved in the register.

a rectangle
: A rectangle is represented by a list of strings.

`(`*window-configuration position*`)`
: This represents a window configuration to restore in one frame, and a position to jump to in the current buffer.

`(`*frame-configuration position*`)`
: This represents a frame configuration to restore, and a position to jump to in the current buffer.

(file *filename*)
: This represents a file to visit; jumping to this value visits file *filename*.

(file-query *filename position*)
: This represents a file to visit and a position in it; jumping to this value visits file *filename* and goes to buffer position *position*. Restoring this type of position asks the user for confirmation first.

The functions in this section return unpredictable values unless otherwise stated.

get-register *reg* Function
: This function returns the contents of the register *reg*, or `nil` if it has no contents.

set-register *reg value* Function
: This function sets the contents of register *reg* to *value*. A register can be set to any value, but the other register functions expect only certain data types. The return value is *value*.

view-register *reg* Command
: This command displays what is contained in register *reg*.

insert-register *reg* &optional *beforep* Command
: This command inserts contents of register *reg* into the current buffer.

  Normally, this command puts point before the inserted text, and the mark after it. However, if the optional second argument *beforep* is non-`nil`, it puts the mark before and point after. You can pass a non-`nil` second argument *beforep* to this function interactively by supplying any prefix argument.

  If the register contains a rectangle, then the rectangle is inserted with its upper left corner at point. This means that text is inserted in the current line and underneath it on successive lines.

  If the register contains something other than saved text (a string) or a rectangle (a list), currently useless things happen. This may be changed in the future.

31.22 Transposition of Text

This subroutine is used by the transposition commands.

transpose-regions *start1 end1 start2 end2* &optional *leave-markers* Function
: This function exchanges two nonoverlapping portions of the buffer. Arguments *start1* and *end1* specify the bounds of one portion and arguments *start2* and *end2* specify the bounds of the other portion.

Normally, `transpose-regions` relocates markers with the transposed text; a marker previously positioned within one of the two transposed portions moves along with that portion, thus remaining between the same two characters in their new position. However, if *leave-markers* is non-`nil`, `transpose-regions` does not do this—it leaves all markers unrelocated.

31.23 Change Hooks

These hook variables let you arrange to take notice of all changes in all buffers (or in a particular buffer, if you make them buffer-local). See also Section 31.19.4 [Special Properties], page 615, for how to detect changes to specific parts of the text.

The functions you use in these hooks should save and restore the match data if they do anything that uses regular expressions; otherwise, they will interfere in bizarre ways with the editing operations that call them.

before-change-functions Variable

This variable holds a list of functions to call before any buffer modification. Each function gets two arguments, the beginning and end of the region that is about to change, represented as integers. The buffer that is about to change is always the current buffer.

after-change-functions Variable

This variable holds a list of functions to call after any buffer modification. Each function receives three arguments: the beginning and end of the region just changed, and the length of the text that existed before the change. All three arguments are integers. The buffer that's about to change is always the current buffer.

The length of the old text is the difference between the buffer positions before and after that text as it was before the change. As for the changed text, its length is simply the difference between the first two arguments.

combine-after-change-calls *body...* Macro

The macro executes *body* normally, but arranges to call the after-change functions just once for a series of several changes—if that seems safe.

If a program makes several text changes in the same area of the buffer, using the macro `combine-after-change-calls` around that part of the program can make it run considerably faster when after-change hooks are in use. When the after-change hooks are ultimately called, the arguments specify a portion of the buffer including all of the changes made within the `combine-after-change-calls` body.

Warning: You must not alter the values of `after-change-functions` and `after-change-function` within the body of a `combine-after-change-calls` form.

Note: If the changes you combine occur in widely scattered parts of the buffer, this will still work, but it is not advisable, because it may lead to inefficient behavior for some change hook functions.

before-change-function Variable

This obsolete variable holds one function to call before any buffer modification (or `nil` for no function). It is called just like the functions in `before-change-functions`.

after-change-function Variable

This obsolete variable holds one function to call after any buffer modification (or `nil` for no function). It is called just like the functions in `after-change-functions`.

The four variables above are temporarily bound to `nil` during the time that any of these functions is running. This means that if one of these functions changes the buffer, that change won't run these functions. If you do want a hook function to make changes that run these functions, make it bind these variables back to their usual values.

One inconvenient result of this protective feature is that you cannot have a function in `after-change-functions` or `before-change-functions` which changes the value of that variable. But that's not a real limitation. If you want those functions to change the list of functions to run, simply add one fixed function to the hook, and code that function to look in another variable for other functions to call. Here is an example:

```
(setq my-own-after-change-functions nil)
(defun indirect-after-change-function (beg end len)
  (let ((list my-own-after-change-functions))
    (while list
      (funcall (car list) beg end len)
      (setq list (cdr list)))))

(add-hooks 'after-change-functions
           'indirect-after-change-function)
```

first-change-hook Variable

This variable is a normal hook that is run whenever a buffer is changed that was previously in the unmodified state.

32 Non-ASCII Characters

This chapter covers the special issues relating to non-ASCII characters and how they are stored in strings and buffers.

32.1 Text Representations

Emacs has two *text representations*—two ways to represent text in a string or buffer. These are called *unibyte* and *multibyte*. Each string, and each buffer, uses one of these two representations. For most purposes, you can ignore the issue of representations, because Emacs converts text between them as appropriate. Occasionally in Lisp programming you will need to pay attention to the difference.

In unibyte representation, each character occupies one byte and therefore the possible character codes range from 0 to 255. Codes 0 through 127 are ASCII characters; the codes from 128 through 255 are used for one non-ASCII character set (you can choose which character set by setting the variable `nonascii-insert-offset`).

In multibyte representation, a character may occupy more than one byte, and as a result, the full range of Emacs character codes can be stored. The first byte of a multibyte character is always in the range 128 through 159 (octal 0200 through 0237). These values are called *leading codes*. The second and subsequent bytes of a multibyte character are always in the range 160 through 255 (octal 0240 through 0377); these values are *trailing codes*.

In a buffer, the buffer-local value of the variable `enable-multibyte-characters` specifies the representation used. The representation for a string is determined based on the string contents when the string is constructed.

enable-multibyte-characters Variable

This variable specifies the current buffer's text representation. If it is non-`nil`, the buffer contains multibyte text; otherwise, it contains unibyte text.

You cannot set this variable directly; instead, use the function `set-buffer-multibyte` to change a buffer's representation.

default-enable-multibyte-characters Variable

This variable's value is entirely equivalent to `(default-value 'enable-multibyte-characters)`, and setting this variable changes that default value. Setting the local binding of `enable-multibyte-characters` in a specific buffer is not allowed, but changing the default value is supported, and it is a reasonable thing to do, because it has no effect on existing buffers.

The '`--unibyte`' command line option does its job by setting the default value to `nil` early in startup.

multibyte-string-p *string* Function
Return `t` if *string* contains multibyte characters.

32.2 Converting Text Representations

Emacs can convert unibyte text to multibyte; it can also convert multibyte text to unibyte, though this conversion loses information. In general these conversions happen when inserting text into a buffer, or when putting text from several strings together in one string. You can also explicitly convert a string's contents to either representation.

Emacs chooses the representation for a string based on the text that it is constructed from. The general rule is to convert unibyte text to multibyte text when combining it with other multibyte text, because the multibyte representation is more general and can hold whatever characters the unibyte text has.

When inserting text into a buffer, Emacs converts the text to the buffer's representation, as specified by `enable-multibyte-characters` in that buffer. In particular, when you insert multibyte text into a unibyte buffer, Emacs converts the text to unibyte, even though this conversion cannot in general preserve all the characters that might be in the multibyte text. The other natural alternative, to convert the buffer contents to multibyte, is not acceptable because the buffer's representation is a choice made by the user that cannot be overridden automatically.

Converting unibyte text to multibyte text leaves ASCII characters unchanged, and likewise 128 through 159. It converts the non-ASCII codes 160 through 255 by adding the value `nonascii-insert-offset` to each character code. By setting this variable, you specify which character set the unibyte characters correspond to (see Section 32.5 [Character Sets], page 632). For example, if `nonascii-insert-offset` is 2048, which is `(- (make-char 'latin-iso8859-1) 128)`, then the unibyte non-ASCII characters correspond to Latin 1. If it is 2688, which is `(- (make-char 'greek-iso8859-7) 128)`, then they correspond to Greek letters.

Converting multibyte text to unibyte is simpler: it performs logical-and of each character code with 255. If `nonascii-insert-offset` has a reasonable value, corresponding to the beginning of some character set, this conversion is the inverse of the other: converting unibyte text to multibyte and back to unibyte reproduces the original unibyte text.

nonascii-insert-offset Variable
This variable specifies the amount to add to a non-ASCII character when converting unibyte text to multibyte. It also applies when `self-insert-command` inserts a character in the unibyte non-ASCII range, 128 through 255. However, the function `insert-char` does not perform this conversion.

The right value to use to select character set *cs* is `(- (make-char` *cs*`) 128)`. If the value of `nonascii-insert-offset` is zero, then conversion actually uses the value for the Latin 1 character set, rather than zero.

nonascii-translation-table Variable

This variable provides a more general alternative to `nonascii-insert-offset`. You can use it to specify independently how to translate each code in the range of 128 through 255 into a multibyte character. The value should be a vector, or `nil`. If this is non-`nil`, it overrides `nonascii-insert-offset`.

string-make-unibyte *string* Function

This function converts the text of *string* to unibyte representation, if it isn't already, and returns the result. If *string* is a unibyte string, it is returned unchanged.

string-make-multibyte *string* Function

This function converts the text of *string* to multibyte representation, if it isn't already, and returns the result. If *string* is a multibyte string, it is returned unchanged.

32.3 Selecting a Representation

Sometimes it is useful to examine an existing buffer or string as multibyte when it was unibyte, or vice versa.

set-buffer-multibyte *multibyte* Function

Set the representation type of the current buffer. If *multibyte* is non-`nil`, the buffer becomes multibyte. If *multibyte* is `nil`, the buffer becomes unibyte.

This function leaves the buffer contents unchanged when viewed as a sequence of bytes. As a consequence, it can change the contents viewed as characters; a sequence of two bytes which is treated as one character in multibyte representation will count as two characters in unibyte representation.

This function sets `enable-multibyte-characters` to record which representation is in use. It also adjusts various data in the buffer (including overlays, text properties and markers) so that they cover the same text as they did before.

string-as-unibyte *string* Function

This function returns a string with the same bytes as *string* but treating each byte as a character. This means that the value may have more characters than *string* has.

If *string* is unibyte already, then the value is *string* itself.

string-as-multibyte *string* Function
This function returns a string with the same bytes as *string* but treating each multibyte sequence as one character. This means that the value may have fewer characters than *string* has.

If *string* is multibyte already, then the value is *string* itself.

32.4 Character Codes

The unibyte and multibyte text representations use different character codes. The valid character codes for unibyte representation range from 0 to 255—the values that can fit in one byte. The valid character codes for multibyte representation range from 0 to 524287, but not all values in that range are valid. In particular, the values 128 through 255 are not legitimate in multibyte text (though they can occur in "raw bytes"; see Section 32.10.7 [Explicit Encoding], page 643). Only the ASCII codes 0 through 127 are fully legitimate in both representations.

char-valid-p *charcode* Function
This returns `t` if *charcode* is valid for either one of the two text representations.

```
(char-valid-p 65)
     ⇒ t
(char-valid-p 256)
     ⇒ nil
(char-valid-p 2248)
     ⇒ t
```

32.5 Character Sets

Emacs classifies characters into various *character sets*, each of which has a name which is a symbol. Each character belongs to one and only one character set.

In general, there is one character set for each distinct script. For example, `latin-iso8859-1` is one character set, `greek-iso8859-7` is another, and `ascii` is another. An Emacs character set can hold at most 9025 characters; therefore, in some cases, characters that would logically be grouped together are split into several character sets. For example, one set of Chinese characters, generally known as Big 5, is divided into two Emacs character sets, `chinese-big5-1` and `chinese-big5-2`.

charsetp *object* Function
Return `t` if *object* is a character set name symbol, `nil` otherwise.

charset-list Function
This function returns a list of all defined character set names.

char-charset *character* Function
This function returns the name of the character set that *character* belongs to.

32.6 Characters and Bytes

In multibyte representation, each character occupies one or more bytes. Each character set has an *introduction sequence*, which is normally one or two bytes long. (Exception: the ASCII character set has a zero-length introduction sequence.) The introduction sequence is the beginning of the byte sequence for any character in the character set. The rest of the character's bytes distinguish it from the other characters in the same character set. Depending on the character set, there are either one or two distinguishing bytes; the number of such bytes is called the *dimension* of the character set.

charset-dimension *charset* Function
This function returns the dimension of *charset*; at present, the dimension is always 1 or 2.

This is the simplest way to determine the byte length of a character set's introduction sequence:

```
(- (char-bytes (make-char charset))
   (charset-dimension charset))
```

32.7 Splitting Characters

The functions in this section convert between characters and the byte values used to represent them. For most purposes, there is no need to be concerned with the sequence of bytes used to represent a character, because Emacs translates automatically when necessary.

char-bytes *character* Function
This function returns the number of bytes used to represent the character *character*. This depends only on the character set that *character* belongs to; it equals the dimension of that character set (see Section 32.5 [Character Sets], page 632), plus the length of its introduction sequence.

```
(char-bytes 2248)
     ⇒ 2
(char-bytes 65)
     ⇒ 1
(char-bytes 192)
     ⇒ 1
```

The reason this function can give correct results for both multibyte and unibyte representations is that the non-ASCII character codes used in those two representations do not overlap.

split-char *character* Function

Return a list containing the name of the character set of *character*, followed by one or two byte values (integers) which identify *character* within that character set. The number of byte values is the character set's dimension.

```
(split-char 2248)
     ⇒ (latin-iso8859-1 72)
(split-char 65)
     ⇒ (ascii 65)
```

Unibyte non-ASCII characters are considered as part of the `ascii` character set:

```
(split-char 192)
     ⇒ (ascii 192)
```

make-char *charset* &rest *byte-values* Function

This function returns the character in character set *charset* identified by *byte-values*. This is roughly the inverse of `split-char`. Normally, you should specify either one or two *byte-values*, according to the dimension of *charset*. For example,

```
(make-char 'latin-iso8859-1 72)
     ⇒ 2248
```

If you call `make-char` with no *byte-values*, the result is a *generic character* which stands for *charset*. A generic character is an integer, but it is *not* valid for insertion in the buffer as a character. It can be used in `char-table-range` to refer to the whole character set (see Section 6.6 [Char-Tables], page 104). `char-valid-p` returns `nil` for generic characters. For example:

```
(make-char 'latin-iso8859-1)
     ⇒ 2176
(char-valid-p 2176)
     ⇒ nil
(split-char 2176)
     ⇒ (latin-iso8859-1 0)
```

32.8 Scanning for Character Sets

Sometimes it is useful to find out which character sets appear in a part of a buffer or a string. One use for this is in determining which coding systems (see Section 32.10 [Coding Systems], page 636) are capable of representing all of the text in question.

find-charset-region *beg end* &optional *translation* Function

This function returns a list of the character sets that appear in the current buffer between positions *beg* and *end*.

The optional argument *translation* specifies a translation table to be used in scanning the text (see Section 32.9 [Translation of Characters], page 635). If it is non-`nil`, then each character in the region is translated through this table, and the value returned describes the translated characters instead of the characters actually in the buffer.

find-charset-string *string* &optional *translation* Function

This function returns a list of the character sets that appear in the string *string*.

The optional argument *translation* specifies a translation table; see `find-charset-region`, above.

32.9 Translation of Characters

A *translation table* specifies a mapping of characters into characters. These tables are used in encoding and decoding, and for other purposes. Some coding systems specify their own particular translation tables; there are also default translation tables which apply to all other coding systems.

make-translation-table *translations* Function

This function returns a translation table based on the arguments *translations*. Each argument—each element of *translations*—should be a list of the form (*from* . *to*); this says to translate the character *from* into *to*.

You can also map one whole character set into another character set with the same dimension. To do this, you specify a generic character (which designates a character set) for *from* (see Section 32.7 [Splitting Characters], page 633). In this case, *to* should also be a generic character, for another character set of the same dimension. Then the translation table translates each character of *from*'s character set into the corresponding character of *to*'s character set.

In decoding, the translation table's translations are applied to the characters that result from ordinary decoding. If a coding system has property `character-translation-table-for-decode`, that specifies the translation table to use. Otherwise, if `standard-character-translation-table-for-decode` is non-`nil`, decoding uses that table.

In encoding, the translation table's translations are applied to the characters in the buffer, and the result of translation is actually encoded. If a coding system has property `character-translation-table-for-encode`, that specifies the translation table to use. Otherwise the variable `standard-character-translation-table-for-encode` specifies the translation table.

standard-character-translation-table-for-decode Variable

This is the default translation table for decoding, for coding systems that don't specify any other translation table.

standard-character-translation-table-for-encode Variable
This is the default translation table for encoding, for coding systems that don't specify any other translation table.

32.10 Coding Systems

When Emacs reads or writes a file, and when Emacs sends text to a subprocess or receives text from a subprocess, it normally performs character code conversion and end-of-line conversion as specified by a particular *coding system*.

32.10.1 Basic Concepts of Coding Systems

Character code conversion involves conversion between the encoding used inside Emacs and some other encoding. Emacs supports many different encodings, in that it can convert to and from them. For example, it can convert text to or from encodings such as Latin 1, Latin 2, Latin 3, Latin 4, Latin 5, and several variants of ISO 2022. In some cases, Emacs supports several alternative encodings for the same characters; for example, there are three coding systems for the Cyrillic (Russian) alphabet: ISO, Alternativnyj, and KOI8.

Most coding systems specify a particular character code for conversion, but some of them leave this unspecified—to be chosen heuristically based on the data.

End of line conversion handles three different conventions used on various systems for representing end of line in files. The Unix convention is to use the linefeed character (also called newline). The DOS convention is to use the two character sequence, carriage-return linefeed, at the end of a line. The Mac convention is to use just carriage-return.

Base coding systems such as `latin-1` leave the end-of-line conversion unspecified, to be chosen based on the data. *Variant coding systems* such as `latin-1-unix`, `latin-1-dos` and `latin-1-mac` specify the end-of-line conversion explicitly as well. Most base coding systems have three corresponding variants whose names are formed by adding '`-unix`', '`-dos`' and '`-mac`'.

The coding system `raw-text` is special in that it prevents character code conversion, and causes the buffer visited with that coding system to be a unibyte buffer. It does not specify the end-of-line conversion, allowing that to be determined as usual by the data, and has the usual three variants which specify the end-of-line conversion. `no-conversion` is equivalent to `raw-text-unix`: it specifies no conversion of either character codes or end-of-line.

The coding system `emacs-mule` specifies that the data is represented in the internal Emacs encoding. This is like `raw-text` in that no code conversion happens, but different in that the result is multibyte data.

coding-system-get *coding-system property* Function

This function returns the specified property of the coding system *coding-system*. Most coding system properties exist for internal purposes, but one that you might find useful is `mime-charset`. That property's value is the name used in MIME for the character coding which this coding system can read and write. Examples:

```
(coding-system-get 'iso-latin-1 'mime-charset)
     ⇒ iso-8859-1
(coding-system-get 'iso-2022-cn 'mime-charset)
     ⇒ iso-2022-cn
(coding-system-get 'cyrillic-koi8 'mime-charset)
     ⇒ koi8-r
```

The value of the `mime-charset` property is also defined as an alias for the coding system.

32.10.2 Encoding and I/O

The principal purpose of coding systems is for use in reading and writing files. The function `insert-file-contents` uses a coding system for decoding the file data, and `write-region` uses one to encode the buffer contents.

You can specify the coding system to use either explicitly (see Section 32.10.6 [Specifying Coding Systems], page 642), or implicitly using the defaulting mechanism (see Section 32.10.5 [Default Coding Systems], page 640). But these methods may not completely specify what to do. For example, they may choose a coding system such as `undefined` which leaves the character code conversion to be determined from the data. In these cases, the I/O operation finishes the job of choosing a coding system. Very often you will want to find out afterwards which coding system was chosen.

buffer-file-coding-system Variable

This variable records the coding system that was used for visiting the current buffer. It is used for saving the buffer, and for writing part of the buffer with `write-region`. When those operations ask the user to specify a different coding system, `buffer-file-coding-system` is updated to the coding system specified.

save-buffer-coding-system Variable

This variable specifies the coding system for saving the buffer—but it is not used for `write-region`. When saving the buffer asks the user to specify a different coding system, and `save-buffer-coding-system` was used, then it is updated to the coding system that was specified.

last-coding-system-used Variable

I/O operations for files and subprocesses set this variable to the coding system name that was used. The explicit encoding and decoding functions (see Section 32.10.7 [Explicit Encoding], page 643) set it too.

Warning: Since receiving subprocess output sets this variable, it can change whenever Emacs waits; therefore, you should use copy the value shortly after the function call which stores the value you are interested in.

The variable `selection-coding-system` specifies how to encode selections for the window system. See Section 28.18 [Window System Selections], page 544.

32.10.3 Coding Systems in Lisp

Here are Lisp facilities for working with coding systems;

coding-system-list &optional *base-only* Function
This function returns a list of all coding system names (symbols). If *base-only* is non-`nil`, the value includes only the base coding systems. Otherwise, it includes variant coding systems as well.

coding-system-p *object* Function
This function returns `t` if *object* is a coding system name.

check-coding-system *coding-system* Function
This function checks the validity of *coding-system*. If that is valid, it returns *coding-system*. Otherwise it signals an error with condition `coding-system-error`.

coding-system-change-eol-conversion *coding-system* *eol-type* Function
This function returns a coding system which is like *coding-system* except for its eol conversion, which is specified by `eol-type`. *eol-type* should be `unix`, `dos`, `mac`, or `nil`. If it is `nil`, the returned coding system determines the end-of-line conversion from the data.

coding-system-change-text-conversion *eol-coding* *text-coding* Function
This function returns a coding system which uses the end-of-line conversion of *eol-coding*, and the text conversion of *text-coding*. If *text-coding* is `nil`, it returns `undecided`, or one of its variants according to *eol-coding*.

find-coding-systems-region *from to* Function
This function returns a list of coding systems that could be used to encode a text between *from* and *to*. All coding systems in the list can safely encode any multibyte characters in that portion of the text.

If the text contains no multibyte characters, the function returns the list `(undecided)`.

find-coding-systems-string *string* Function
This function returns a list of coding systems that could be used to encode the text of *string*. All coding systems in the list can safely encode any multibyte characters in *string*. If the text contains no multibyte characters, this returns the list `(undecided)`.

find-coding-systems-for-charsets *charsets* Function
This function returns a list of coding systems that could be used to encode all the character sets in the list *charsets*.

detect-coding-region *start end* &optional *highest* Function
This function chooses a plausible coding system for decoding the text from *start* to *end*. This text should be "raw bytes" (see Section 32.10.7 [Explicit Encoding], page 643).

Normally this function returns a list of coding systems that could handle decoding the text that was scanned. They are listed in order of decreasing priority. But if *highest* is non-`nil`, then the return value is just one coding system, the one that is highest in priority.

If the region contains only ASCII characters, the value is `undecided` or `(undecided)`.

detect-coding-string *string highest* Function
This function is like `detect-coding-region` except that it operates on the contents of *string* instead of bytes in the buffer.

See Section 36.6 [Process Information], page 697, for how to examine or set the coding systems used for I/O to a subprocess.

32.10.4 User-Chosen Coding Systems

select-safe-coding-system *from to* &optional *preferred-coding-system* Function
This function selects a coding system for encoding the text between *from* and *to*, asking the user to choose if necessary.

The optional argument *preferred-coding-system* specifies a coding system to try first. If that one can handle the text in the specified region, then it is used. If this argument is omitted, the current buffer's value of `buffer-file-coding-system` is tried first.

If the region contains some multibyte characters that the preferred coding system cannot encode, this function asks the user to choose from a list of coding systems which can encode the text, and returns the user's choice.

One other kludgy feature: if *from* is a string, the string is the target text, and *to* is ignored.

Here are two functions you can use to let the user specify a coding system, with completion. See Section 19.5 [Completion], page 301.

read-coding-system *prompt* &optional *default* Function
This function reads a coding system using the minibuffer, prompting with string *prompt*, and returns the coding system name as a symbol. If the user enters null input, *default* specifies which coding system to return. It should be a symbol or a string.

read-non-nil-coding-system *prompt* Function
This function reads a coding system using the minibuffer, prompting with string *prompt*, and returns the coding system name as a symbol. If the user tries to enter null input, it asks the user to try again. See Section 32.10 [Coding Systems], page 636.

32.10.5 Default Coding Systems

This section describes variables that specify the default coding system for certain files or when running certain subprograms, and the function that I/O operations use to access them.

The idea of these variables is that you set them once and for all to the defaults you want, and then do not change them again. To specify a particular coding system for a particular operation in a Lisp program, don't change these variables; instead, override them using `coding-system-for-read` and `coding-system-for-write` (see Section 32.10.6 [Specifying Coding Systems], page 642).

file-coding-system-alist Variable
This variable is an alist that specifies the coding systems to use for reading and writing particular files. Each element has the form (*pattern* . *coding*), where *pattern* is a regular expression that matches certain file names. The element applies to file names that match *pattern*.

The CDR of the element, *coding*, should be either a coding system, a cons cell containing two coding systems, or a function symbol. If *val* is a coding system, that coding system is used for both reading the file and writing it. If *val* is a cons cell containing two coding systems, its CAR specifies the coding system for decoding, and its CDR specifies the coding system for encoding.

If *val* is a function symbol, the function must return a coding system or a cons cell containing two coding systems. This value is used as described above.

process-coding-system-alist Variable
This variable is an alist specifying which coding systems to use for a subprocess, depending on which program is running in the subprocess. It

works like `file-coding-system-alist`, except that *pattern* is matched against the program name used to start the subprocess. The coding system or systems specified in this alist are used to initialize the coding systems used for I/O to the subprocess, but you can specify other coding systems later using `set-process-coding-system`.

Warning: Coding systems such as `undecided` which determine the coding system from the data do not work entirely reliably with asynchronous subprocess output. This is because Emacs handles asynchronous subprocess output in batches, as it arrives. If the coding system leaves the character code conversion unspecified, or leaves the end-of-line conversion unspecified, Emacs must try to detect the proper conversion from one batch at a time, and this does not always work.

Therefore, with an asynchronous subprocess, if at all possible, use a coding system which determines both the character code conversion and the end of line conversion—that is, one like `latin-1-unix`, rather than `undecided` or `latin-1`.

network-coding-system-alist Variable

This variable is an alist that specifies the coding system to use for network streams. It works much like `file-coding-system-alist`, with the difference that the *pattern* in an element may be either a port number or a regular expression. If it is a regular expression, it is matched against the network service name used to open the network stream.

default-process-coding-system Variable

This variable specifies the coding systems to use for subprocess (and network stream) input and output, when nothing else specifies what to do.

The value should be a cons cell of the form (*input-coding* . *output-coding*). Here *input-coding* applies to input from the subprocess, and *output-coding* applies to output to it.

find-operation-coding-system *operation* &rest *arguments* Function

This function returns the coding system to use (by default) for performing *operation* with *arguments*. The value has this form:

(*decoding-system encoding-system*)

The first element, *decoding-system*, is the coding system to use for decoding (in case *operation* does decoding), and *encoding-system* is the coding system for encoding (in case *operation* does encoding).

The argument *operation* should be an Emacs I/O primitive: `insert-file-contents`, `write-region`, `call-process`, `call-process-region`, `start-process`, or `open-network-stream`.

The remaining arguments should be the same arguments that might be given to that I/O primitive. Depending on which primitive, one of those

arguments is selected as the *target*. For example, if *operation* does file I/O, whichever argument specifies the file name is the target. For subprocess primitives, the process name is the target. For `open-network-stream`, the target is the service name or port number.

This function looks up the target in `file-coding-system-alist`, `process-coding-system-alist`, or `network-coding-system-alist`, depending on *operation*. See Section 32.10.5 [Default Coding Systems], page 640.

32.10.6 Specifying a Coding System for One Operation

You can specify the coding system for a specific operation by binding the variables `coding-system-for-read` and/or `coding-system-for-write`.

coding-system-for-read Variable

If this variable is non-`nil`, it specifies the coding system to use for reading a file, or for input from a synchronous subprocess.

It also applies to any asynchronous subprocess or network stream, but in a different way: the value of `coding-system-for-read` when you start the subprocess or open the network stream specifies the input decoding method for that subprocess or network stream. It remains in use for that subprocess or network stream unless and until overridden.

The right way to use this variable is to bind it with `let` for a specific I/O operation. Its global value is normally `nil`, and you should not globally set it to any other value. Here is an example of the right way to use the variable:

```
;; Read the file with no character code conversion.
;; Assume CRLF represents end-of-line.
(let ((coding-system-for-write 'emacs-mule-dos))
  (insert-file-contents filename))
```

When its value is non-`nil`, `coding-system-for-read` takes precedence over all other methods of specifying a coding system to use for input, including `file-coding-system-alist`, `process-coding-system-alist` and `network-coding-system-alist`.

coding-system-for-write Variable

This works much like `coding-system-for-read`, except that it applies to output rather than input. It affects writing to files, subprocesses, and net connections.

When a single operation does both input and output, as do `call-process-region` and `start-process`, both `coding-system-for-read` and `coding-system-for-write` affect it.

inhibit-eol-conversion Variable

When this variable is non-`nil`, no end-of-line conversion is done, no matter which coding system is specified. This applies to all the Emacs I/O and subprocess primitives, and to the explicit encoding and decoding functions (see Section 32.10.7 [Explicit Encoding], page 643).

32.10.7 Explicit Encoding and Decoding

All the operations that transfer text in and out of Emacs have the ability to use a coding system to encode or decode the text. You can also explicitly encode and decode text using the functions in this section.

The result of encoding, and the input to decoding, are not ordinary text. They are "raw bytes"—bytes that represent text in the same way that an external file would. When a buffer contains raw bytes, it is most natural to mark that buffer as using unibyte representation, using `set-buffer-multibyte` (see Section 32.3 [Selecting a Representation], page 631), but this is not required. If the buffer's contents are only temporarily raw, leave the buffer multibyte, which will be correct after you decode them.

The usual way to get raw bytes in a buffer, for explicit decoding, is to read them from a file with `insert-file-contents-literally` (see Section 24.3 [Reading from Files], page 439) or specify a non-`nil` *rawfile* argument when visiting a file with `find-file-noselect`.

The usual way to use the raw bytes that result from explicitly encoding text is to copy them to a file or process—for example, to write them with `write-region` (see Section 24.4 [Writing to Files], page 440), and suppress encoding for that `write-region` call by binding `coding-system-for-write` to `no-conversion`.

Raw bytes sometimes contain overlong byte-sequences that look like a proper multibyte character plus extra bytes containing trailing codes. For most purposes, Emacs treats such a sequence in a buffer or string as a single character, and if you look at its character code, you get the value that corresponds to the multibyte character sequence—the extra bytes are disregarded. This behavior is not quite clean, but raw bytes are used only in limited places in Emacs, so as a practical matter problems can be avoided.

encode-coding-region *start end coding-system* Function

This function encodes the text from *start* to *end* according to coding system *coding-system*. The encoded text replaces the original text in the buffer. The result of encoding is "raw bytes," but the buffer remains multibyte if it was multibyte before.

encode-coding-string *string coding-system* Function

This function encodes the text in *string* according to coding system *coding-system*. It returns a new string containing the encoded text. The result of encoding is a unibyte string of "raw bytes."

decode-coding-region *start end coding-system* Function
This function decodes the text from *start* to *end* according to coding system *coding-system*. The decoded text replaces the original text in the buffer. To make explicit decoding useful, the text before decoding ought to be "raw bytes."

decode-coding-string *string coding-system* Function
This function decodes the text in *string* according to coding system *coding-system*. It returns a new string containing the decoded text. To make explicit decoding useful, the contents of *string* ought to be "raw bytes."

32.10.8 Terminal I/O Encoding

Emacs can decode keyboard input using a coding system, and encode terminal output. This is useful for terminals that transmit or display text using a particular encoding such as Latin-1. Emacs does not set `last-coding-system-used` for encoding or decoding for the terminal.

keyboard-coding-system Function
This function returns the coding system that is in use for decoding keyboard input—or `nil` if no coding system is to be used.

set-keyboard-coding-system *coding-system* Function
This function specifies *coding-system* as the coding system to use for decoding keyboard input. If *coding-system* is `nil`, that means do not decode keyboard input.

terminal-coding-system Function
This function returns the coding system that is in use for encoding terminal output—or `nil` for no encoding.

set-terminal-coding-system *coding-system* Function
This function specifies *coding-system* as the coding system to use for encoding terminal output. If *coding-system* is `nil`, that means do not encode terminal output.

32.10.9 MS-DOS File Types

Emacs on MS-DOS and on MS-Windows recognizes certain file names as text files or binary files. By "binary file" we mean a file of literal byte values that are not necessary meant to be characters. Emacs does no end-of-line conversion and no character code conversion for a binary file. Meanwhile, when you create a new file which is marked by its name as a "text file", Emacs uses DOS end-of-line conversion.

buffer-file-type Variable

This variable, automatically buffer-local in each buffer, records the file type of the buffer's visited file. When a buffer does not specify a coding system with `buffer-file-coding-system`, this variable is used to determine which coding system to use when writing the contents of the buffer. It should be `nil` for text, `t` for binary. If it is `t`, the coding system is `no-conversion`. Otherwise, `undecided-dos` is used.

Normally this variable is set by visiting a file; it is set to `nil` if the file was visited without any actual conversion.

file-name-buffer-file-type-alist User Option

This variable holds an alist for recognizing text and binary files. Each element has the form (*regexp* . *type*), where *regexp* is matched against the file name, and *type* may be `nil` for text, `t` for binary, or a function to call to compute which. If it is a function, then it is called with a single argument (the file name) and should return `t` or `nil`.

Emacs when running on MS-DOS or MS-Windows checks this alist to decide which coding system to use when reading a file. For a text file, `undecided-dos` is used. For a binary file, `no-conversion` is used.

If no element in this alist matches a given file name, then `default-buffer-file-type` says how to treat the file.

default-buffer-file-type User Option

This variable says how to handle files for which `file-name-buffer-file-type-alist` says nothing about the type.

If this variable is non-`nil`, then these files are treated as binary: the coding system `no-conversion` is used. Otherwise, nothing special is done for them—the coding system is deduced solely from the file contents, in the usual Emacs fashion.

32.11 Input Methods

Input methods provide convenient ways of entering non-ASCII characters from the keyboard. Unlike coding systems, which translate non-ASCII characters to and from encodings meant to be read by programs, input methods provide human-friendly commands. (See section "Input Methods" in *The GNU Emacs Manual*, for information on how users use input methods to enter text.) How to define input methods is not yet documented in this manual, but here we describe how to use them.

Each input method has a name, which is currently a string; in the future, symbols may also be usable as input method names.

current-input-method Variable

This variable holds the name of the input method now active in the current buffer. (It automatically becomes local in each buffer when set in any fashion.) It is `nil` if no input method is active in the buffer now.

default-input-method Variable

This variable holds the default input method for commands that choose an input method. Unlike `current-input-method`, this variable is normally global.

set-input-method *input-method* Function

This function activates input method *input-method* for the current buffer. It also sets `default-input-method` to *input-method*. If *input-method* is `nil`, this function deactivates any input method for the current buffer.

read-input-method-name *prompt* &optional *default* *inhibit-null* Function

This function reads an input method name with the minibuffer, prompting with *prompt*. If *default* is non-`nil`, that is returned by default, if the user enters empty input. However, if *inhibit-null* is non-`nil`, empty input signals an error.

The returned value is a string.

input-method-alist Variable

This variable defines all the supported input methods. Each element defines one input method, and should have the form:

(*input-method language-env activate-func*
title description args...)

Here *input-method* is the input method name, a string; *language-env* is another string, the name of the language environment this input method is recommended for. (That serves only for documentation purposes.)

title is a string to display in the mode line while this method is active. *description* is a string describing this method and what it is good for.

activate-func is a function to call to activate this method. The *args*, if any, are passed as arguments to *activate-func*. All told, the arguments to *activate-func* are *input-method* and the *args*.

The fundamental interface to input methods is through the variable `input-method-function`. See Section 20.6.2 [Reading One Event], page 345.

33 Searching and Matching

GNU Emacs provides two ways to search through a buffer for specified text: exact string searches and regular expression searches. After a regular expression search, you can examine the *match data* to determine which text matched the whole regular expression or various portions of it.

The '`skip-chars...`' functions also perform a kind of searching. See Section 29.2.7 [Skipping Characters], page 559.

33.1 Searching for Strings

These are the primitive functions for searching through the text in a buffer. They are meant for use in programs, but you may call them interactively. If you do so, they prompt for the search string; *limit* and *noerror* are set to `nil`, and *repeat* is set to 1.

These search functions convert the search string to multibyte if the buffer is multibyte; they convert the search string to unibyte if the buffer is unibyte. See Section 32.1 [Text Representations], page 629.

search-forward *string* &optional *limit noerror repeat* Command

This function searches forward from point for an exact match for *string*. If successful, it sets point to the end of the occurrence found, and returns the new value of point. If no match is found, the value and side effects depend on *noerror* (see below).

In the following example, point is initially at the beginning of the line. Then `(search-forward "fox")` moves point after the last letter of '`fox`':

```
---------- Buffer: foo ----------
*The quick brown fox jumped over the lazy dog.
---------- Buffer: foo ----------

(search-forward "fox")
     ⇒ 20

---------- Buffer: foo ----------
The quick brown fox* jumped over the lazy dog.
---------- Buffer: foo ----------
```

The argument *limit* specifies the upper bound to the search. (It must be a position in the current buffer.) No match extending after that position is accepted. If *limit* is omitted or `nil`, it defaults to the end of the accessible portion of the buffer.

What happens when the search fails depends on the value of *noerror*. If *noerror* is `nil`, a `search-failed` error is signaled. If *noerror* is `t`, `search-forward` returns `nil` and does nothing. If *noerror* is neither `nil` nor `t`, then `search-forward` moves point to the upper bound and returns

`nil`. (It would be more consistent now to return the new position of point in that case, but some existing programs may depend on a value of `nil`.)

If *repeat* is supplied (it must be a positive number), then the search is repeated that many times (each time starting at the end of the previous time's match). If these successive searches succeed, the function succeeds, moving point and returning its new value. Otherwise the search fails.

search-backward *string* &optional *limit noerror repeat* Command

This function searches backward from point for *string*. It is just like `search-forward` except that it searches backwards and leaves point at the beginning of the match.

word-search-forward *string* &optional *limit noerror repeat* Command

This function searches forward from point for a "word" match for *string*. If it finds a match, it sets point to the end of the match found, and returns the new value of point.

Word matching regards *string* as a sequence of words, disregarding punctuation that separates them. It searches the buffer for the same sequence of words. Each word must be distinct in the buffer (searching for the word '`ball`' does not match the word '`balls`'), but the details of punctuation and spacing are ignored (searching for '`ball boy`' does match '`ball. Boy!`').

In this example, point is initially at the beginning of the buffer; the search leaves it between the '`y`' and the '`!`'.

```
---------- Buffer: foo ----------
*He said "Please!  Find
the ball boy!"
---------- Buffer: foo ----------

(word-search-forward "Please find the ball, boy.")
     ⇒ 35

---------- Buffer: foo ----------
He said "Please!  Find
the ball boy*!"
---------- Buffer: foo ----------
```

If *limit* is non-`nil` (it must be a position in the current buffer), then it is the upper bound to the search. The match found must not extend after that position.

If *noerror* is `nil`, then `word-search-forward` signals an error if the search fails. If *noerror* is `t`, then it returns `nil` instead of signaling an error. If *noerror* is neither `nil` nor `t`, it moves point to *limit* (or the end of the buffer) and returns `nil`.

If *repeat* is non-`nil`, then the search is repeated that many times. Point is positioned at the end of the last match.

word-search-backward *string* &optional *limit noerror* *repeat* Command

This function searches backward from point for a word match to *string*. This function is just like `word-search-forward` except that it searches backward and normally leaves point at the beginning of the match.

33.2 Regular Expressions

A *regular expression* (*regexp*, for short) is a pattern that denotes a (possibly infinite) set of strings. Searching for matches for a regexp is a very powerful operation. This section explains how to write regexps; the following section says how to search for them.

33.2.1 Syntax of Regular Expressions

Regular expressions have a syntax in which a few characters are special constructs and the rest are *ordinary*. An ordinary character is a simple regular expression that matches that character and nothing else. The special characters are '`.`', '`*`', '`+`', '`?`', '`[`', '`]`', '`^`', '`$`', and '`\`'; no new special characters will be defined in the future. Any other character appearing in a regular expression is ordinary, unless a '`\`' precedes it.

For example, '`f`' is not a special character, so it is ordinary, and therefore '`f`' is a regular expression that matches the string '`f`' and no other string. (It does *not* match the string '`ff`'.) Likewise, '`o`' is a regular expression that matches only '`o`'.

Any two regular expressions *a* and *b* can be concatenated. The result is a regular expression that matches a string if *a* matches some amount of the beginning of that string and *b* matches the rest of the string.

As a simple example, we can concatenate the regular expressions '`f`' and '`o`' to get the regular expression '`fo`', which matches only the string '`fo`'. Still trivial. To do something more powerful, you need to use one of the special characters. Here is a list of them:

'`.`' (Period)
: is a special character that matches any single character except a newline. Using concatenation, we can make regular expressions like '`a.b`', which matches any three-character string that begins with '`a`' and ends with '`b`'.

'`*`'
: is not a construct by itself; it is a postfix operator that means to match the preceding regular expression repetitively as many times as possible. Thus, '`o*`' matches any number of '`o`'s (including no '`o`'s).

'`*`' always applies to the *smallest* possible preceding expression. Thus, '`fo*`' has a repeating '`o`', not a repeating '`fo`'. It matches '`f`', '`fo`', '`foo`', and so on.

The matcher processes a '`*`' construct by matching, immediately, as many repetitions as can be found. Then it continues with the rest of the pattern. If that fails, backtracking occurs, discarding some of the matches of the '`*`'-modified construct in the hope that that will make it possible to match the rest of the pattern. For example, in matching '`ca*ar`' against the string '`caaar`', the '`a*`' first tries to match all three '`a`'s; but the rest of the pattern is '`ar`' and there is only '`r`' left to match, so this try fails. The next alternative is for '`a*`' to match only two '`a`'s. With this choice, the rest of the regexp matches successfully.

Nested repetition operators can be extremely slow if they specify backtracking loops. For example, it could take hours for the regular expression '`\(x+y*\)*a`' to try to match the sequence '`xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxz`', before it ultimately fails. The slowness is because Emacs must try each imaginable way of grouping the 35 '`x`'s before concluding that none of them can work. To make sure your regular expressions run fast, check nested repetitions carefully.

'`+`' is a postfix operator, similar to '`*`' except that it must match the preceding expression at least once. So, for example, '`ca+r`' matches the strings '`car`' and '`caaaar`' but not the string '`cr`', whereas '`ca*r`' matches all three strings.

'`?`' is a postfix operator, similar to '`*`' except that it must match the preceding expression either once or not at all. For example, '`ca?r`' matches '`car`' or '`cr`'; nothing else.

'`[ ... ]`' is a *character alternative*, which begins with '`[`' and is terminated by '`]`'. In the simplest case, the characters between the two brackets are what this character alternative can match.

Thus, '`[ad]`' matches either one '`a`' or one '`d`', and '`[ad]*`' matches any string composed of just '`a`'s and '`d`'s (including the empty string), from which it follows that '`c[ad]*r`' matches '`cr`', '`car`', '`cdr`', '`caddaar`', etc.

You can also include character ranges in a character alternative, by writing the starting and ending characters with a '`-`' between them. Thus, '`[a-z]`' matches any lower-case ASCII letter. Ranges may be intermixed freely with individual characters, as in '`[a-z$%.]`', which matches any lower case ASCII letter or '`$`', '`%`' or period.

You cannot always match all non-ASCII characters with the regular expression '`[\200-\377]`'. This works when searching

a unibyte buffer or string (see Section 32.1 [Text Representations], page 629), but not in a multibyte buffer or string, because many non-ASCII characters have codes above octal 0377. However, the regular expression '`[^\000-\177]`' does match all non-ASCII characters, in both multibyte and unibyte representations, because only the ASCII characters are excluded.

The beginning and end of a range must be in the same character set (see Section 32.5 [Character Sets], page 632). Thus, '`[a-\x8e0]`' is invalid because '`a`' is in the ASCII character set but the character 0x8e0 ('`a`' with grave accent) is in the Emacs character set for Latin-1.

Note that the usual regexp special characters are not special inside a character alternative. A completely different set of characters are special inside character alternatives: '`]`', '`-`' and '`^`'.

To include a '`]`' in a character alternative, you must make it the first character. For example, '`[]a]`' matches '`]`' or '`a`'. To include a '`-`', write '`-`' as the first or last character of the character alternative, or put it after a range. Thus, '`[]-]`' matches both '`]`' and '`-`'.

To include '`^`' in a character alternative, put it anywhere but at the beginning.

'`[^ ... ]`' '`[^`' begins a *complemented character alternative*, which matches any character except the ones specified. Thus, '`[^a-z0-9A-Z]`' matches all characters *except* letters and digits.

'`^`' is not special in a character alternative unless it is the first character. The character following the '`^`' is treated as if it were first (in other words, '`-`' and '`]`' are not special there).

A complemented character alternative can match a newline, unless newline is mentioned as one of the characters not to match. This is in contrast to the handling of regexps in programs such as `grep`.

'`^`' is a special character that matches the empty string, but only at the beginning of a line in the text being matched. Otherwise it fails to match anything. Thus, '`^foo`' matches a '`foo`' that occurs at the beginning of a line.

When matching a string instead of a buffer, '`^`' matches at the beginning of the string or after a newline character '`\n`'.

'`$`' is similar to '`^`' but matches only at the end of a line. Thus, '`x+$`' matches a string of one '`x`' or more at the end of a line.

When matching a string instead of a buffer, '`$`' matches at the end of the string or before a newline character '`\n`'.

'`\`' has two functions: it quotes the special characters (including '`\`'), and it introduces additional special constructs.

Because '`\`' quotes special characters, '`\$`' is a regular expression that matches only '`$`', and '`\[`' is a regular expression that matches only '`[`', and so on.

Note that '`\`' also has special meaning in the read syntax of Lisp strings (see Section 2.3.8 [String Type], page 27), and must be quoted with '`\`'. For example, the regular expression that matches the '`\`' character is '`\\`'. To write a Lisp string that contains the characters '`\\`', Lisp syntax requires you to quote each '`\`' with another '`\`'. Therefore, the read syntax for a regular expression matching '`\`' is `"\\\\"`.

Please note: For historical compatibility, special characters are treated as ordinary ones if they are in contexts where their special meanings make no sense. For example, '`*foo`' treats '`*`' as ordinary since there is no preceding expression on which the '`*`' can act. It is poor practice to depend on this behavior; quote the special character anyway, regardless of where it appears.

For the most part, '`\`' followed by any character matches only that character. However, there are several exceptions: two-character sequences starting with '`\`' which have special meanings. (The second character in such a sequence is always ordinary when used on its own.) Here is a table of '`\`' constructs.

'`\|`' specifies an alternative. Two regular expressions *a* and *b* with '`\|`' in between form an expression that matches anything that either *a* or *b* matches.

Thus, '`foo\|bar`' matches either '`foo`' or '`bar`' but no other string.

'`\|`' applies to the largest possible surrounding expressions. Only a surrounding '`\( ... \)`' grouping can limit the grouping power of '`\|`'.

Full backtracking capability exists to handle multiple uses of '`\|`'.

'`\( ... \)`' is a grouping construct that serves three purposes:

1. To enclose a set of '`\|`' alternatives for other operations. Thus, the regular expression '`\(foo\|bar\)x`' matches either '`foox`' or '`barx`'.
2. To enclose a complicated expression for the postfix operators '`*`', '`+`' and '`?`' to operate on. Thus, '`ba\(na\)*`' matches '`ba`', '`bana`', '`banana`', '`bananana`', etc., with any number (zero or more) of '`na`' strings.
3. To record a matched substring for future reference.

This last application is not a consequence of the idea of a parenthetical grouping; it is a separate feature that happens to be

assigned as a second meaning to the same '`\( ... \)`' construct because there is no conflict in practice between the two meanings. Here is an explanation of this feature:

'`\`*digit*' matches the same text that matched the *digit*th occurrence of a '`\( ... \)`' construct.

In other words, after the end of a '`\( ... \)`' construct, the matcher remembers the beginning and end of the text matched by that construct. Then, later on in the regular expression, you can use '`\`' followed by *digit* to match that same text, whatever it may have been.

The strings matching the first nine '`\( ... \)`' constructs appearing in a regular expression are assigned numbers 1 through 9 in the order that the open parentheses appear in the regular expression. So you can use '`\1`' through '`\9`' to refer to the text matched by the corresponding '`\( ... \)`' constructs.

For example, '`\(.*\)\1`' matches any newline-free string that is composed of two identical halves. The '`\(.*\)`' matches the first half, which may be anything, but the '`\1`' that follows must match the same exact text.

'`\w`' matches any word-constituent character. The editor syntax table determines which characters these are. See Chapter 34 [Syntax Tables], page 669.

'`\W`' matches any character that is not a word constituent.

'`\s`*code*' matches any character whose syntax is *code*. Here *code* is a character that represents a syntax code: thus, '`w`' for word constituent, '`-`' for whitespace, '`(`' for open parenthesis, etc. To represent whitespace syntax, use either '`-`' or a space character. See Section 34.2.1 [Syntax Class Table], page 670, for a list of syntax codes and the characters that stand for them.

'`\S`*code*' matches any character whose syntax is not *code*.

The following regular expression constructs match the empty string—that is, they don't use up any characters—but whether they match depends on the context.

'`\``' matches the empty string, but only at the beginning of the buffer or string being matched against.

'`\'`' matches the empty string, but only at the end of the buffer or string being matched against.

'`\=`' matches the empty string, but only at point. (This construct is not defined when matching against a string.)

'`\b`' matches the empty string, but only at the beginning or end of a word. Thus, '`\bfoo\b`' matches any occurrence of '`foo`' as

a separate word. '`\bballs?\b`' matches '`ball`' or '`balls`' as a separate word.

'`\b`' matches at the beginning or end of the buffer regardless of what text appears next to it.

'`\B`' matches the empty string, but *not* at the beginning or end of a word.

'`\<`' matches the empty string, but only at the beginning of a word. '`\<`' matches at the beginning of the buffer only if a word-constituent character follows.

'`\>`' matches the empty string, but only at the end of a word. '`\>`' matches at the end of the buffer only if the contents end with a word-constituent character.

Not every string is a valid regular expression. For example, a string with unbalanced square brackets is invalid (with a few exceptions, such as '`[]]`'), and so is a string that ends with a single '`\`'. If an invalid regular expression is passed to any of the search functions, an `invalid-regexp` error is signaled.

regexp-quote *string* Function

This function returns a regular expression string that matches exactly *string* and nothing else. This allows you to request an exact string match when calling a function that wants a regular expression.

```
(regexp-quote "^The cat$")
     ⇒ "\\^The cat\\$"
```

One use of `regexp-quote` is to combine an exact string match with context described as a regular expression. For example, this searches for the string that is the value of *string*, surrounded by whitespace:

```
(re-search-forward
 (concat "\\s-" (regexp-quote string) "\\s-"))
```

regexp-opt *strings* &optional *paren* Function

This function returns an efficient regular expression that will match any of the strings *strings*. This is useful when you need to make matching or searching as fast as possible—for example, for Font Lock mode.

If the optional argument *paren* is non-`nil`, then the returned regular expression is always enclosed by at least one parentheses-grouping construct.

This simplified definition of `regexp-opt` produces a regular expression which is equivalent to the actual value (but not as efficient):

```
(defun regexp-opt (strings paren)
  (let ((open-paren (if paren "\\(" ""))
        (close-paren (if paren "\\)" "")))
    (concat open-paren
```

```
                (mapconcat 'regexp-quote strings "\\|")
                close-paren)))
```

regexp-opt-depth *regexp* Function

This function returns the total number of grouping constructs (parenthesized expressions) in *regexp*.

33.2.2 Complex Regexp Example

Here is a complicated regexp, used by Emacs to recognize the end of a sentence together with any whitespace that follows. It is the value of the variable `sentence-end`.

First, we show the regexp as a string in Lisp syntax to distinguish spaces from tab characters. The string constant begins and ends with a double-quote. '`\"`' stands for a double-quote as part of the string, '`\\`' for a backslash as part of the string, '`\t`' for a tab and '`\n`' for a newline.

```
"[.?!][]\"')}]*\\($\\| $\\|\t\\|  \\)[ \t\n]*"
```

In contrast, if you evaluate the variable `sentence-end`, you will see the following:

```
sentence-end
     ⇒ "[.?!][]\"')}]*\\($\\| $\\|  \\|  \\)[
]*"
```

In this output, tab and newline appear as themselves.

This regular expression contains four parts in succession and can be deciphered as follows:

`[.?!]` The first part of the pattern is a character alternative that matches any one of three characters: period, question mark, and exclamation mark. The match must begin with one of these three characters.

`[]\"')}]*`
The second part of the pattern matches any closing braces and quotation marks, zero or more of them, that may follow the period, question mark or exclamation mark. The `\"` is Lisp syntax for a double-quote in a string. The '`*`' at the end indicates that the immediately preceding regular expression (a character alternative, in this case) may be repeated zero or more times.

`\\($\\| $\\|\t\\|  \\)`
The third part of the pattern matches the whitespace that follows the end of a sentence: the end of a line (optionally with a space), or a tab, or two spaces. The double backslashes mark the parentheses and vertical bars as regular expression syntax; the parentheses delimit a group and the vertical bars separate alternatives. The dollar sign is used to match the end of a line.

`[ \t\n]*` Finally, the last part of the pattern matches any additional whitespace beyond the minimum needed to end a sentence.

33.3 Regular Expression Searching

In GNU Emacs, you can search for the next match for a regular expression either incrementally or not. For incremental search commands, see section "Regular Expression Search" in *The GNU Emacs Manual*. Here we describe only the search functions useful in programs. The principal one is `re-search-forward`.

These search functions convert the regular expression to multibyte if the buffer is multibyte; they convert the regular expression to unibyte if the buffer is unibyte. See Section 32.1 [Text Representations], page 629.

re-search-forward *regexp* &optional *limit noerror repeat* Command

This function searches forward in the current buffer for a string of text that is matched by the regular expression *regexp*. The function skips over any amount of text that is not matched by *regexp*, and leaves point at the end of the first match found. It returns the new value of point.

If *limit* is non-`nil` (it must be a position in the current buffer), then it is the upper bound to the search. No match extending after that position is accepted.

If *repeat* is supplied (it must be a positive number), then the search is repeated that many times (each time starting at the end of the previous time's match). If all these successive searches succeed, the function succeeds, moving point and returning its new value. Otherwise the function fails.

What happens when the function fails depends on the value of *noerror*. If *noerror* is `nil`, a `search-failed` error is signaled. If *noerror* is `t`, `re-search-forward` does nothing and returns `nil`. If *noerror* is neither `nil` nor `t`, then `re-search-forward` moves point to *limit* (or the end of the buffer) and returns `nil`.

In the following example, point is initially before the '`T`'. Evaluating the search call moves point to the end of that line (between the '`t`' of '`hat`' and the newline).

```
---------- Buffer: foo ----------
I read "*The cat in the hat
comes back" twice.
---------- Buffer: foo ----------
```

```
(re-search-forward "[a-z]+" nil t 5)
     ⇒ 27

---------- Buffer: foo ----------
I read "The cat in the hat*
comes back" twice.
---------- Buffer: foo ----------
```

re-search-backward *regexp* &optional *limit noerror repeat* Command

This function searches backward in the current buffer for a string of text that is matched by the regular expression *regexp*, leaving point at the beginning of the first text found.

This function is analogous to `re-search-forward`, but they are not simple mirror images. `re-search-forward` finds the match whose beginning is as close as possible to the starting point. If `re-search-backward` were a perfect mirror image, it would find the match whose end is as close as possible. However, in fact it finds the match whose beginning is as close as possible. The reason is that matching a regular expression at a given spot always works from beginning to end, and starts at a specified beginning position.

A true mirror-image of `re-search-forward` would require a special feature for matching regular expressions from end to beginning. It's not worth the trouble of implementing that.

string-match *regexp string* &optional *start* Function

This function returns the index of the start of the first match for the regular expression *regexp* in *string*, or `nil` if there is no match. If *start* is non-`nil`, the search starts at that index in *string*.

For example,

```
(string-match
 "quick" "The quick brown fox jumped quickly.")
     ⇒ 4
(string-match
 "quick" "The quick brown fox jumped quickly." 8)
     ⇒ 27
```

The index of the first character of the string is 0, the index of the second character is 1, and so on.

After this function returns, the index of the first character beyond the match is available as `(match-end 0)`. See Section 33.6 [Match Data], page 660.

```
(string-match
 "quick" "The quick brown fox jumped quickly." 8)
     ⇒ 27
```

```
(match-end 0)
     ⇒ 32
```

looking-at *regexp* Function

This function determines whether the text in the current buffer directly following point matches the regular expression *regexp*. "Directly following" means precisely that: the search is "anchored" and it can succeed only starting with the first character following point. The result is `t` if so, `nil` otherwise.

This function does not move point, but it updates the match data, which you can access using `match-beginning` and `match-end`. See Section 33.6 [Match Data], page 660.

In this example, point is located directly before the 'T'. If it were anywhere else, the result would be `nil`.

```
---------- Buffer: foo ----------
I read "⋆The cat in the hat
comes back" twice.
---------- Buffer: foo ----------

(looking-at "The cat in the hat$")
     ⇒ t
```

33.4 POSIX Regular Expression Searching

The usual regular expression functions do backtracking when necessary to handle the '`\|`' and repetition constructs, but they continue this only until they find *some* match. Then they succeed and report the first match found.

This section describes alternative search functions which perform the full backtracking specified by the POSIX standard for regular expression matching. They continue backtracking until they have tried all possibilities and found all matches, so they can report the longest match, as required by POSIX. This is much slower, so use these functions only when you really need the longest match.

posix-search-forward *regexp* &optional *limit noerror repeat* Function

This is like `re-search-forward` except that it performs the full backtracking specified by the POSIX standard for regular expression matching.

posix-search-backward *regexp* &optional *limit noerror repeat* Function

This is like `re-search-backward` except that it performs the full backtracking specified by the POSIX standard for regular expression matching.

posix-looking-at *regexp* Function

This is like `looking-at` except that it performs the full backtracking specified by the POSIX standard for regular expression matching.

posix-string-match *regexp string* &optional *start* Function

This is like `string-match` except that it performs the full backtracking specified by the POSIX standard for regular expression matching.

33.5 Search and Replace

perform-replace *from-string replacements query-flag regexp-flag delimited-flag* &optional *repeat-count map* Function

This function is the guts of `query-replace` and related commands. It searches for occurrences of *from-string* and replaces some or all of them. If *query-flag* is `nil`, it replaces all occurrences; otherwise, it asks the user what to do about each one.

If *regexp-flag* is non-`nil`, then *from-string* is considered a regular expression; otherwise, it must match literally. If *delimited-flag* is non-`nil`, then only replacements surrounded by word boundaries are considered.

The argument *replacements* specifies what to replace occurrences with. If it is a string, that string is used. It can also be a list of strings, to be used in cyclic order.

If *repeat-count* is non-`nil`, it should be an integer. Then it specifies how many times to use each of the strings in the *replacements* list before advancing cyclicly to the next one.

Normally, the keymap `query-replace-map` defines the possible user responses for queries. The argument *map*, if non-`nil`, is a keymap to use instead of `query-replace-map`.

query-replace-map Variable

This variable holds a special keymap that defines the valid user responses for `query-replace` and related functions, as well as `y-or-n-p` and `map-y-or-n-p`. It is unusual in two ways:

- The "key bindings" are not commands, just symbols that are meaningful to the functions that use this map.

- Prefix keys are not supported; each key binding must be for a single-event key sequence. This is because the functions don't use `read-key-sequence` to get the input; instead, they read a single event and look it up "by hand."

Here are the meaningful "bindings" for `query-replace-map`. Several of them are meaningful only for `query-replace` and friends.

`act` Do take the action being considered—in other words, "yes."

`skip` Do not take action for this question—in other words, "no."

`exit` Answer this question "no," and give up on the entire series of questions, assuming that the answers will be "no."

`act-and-exit`
Answer this question "yes," and give up on the entire series of questions, assuming that subsequent answers will be "no."

`act-and-show`
Answer this question "yes," but show the results—don't advance yet to the next question.

`automatic`
Answer this question and all subsequent questions in the series with "yes," without further user interaction.

`backup` Move back to the previous place that a question was asked about.

`edit` Enter a recursive edit to deal with this question—instead of any other action that would normally be taken.

`delete-and-edit`
Delete the text being considered, then enter a recursive edit to replace it.

`recenter` Redisplay and center the window, then ask the same question again.

`quit` Perform a quit right away. Only `y-or-n-p` and related functions use this answer.

`help` Display some help, then ask again.

33.6 The Match Data

Emacs keeps track of the positions of the start and end of segments of text found during a regular expression search. This means, for example, that you can search for a complex pattern, such as a date in an Rmail message, and then extract parts of the match under control of the pattern.

Because the match data normally describe the most recent search only, you must be careful not to do another search inadvertently between the

search you wish to refer back to and the use of the match data. If you can't avoid another intervening search, you must save and restore the match data around it, to prevent it from being overwritten.

33.6.1 Replacing the Text That Matched

This function replaces the text matched by the last search with *replacement*.

replace-match *replacement* &optional *fixedcase literal* *string subexp* Function

This function replaces the text in the buffer (or in *string*) that was matched by the last search. It replaces that text with *replacement*.

If you did the last search in a buffer, you should specify `nil` for *string*. Then `replace-match` does the replacement by editing the buffer; it leaves point at the end of the replacement text, and returns `t`.

If you did the search in a string, pass the same string as *string*. Then `replace-match` does the replacement by constructing and returning a new string.

If *fixedcase* is non-`nil`, then the case of the replacement text is not changed; otherwise, the replacement text is converted to a different case depending upon the capitalization of the text to be replaced. If the original text is all upper case, the replacement text is converted to upper case. If the first word of the original text is capitalized, then the first word of the replacement text is capitalized. If the original text contains just one word, and that word is a capital letter, `replace-match` considers this a capitalized first word rather than all upper case.

If `case-replace` is `nil`, then case conversion is not done, regardless of the value of *fixed-case*. See Section 33.7 [Searching and Case], page 665.

If *literal* is non-`nil`, then *replacement* is inserted exactly as it is, the only alterations being case changes as needed. If it is `nil` (the default), then the character '`\`' is treated specially. If a '`\`' appears in *replacement*, then it must be part of one of the following sequences:

'`\&`' '`\&`' stands for the entire text being replaced.

'`\`*n*' '`\`*n*', where *n* is a digit, stands for the text that matched the *n*th subexpression in the original regexp. Subexpressions are those expressions grouped inside '`\(...\)`'.

'`\\`' '`\\`' stands for a single '`\`' in the replacement text.

If *subexp* is non-`nil`, that says to replace just subexpression number *subexp* of the regexp that was matched, not the entire match. For example, after matching '`foo \(ba*r\)`', calling `replace-match` with 1 as *subexp* means to replace just the text that matched '`\(ba*r\)`'.

33.6.2 Simple Match Data Access

This section explains how to use the match data to find out what was matched by the last search or match operation.

You can ask about the entire matching text, or about a particular parenthetical subexpression of a regular expression. The *count* argument in the functions below specifies which. If *count* is zero, you are asking about the entire match. If *count* is positive, it specifies which subexpression you want.

Recall that the subexpressions of a regular expression are those expressions grouped with escaped parentheses, '`\(...\)`'. The *count*th subexpression is found by counting occurrences of '`\(`' from the beginning of the whole regular expression. The first subexpression is numbered 1, the second 2, and so on. Only regular expressions can have subexpressions—after a simple string search, the only information available is about the entire match.

A search which fails may or may not alter the match data. In the past, a failing search did not do this, but we may change it in the future.

match-string *count* &optional *in-string* Function

This function returns, as a string, the text matched in the last search or match operation. It returns the entire text if *count* is zero, or just the portion corresponding to the *count*th parenthetical subexpression, if *count* is positive. If *count* is out of range, or if that subexpression didn't match anything, the value is `nil`.

If the last such operation was done against a string with `string-match`, then you should pass the same string as the argument *in-string*. After a buffer search or match, you should omit *in-string* or pass `nil` for it; but you should make sure that the current buffer when you call `match-string` is the one in which you did the searching or matching.

match-string-no-properties *count* Function

This function is like `match-string` except that the result has no text properties.

match-beginning *count* Function

This function returns the position of the start of text matched by the last regular expression searched for, or a subexpression of it.

If *count* is zero, then the value is the position of the start of the entire match. Otherwise, *count* specifies a subexpression in the regular expression, and the value of the function is the starting position of the match for that subexpression.

The value is `nil` for a subexpression inside a '`\|`' alternative that wasn't used in the match.

match-end *count* Function

This function is like `match-beginning` except that it returns the position of the end of the match, rather than the position of the beginning.

Here is an example of using the match data, with a comment showing the positions within the text:

```
(string-match "\\(qu\\)\\(ick\\)"
              "The quick fox jumped quickly.")
              ;0123456789
     ⇒ 4

(match-string 0 "The quick fox jumped quickly.")
     ⇒ "quick"
(match-string 1 "The quick fox jumped quickly.")
     ⇒ "qu"
(match-string 2 "The quick fox jumped quickly.")
     ⇒ "ick"

(match-beginning 1)       ; The beginning of the match
     ⇒ 4                  ;    with 'qu' is at index 4.

(match-beginning 2)       ; The beginning of the match
     ⇒ 6                  ;    with 'ick' is at index 6.

(match-end 1)             ; The end of the match
     ⇒ 6                  ;    with 'qu' is at index 6.

(match-end 2)             ; The end of the match
     ⇒ 9                  ;    with 'ick' is at index 9.
```

Here is another example. Point is initially located at the beginning of the line. Searching moves point to between the space and the word 'in'. The beginning of the entire match is at the 9th character of the buffer ('T'), and the beginning of the match for the first subexpression is at the 13th character ('c').

```
(list
  (re-search-forward "The \\(cat \\)")
  (match-beginning 0)
  (match-beginning 1))
    ⇒ (9 9 13)

---------- Buffer: foo ----------
I read "The cat *in the hat comes back" twice.
        ^   ^
        9  13
---------- Buffer: foo ----------
```

(In this case, the index returned is a buffer position; the first character of the buffer counts as 1.)

33.6.3 Accessing the Entire Match Data

The functions `match-data` and `set-match-data` read or write the entire match data, all at once.

match-data Function

This function returns a newly constructed list containing all the information on what text the last search matched. Element zero is the position of the beginning of the match for the whole expression; element one is the position of the end of the match for the expression. The next two elements are the positions of the beginning and end of the match for the first subexpression, and so on. In general, element number $2n$ corresponds to `(match-beginning` *n*`)`; and element number $2n+1$ corresponds to `(match-end` *n*`)`.

All the elements are markers or `nil` if matching was done on a buffer, and all are integers or `nil` if matching was done on a string with `string-match`.

As always, there must be no possibility of intervening searches between the call to a search function and the call to `match-data` that is intended to access the match data for that search.

```
(match-data)
     ⇒  (#<marker at 9 in foo>
         #<marker at 17 in foo>
         #<marker at 13 in foo>
         #<marker at 17 in foo>)
```

set-match-data *match-list* Function

This function sets the match data from the elements of *match-list*, which should be a list that was the value of a previous call to `match-data`.

If *match-list* refers to a buffer that doesn't exist, you don't get an error; that sets the match data in a meaningless but harmless way.

`store-match-data` is a semi-obsolete alias for `set-match-data`.

33.6.4 Saving and Restoring the Match Data

When you call a function that may do a search, you may need to save and restore the match data around that call, if you want to preserve the match data from an earlier search for later use. Here is an example that shows the problem that arises if you fail to save the match data:

```
(re-search-forward "The \\(cat \\)")
     ⇒ 48
(foo)                   ; Perhaps foo does
                        ;   more searching.
(match-end 0)
     ⇒ 61               ; Unexpected result—not 48!
```

You can save and restore the match data with `save-match-data`:

save-match-data *body...* Macro
This macro executes *body*, saving and restoring the match data around it.

You could use `set-match-data` together with `match-data` to imitate the effect of the special form `save-match-data`. Here is how:

```
(let ((data (match-data)))
  (unwind-protect
      ...   ; Ok to change the original match data.
    (set-match-data data)))
```

Emacs automatically saves and restores the match data when it runs process filter functions (see Section 36.9.2 [Filter Functions], page 703) and process sentinels (see Section 36.10 [Sentinels], page 706).

33.7 Searching and Case

By default, searches in Emacs ignore the case of the text they are searching through; if you specify searching for '`FOO`', then '`Foo`' or '`foo`' is also considered a match. This applies to regular expressions, too; thus, '`[aB]`' would match '`a`' or '`A`' or '`b`' or '`B`'.

If you do not want this feature, set the variable `case-fold-search` to `nil`. Then all letters must match exactly, including case. This is a buffer-local variable; altering the variable affects only the current buffer. (See Section 10.10.1 [Intro to Buffer-Local], page 162.) Alternatively, you may change the value of `default-case-fold-search`, which is the default value of `case-fold-search` for buffers that do not override it.

Note that the user-level incremental search feature handles case distinctions differently. When given a lower case letter, it looks for a match of either case, but when given an upper case letter, it looks for an upper case letter only. But this has nothing to do with the searching functions used in Lisp code.

case-replace User Option
This variable determines whether the replacement functions should preserve case. If the variable is `nil`, that means to use the replacement text verbatim. A non-`nil` value means to convert the case of the replacement text according to the text being replaced.

The function `replace-match` is where this variable actually has its effect. See Section 33.6.1 [Replacing Match], page 661.

case-fold-search User Option
This buffer-local variable determines whether searches should ignore case. If the variable is `nil` they do not ignore case; otherwise they do ignore case.

default-case-fold-search Variable
The value of this variable is the default value for `case-fold-search` in buffers that do not override it. This is the same as `(default-value 'case-fold-search)`.

33.8 Standard Regular Expressions Used in Editing

This section describes some variables that hold regular expressions used for certain purposes in editing:

page-delimiter Variable
This is the regular expression describing line-beginnings that separate pages. The default value is `"^\014"` (i.e., `"^^L"` or `"^\C-l"`); this matches a line that starts with a formfeed character.

The following two regular expressions should *not* assume the match always starts at the beginning of a line; they should not use '`^`' to anchor the match. Most often, the paragraph commands do check for a match only at the beginning of a line, which means that '`^`' would be superfluous. When there is a nonzero left margin, they accept matches that start after the left margin. In that case, a '`^`' would be incorrect. However, a '`^`' is harmless in modes where a left margin is never used.

paragraph-separate Variable
This is the regular expression for recognizing the beginning of a line that separates paragraphs. (If you change this, you may have to change `paragraph-start` also.) The default value is `"[ \t\f]*$"`, which matches a line that consists entirely of spaces, tabs, and form feeds (after its left margin).

paragraph-start Variable
This is the regular expression for recognizing the beginning of a line that starts *or* separates paragraphs. The default value is `"[ \t\n\f]"`, which matches a line starting with a space, tab, newline, or form feed (after its left margin).

sentence-end Variable
This is the regular expression describing the end of a sentence. (All paragraph boundaries also end sentences, regardless.) The default value is:

```
"[.?!][]\"')}]*\\($\\| $\\|\t\\| \\)[ \t\n]*"
```

This means a period, question mark or exclamation mark, followed optionally by a closing parenthetical character, followed by tabs, spaces or new lines.

For a detailed explanation of this regular expression, see Section 33.2.2 [Regexp Example], page 655.

34 Syntax Tables

A *syntax table* specifies the syntactic textual function of each character. This information is used by the *parsing functions*, the complex movement commands, and others to determine where words, symbols, and other syntactic constructs begin and end. The current syntax table controls the meaning of the word motion functions (see Section 29.2.2 [Word Motion], page 553) and the list motion functions (see Section 29.2.6 [List Motion], page 558), as well as the functions in this chapter.

34.1 Syntax Table Concepts

A syntax table is a char-table (see Section 6.6 [Char-Tables], page 104). The element at index *c* describes the character with code *c*. The element's value should be a list that encodes the syntax of the character in question.

Syntax tables are used only for moving across text, not for the Emacs Lisp reader. Emacs Lisp uses built-in syntactic rules when reading Lisp expressions, and these rules cannot be changed. (Some Lisp systems provide ways to redefine the read syntax, but we decided to leave this feature out of Emacs Lisp for simplicity.)

Each buffer has its own major mode, and each major mode has its own idea of the syntactic class of various characters. For example, in Lisp mode, the character '`;`' begins a comment, but in C mode, it terminates a statement. To support these variations, Emacs makes the choice of syntax table local to each buffer. Typically, each major mode has its own syntax table and installs that table in each buffer that uses that mode. Changing this table alters the syntax in all those buffers as well as in any buffers subsequently put in that mode. Occasionally several similar modes share one syntax table. See Section 22.1.2 [Example Major Modes], page 394, for an example of how to set up a syntax table.

A syntax table can inherit the data for some characters from the standard syntax table, while specifying other characters itself. The "inherit" syntax class means "inherit this character's syntax from the standard syntax table." Just changing the standard syntax for a characters affects all syntax tables which inherit from it.

syntax-table-p *object* Function

This function returns `t` if *object* is a syntax table.

34.2 Syntax Descriptors

This section describes the syntax classes and flags that denote the syntax of a character, and how they are represented as a *syntax descriptor*, which is a Lisp string that you pass to `modify-syntax-entry` to specify the syntax you want.

The syntax table specifies a syntax class for each character. There is no necessary relationship between the class of a character in one syntax table and its class in any other table.

Each class is designated by a mnemonic character, which serves as the name of the class when you need to specify a class. Usually the designator character is one that is frequently in that class; however, its meaning as a designator is unvarying and independent of what syntax that character currently has.

A syntax descriptor is a Lisp string that specifies a syntax class, a matching character (used only for the parenthesis classes) and flags. The first character is the designator for a syntax class. The second character is the character to match; if it is unused, put a space there. Then come the characters for any desired flags. If no matching character or flags are needed, one character is sufficient.

For example, the syntax descriptor for the character '`*`' in C mode is '`. 23`' (i.e., punctuation, matching character slot unused, second character of a comment-starter, first character of an comment-ender), and the entry for '`/`' is '`. 14`' (i.e., punctuation, matching character slot unused, first character of a comment-starter, second character of a comment-ender).

34.2.1 Table of Syntax Classes

Here is a table of syntax classes, the characters that stand for them, their meanings, and examples of their use.

whitespace character Syntax class

Whitespace characters (designated by ' ' or '`-`') separate symbols and words from each other. Typically, whitespace characters have no other syntactic significance, and multiple whitespace characters are syntactically equivalent to a single one. Space, tab, newline and formfeed are classified as whitespace in almost all major modes.

word constituent Syntax class

Word constituents (designated by '`w`') are parts of normal English words and are typically used in variable and command names in programs. All upper- and lower-case letters, and the digits, are typically word constituents.

symbol constituent Syntax class

Symbol constituents (designated by '`_`') are the extra characters that are used in variable and command names along with word constituents. For example, the symbol constituents class is used in Lisp mode to indicate that certain characters may be part of symbol names even though they are not part of English words. These characters are '`$&*+-_<>`'. In standard C, the only non-word-constituent character that is valid in symbols is underscore ('`_`').

punctuation character Syntax class

Punctuation characters (designated by '`.`') are those characters that are used as punctuation in English, or are used in some way in a programming language to separate symbols from one another. Most programming language modes, including Emacs Lisp mode, have no characters in this class since the few characters that are not symbol or word constituents all have other uses.

open parenthesis character Syntax class

close parenthesis character Syntax class

Open and close *parenthesis characters* are characters used in dissimilar pairs to surround sentences or expressions. Such a grouping is begun with an open parenthesis character and terminated with a close. Each open parenthesis character matches a particular close parenthesis character, and vice versa. Normally, Emacs indicates momentarily the matching open parenthesis when you insert a close parenthesis. See Section 38.11 [Blinking], page 759.

The class of open parentheses is designated by '`(`', and that of close parentheses by '`)`'.

In English text, and in C code, the parenthesis pairs are '`()`', '`[]`', and '`{}`'. In Emacs Lisp, the delimiters for lists and vectors ('`()`' and '`[]`') are classified as parenthesis characters.

string quote Syntax class

String quote characters (designated by '`"`') are used in many languages, including Lisp and C, to delimit string constants. The same string quote character appears at the beginning and the end of a string. Such quoted strings do not nest.

The parsing facilities of Emacs consider a string as a single token. The usual syntactic meanings of the characters in the string are suppressed.

The Lisp modes have two string quote characters: double-quote ('`"`') and vertical bar ('`|`'). '`|`' is not used in Emacs Lisp, but it is used in Common Lisp. C also has two string quote characters: double-quote for strings, and single-quote ('`'`') for character constants.

English text has no string quote characters because English is not a programming language. Although quotation marks are used in English, we do not want them to turn off the usual syntactic properties of other characters in the quotation.

escape Syntax class

An *escape character* (designated by '`\`') starts an escape sequence such as is used in C string and character constants. The character '`\`' belongs to this class in both C and Lisp. (In C, it is used thus only inside strings, but it turns out to cause no trouble to treat it this way throughout C code.)

Characters in this class count as part of words if `words-include-escapes` is non-`nil`. See Section 29.2.2 [Word Motion], page 553.

character quote Syntax class

A *character quote character* (designated by '`/`') quotes the following character so that it loses its normal syntactic meaning. This differs from an escape character in that only the character immediately following is ever affected.

Characters in this class count as part of words if `words-include-escapes` is non-`nil`. See Section 29.2.2 [Word Motion], page 553.

This class is used for backslash in TEX mode.

paired delimiter Syntax class

Paired delimiter characters (designated by '`$`') are like string quote characters except that the syntactic properties of the characters between the delimiters are not suppressed. Only TEX mode uses a paired delimiter presently—the '`$`' that both enters and leaves math mode.

expression prefix Syntax class

An *expression prefix operator* (designated by '`'`') is used for syntactic operators that are considered as part of an expression if they appear next to one. In Lisp modes, these characters include the apostrophe, '`'`' (used for quoting), the comma, '`,`' (used in macros), and '`#`' (used in the read syntax for certain data types).

comment starter Syntax class

comment ender Syntax class

The *comment starter* and *comment ender* characters are used in various languages to delimit comments. These classes are designated by '`<`' and '`>`', respectively.

English text has no comment characters. In Lisp, the semicolon ('`;`') starts a comment and a newline or formfeed ends one.

inherit Syntax class

This syntax class does not specify a particular syntax. It says to look in the standard syntax table to find the syntax of this character. The designator for this syntax code is '`@`'.

generic comment delimiter Syntax class

A *generic comment delimiter* character starts or ends a special kind of comment. *Any* generic comment delimiter matches *any* generic comment delimiter, but they cannot match a comment starter or comment ender; generic comment delimiters can only match each other.

This syntax class is primarily meant for use with the `syntax-table` text property (see Section 34.4 [Syntax Properties], page 676). You can mark

any range of characters as forming a comment, by giving the first and last characters of the range `syntax-table` properties identifying them as generic comment delimiters.

generic string delimiter Syntax class

A *generic string delimiter* character starts or ends a string. This class differs from the string quote class in that *any* generic string delimiter can match any other generic string delimiter; but they do not match ordinary string quote characters.

This syntax class is primarily meant for use with the `syntax-table` text property (see Section 34.4 [Syntax Properties], page 676). You can mark any range of characters as forming a string constant, by giving the first and last characters of the range `syntax-table` properties identifying them as generic string delimiters.

34.2.2 Syntax Flags

In addition to the classes, entries for characters in a syntax table can specify flags. There are six possible flags, represented by the characters '`1`', '`2`', '`3`', '`4`', '`b`' and '`p`'.

All the flags except '`p`' are used to describe multi-character comment delimiters. The digit flags indicate that a character can *also* be part of a comment sequence, in addition to the syntactic properties associated with its character class. The flags are independent of the class and each other for the sake of characters such as '`*`' in C mode, which is a punctuation character, *and* the second character of a start-of-comment sequence ('`/*`'), *and* the first character of an end-of-comment sequence ('`*/`').

Here is a table of the possible flags for a character *c*, and what they mean:

- '`1`' means *c* is the start of a two-character comment-start sequence.
- '`2`' means *c* is the second character of such a sequence.
- '`3`' means *c* is the start of a two-character comment-end sequence.
- '`4`' means *c* is the second character of such a sequence.
- '`b`' means that *c* as a comment delimiter belongs to the alternative "b" comment style.

  Emacs supports two comment styles simultaneously in any one syntax table. This is for the sake of C++. Each style of comment syntax has its own comment-start sequence and its own comment-end sequence. Each comment must stick to one style or the other; thus, if it starts with the comment-start sequence of style "b", it must also end with the comment-end sequence of style "b".

  The two comment-start sequences must begin with the same character; only the second character may differ. Mark the second character of the "b"-style comment-start sequence with the '`b`' flag.

A comment-end sequence (one or two characters) applies to the "b" style if its first character has the '`b`' flag set; otherwise, it applies to the "a" style.

The appropriate comment syntax settings for C++ are as follows:

'`/`' '`124b`'

'`*`' '`23`'

newline '`>b`'

This defines four comment-delimiting sequences:

'`/*`' This is a comment-start sequence for "a" style because the second character, '`*`', does not have the '`b`' flag.

'`//`' This is a comment-start sequence for "b" style because the second character, '`/`', does have the '`b`' flag.

'`*/`' This is a comment-end sequence for "a" style because the first character, '`*`', does not have the '`b`' flag.

newline This is a comment-end sequence for "b" style, because the newline character has the '`b`' flag.

- '`p`' identifies an additional "prefix character" for Lisp syntax. These characters are treated as whitespace when they appear between expressions. When they appear within an expression, they are handled according to their usual syntax codes.

  The function `backward-prefix-chars` moves back over these characters, as well as over characters whose primary syntax class is prefix ('`'`'). See Section 34.5 [Motion and Syntax], page 677.

34.3 Syntax Table Functions

In this section we describe functions for creating, accessing and altering syntax tables.

make-syntax-table Function

This function creates a new syntax table. It inherits the syntax for letters and control characters from the standard syntax table. For other characters, the syntax is copied from the standard syntax table.

Most major mode syntax tables are created in this way.

copy-syntax-table &optional *table* Function

This function constructs a copy of *table* and returns it. If *table* is not supplied (or is `nil`), it returns a copy of the current syntax table. Otherwise, an error is signaled if *table* is not a syntax table.

modify-syntax-entry *char syntax-descriptor* &optional *table* Command

This function sets the syntax entry for *char* according to *syntax-descriptor*. The syntax is changed only for *table*, which defaults to the current buffer's syntax table, and not in any other syntax table. The argument *syntax-descriptor* specifies the desired syntax; this is a string beginning with a class designator character, and optionally containing a matching character and flags as well. See Section 34.2 [Syntax Descriptors], page 669.

This function always returns `nil`. The old syntax information in the table for this character is discarded.

An error is signaled if the first character of the syntax descriptor is not one of the twelve syntax class designator characters. An error is also signaled if *char* is not a character.

Examples:

```
;; Put the space character in class whitespace.
(modify-syntax-entry ?\   " ")
     ⇒ nil

;; Make '$' an open parenthesis character,
;;    with '^' as its matching close.
(modify-syntax-entry ?$ "(^")
     ⇒ nil

;; Make '^' a close parenthesis character,
;;    with '$' as its matching open.
(modify-syntax-entry ?^ ")$")
     ⇒ nil

;; Make '/' a punctuation character,
;;    the first character of a start-comment sequence,
;;    and the second character of an end-comment sequence.
;;    This is used in C mode.
(modify-syntax-entry ?/ ". 14")
     ⇒ nil
```

char-syntax *character* Function

This function returns the syntax class of *character*, represented by its mnemonic designator character. This returns *only* the class, not any matching parenthesis or flags.

An error is signaled if *char* is not a character.

The following examples apply to C mode. The first example shows that the syntax class of space is whitespace (represented by a space). The second example shows that the syntax of '`/`' is punctuation. This does not show the fact that it is also part of comment-start and -end sequences.

The third example shows that open parenthesis is in the class of open parentheses. This does not show the fact that it has a matching character, ')'.

```
(string (char-syntax ?\ ))
     ⇒ " "

(string (char-syntax ?/))
     ⇒ "."

(string (char-syntax ?\())
     ⇒ "("
```

We use `string` to make it easier to see the character returned by `char-syntax`.

set-syntax-table *table* Function

This function makes *table* the syntax table for the current buffer. It returns *table*.

syntax-table Function

This function returns the current syntax table, which is the table for the current buffer.

34.4 Syntax Properties

When the syntax table is not flexible enough to specify the syntax of a language, you can use `syntax-table` text properties to override the syntax table for specific character occurrences in the buffer. See Section 31.19 [Text Properties], page 610.

The valid values of `syntax-table` text property are:

syntax-table
: If the property value is a syntax table, that table is used instead of the current buffer's syntax table to determine the syntax for this occurrence of the character.

(*syntax-code* . *matching-char*)
: A cons cell of this format specifies the syntax for this occurrence of the character.

`nil`
: If the property is `nil`, the character's syntax is determined from the current syntax table in the usual way.

parse-sexp-lookup-properties Variable

If this is non-`nil`, the syntax scanning functions pay attention to syntax text properties. Otherwise they use only the current syntax table.

34.5 Motion and Syntax

This section describes functions for moving across characters that have certain syntax classes.

skip-syntax-forward *syntaxes* &optional *limit* Function
This function moves point forward across characters having syntax classes mentioned in *syntaxes*. It stops when it encounters the end of the buffer, or position *limit* (if specified), or a character it is not supposed to skip. The return value is the distance traveled, which is a nonnegative integer.

skip-syntax-backward *syntaxes* &optional *limit* Function
This function moves point backward across characters whose syntax classes are mentioned in *syntaxes*. It stops when it encounters the beginning of the buffer, or position *limit* (if specified), or a character it is not supposed to skip.

The return value indicates the distance traveled. It is an integer that is zero or less.

backward-prefix-chars Function
This function moves point backward over any number of characters with expression prefix syntax. This includes both characters in the expression prefix syntax class, and characters with the '`p`' flag.

34.6 Parsing Balanced Expressions

Here are several functions for parsing and scanning balanced expressions, also known as *sexps*, in which parentheses match in pairs. The syntax table controls the interpretation of characters, so these functions can be used for Lisp expressions when in Lisp mode and for C expressions when in C mode. See Section 29.2.6 [List Motion], page 558, for convenient higher-level functions for moving over balanced expressions.

parse-partial-sexp *start limit* &optional *target-depth* *stop-before* *state* *stop-comment* Function
This function parses a sexp in the current buffer starting at *start*, not scanning past *limit*. It stops at position *limit* or when certain criteria described below are met, and sets point to the location where parsing stops. It returns a value describing the status of the parse at the point where it stops.

If *state* is `nil`, *start* is assumed to be at the top level of parenthesis structure, such as the beginning of a function definition. Alternatively, you might wish to resume parsing in the middle of the structure. To do this, you must provide a *state* argument that describes the initial status of parsing.

If the third argument *target-depth* is non-`nil`, parsing stops if the depth in parentheses becomes equal to *target-depth*. The depth starts at 0, or at whatever is given in *state*.

If the fourth argument *stop-before* is non-`nil`, parsing stops when it comes to any character that starts a sexp. If *stop-comment* is non-`nil`, parsing stops when it comes to the start of a comment. If *stop-comment* is the symbol `syntax-table`, parsing stops after the start of a comment or a string, or the end of a comment or a string, whichever comes first.

The fifth argument *state* is a nine-element list of the same form as the value of this function, described below. (It is OK to omit the last element of the nine.) The return value of one call may be used to initialize the state of the parse on another call to `parse-partial-sexp`.

The result is a list of nine elements describing the final state of the parse:

0. The depth in parentheses, counting from 0.
1. The character position of the start of the innermost parenthetical grouping containing the stopping point; `nil` if none.
2. The character position of the start of the last complete subexpression terminated; `nil` if none.
3. Non-`nil` if inside a string. More precisely, this is the character that will terminate the string, or `t` if a generic string delimiter character should terminate it.
4. `t` if inside a comment (of either style).
5. `t` if point is just after a quote character.
6. The minimum parenthesis depth encountered during this scan.
7. What kind of comment is active: `nil` for a comment of style "a", `t` for a comment of style "b", and `syntax-table` for a comment that should be ended by a generic comment delimiter character.
8. The string or comment start position. While inside a comment, this is the position where the comment began; while inside a string, this is the position where the string began. When outside of strings and comments, this element is `nil`.

Elements 0, 3, 4, 5 and 7 are significant in the argument *state*.

This function is most often used to compute indentation for languages that have nested parentheses.

scan-lists *from count depth* Function

This function scans forward *count* balanced parenthetical groupings from position *from*. It returns the position where the scan stops. If *count* is negative, the scan moves backwards.

If *depth* is nonzero, parenthesis depth counting begins from that value. The only candidates for stopping are places where the depth in parentheses becomes zero; `scan-lists` counts *count* such places and then stops. Thus, a positive value for *depth* means go out *depth* levels of parenthesis.

Scanning ignores comments if `parse-sexp-ignore-comments` is non-`nil`.

If the scan reaches the beginning or end of the buffer (or its accessible portion), and the depth is not zero, an error is signaled. If the depth is zero but the count is not used up, `nil` is returned.

scan-sexps *from count* Function

This function scans forward *count* sexps from position *from*. It returns the position where the scan stops. If *count* is negative, the scan moves backwards.

Scanning ignores comments if `parse-sexp-ignore-comments` is non-`nil`.

If the scan reaches the beginning or end of (the accessible part of) the buffer while in the middle of a parenthetical grouping, an error is signaled. If it reaches the beginning or end between groupings but before count is used up, `nil` is returned.

parse-sexp-ignore-comments Variable

If the value is non-`nil`, then comments are treated as whitespace by the functions in this section and by `forward-sexp`.

In older Emacs versions, this feature worked only when the comment terminator is something like '`*/`', and appears only to end a comment. In languages where newlines terminate comments, it was necessary make this variable `nil`, since not every newline is the end of a comment. This limitation no longer exists.

You can use `forward-comment` to move forward or backward over one comment or several comments.

forward-comment *count* Function

This function moves point forward across *count* comments (backward, if *count* is negative). If it finds anything other than a comment or whitespace, it stops, leaving point at the place where it stopped. It also stops after satisfying *count*.

To move forward over all comments and whitespace following point, use `(forward-comment (buffer-size))`. `(buffer-size)` is a good argument to use, because the number of comments in the buffer cannot exceed that many.

34.7 Some Standard Syntax Tables

Most of the major modes in Emacs have their own syntax tables. Here are several of them:

standard-syntax-table Function

This function returns the standard syntax table, which is the syntax table used in Fundamental mode.

text-mode-syntax-table Variable
The value of this variable is the syntax table used in Text mode.

c-mode-syntax-table Variable
The value of this variable is the syntax table for C-mode buffers.

emacs-lisp-mode-syntax-table Variable
The value of this variable is the syntax table used in Emacs Lisp mode by editing commands. (It has no effect on the Lisp `read` function.)

34.8 Syntax Table Internals

Lisp programs don't usually work with the elements directly; the Lisp-level syntax table functions usually work with syntax descriptors (see Section 34.2 [Syntax Descriptors], page 669). Nonetheless, here we document the internal format.

Each element of a syntax table is a cons cell of the form (*syntax-code* . *matching-char*). The CAR, *syntax-code*, is an integer that encodes the syntax class, and any flags. The CDR, *matching-char*, is non-`nil` if a character to match was specified.

This table gives the value of *syntax-code* which corresponds to each syntactic type.

Integer	*Class*	*Integer*	*Class*	*Integer*	*Class*
0	whitespace	5	close parenthesis	10	character quote
1	punctuation	6	expression prefix	11	comment-start
2	word	7	string quote	12	comment-end
3	symbol	8	paired delimiter	13	inherit
4	open parenthesis	9	escape	14	comment-fence
15	string-fence				

For example, the usual syntax value for '`(`' is `(4 . 41)`. (41 is the character code for '`)`'.)

The flags are encoded in higher order bits, starting 16 bits from the least significant bit. This table gives the power of two which corresponds to each syntax flag.

Prefix	*Flag*	*Prefix*	*Flag*	*Prefix*	*Flag*
'1'	`(lsh 1 16)`	'3'	`(lsh 1 18)`	'p'	`(lsh 1 20)`
'2'	`(lsh 1 17)`	'4'	`(lsh 1 19)`	'b'	`(lsh 1 21)`

34.9 Categories

Categories provide an alternate way of classifying characters syntactically. You can define several categories as needed, then independently assign each character to one or more categories. Unlike syntax classes, categories are

not mutually exclusive; it is normal for one character to belong to several categories.

Each buffer has a *category table* which records which categories are defined and also which characters belong to each category. Each category table defines its own categories, but normally these are initialized by copying from the standard categories table, so that the standard categories are available in all modes.

Each category has a name, which is an ASCII printing character in the range ' ' to '~'. You specify the name of a category when you define it with `define-category`.

The category table is actually a char-table (see Section 6.6 [Char-Tables], page 104). The element of the category table at index *c* is a *category set*—a bool-vector—that indicates which categories character *c* belongs to. In this category set, if the element at index *cat* is `t`, that means category *cat* is a member of the set, and that character *c* belongs to category *cat*.

define-category *char docstring* &optional *table* Function

This function defines a new category, with name *char* and documentation *docstring*.

The new category is defined for category table *table*, which defaults to the current buffer's category table.

category-docstring *category* &optional *table* Function

This function returns the documentation string of category *category* in category table *table*.

```
(category-docstring ?a)
     ⇒ "ASCII"
(category-docstring ?l)
     ⇒ "Latin"
```

get-unused-category *table* Function

This function returns a category name (a character) which is not currently defined in *table*. If all possible categories are in use in *table*, it returns `nil`.

category-table Function

This function returns the current buffer's category table.

category-table-p *object* Function

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

standard-category-table Function

This function returns the standard category table.

copy-category-table &optional *table* Function
This function constructs a copy of *table* and returns it. If *table* is not supplied (or is `nil`), it returns a copy of the current category table. Otherwise, an error is signaled if *table* is not a category table.

set-category-table *table* Function
This function makes *table* the category table for the current buffer. It returns *table*.

make-category-set *categories* Function
This function returns a new category set—a bool-vector—whose initial contents are the categories listed in the string *categories*. The elements of *categories* should be category names; the new category set has `t` for each of those categories, and `nil` for all other categories.

```
(make-category-set "al")
     ⇒ #&128"\0\0\0\0\0\0\0\0\0\0\0\0\2\20\0\0"
```

char-category-set *char* Function
This function returns the category set for character *char*. This is the bool-vector which records which categories the character *char* belongs to. The function `char-category-set` does not allocate storage, because it returns the same bool-vector that exists in the category table.

```
(char-category-set ?a)
     ⇒ #&128"\0\0\0\0\0\0\0\0\0\0\0\0\2\20\0\0"
```

category-set-mnemonics *category-set* Function
This function converts the category set *category-set* into a string containing the names of all the categories that are members of the set.

```
(category-set-mnemonics (char-category-set ?a))
     ⇒ "al"
```

modify-category-entry *character category* &optional *table reset* Function
This function modifies the category set of *character* in category table *table* (which defaults to the current buffer's category table).

Normally, it modifies the category set by adding *category* to it. But if *reset* is non-`nil`, then it deletes *category* instead.

35 Abbrevs And Abbrev Expansion

An abbreviation or *abbrev* is a string of characters that may be expanded to a longer string. The user can insert the abbrev string and find it replaced automatically with the expansion of the abbrev. This saves typing.

The set of abbrevs currently in effect is recorded in an *abbrev table*. Each buffer has a local abbrev table, but normally all buffers in the same major mode share one abbrev table. There is also a global abbrev table. Normally both are used.

An abbrev table is represented as an obarray containing a symbol for each abbreviation. The symbol's name is the abbreviation; its value is the expansion; its function definition is the hook function to do the expansion (see Section 35.3 [Defining Abbrevs], page 684); its property list cell contains the use count, the number of times the abbreviation has been expanded. Because these symbols are not interned in the usual obarray, they will never appear as the result of reading a Lisp expression; in fact, normally they are never used except by the code that handles abbrevs. Therefore, it is safe to use them in an extremely nonstandard way. See Section 7.3 [Creating Symbols], page 111.

For the user-level commands for abbrevs, see section "Abbrev Mode" in *The GNU Emacs Manual*.

35.1 Setting Up Abbrev Mode

Abbrev mode is a minor mode controlled by the value of the variable `abbrev-mode`.

abbrev-mode Variable

A non-`nil` value of this variable turns on the automatic expansion of abbrevs when their abbreviations are inserted into a buffer. If the value is `nil`, abbrevs may be defined, but they are not expanded automatically.

This variable automatically becomes buffer-local when set in any fashion.

default-abbrev-mode Variable

This is the value of `abbrev-mode` for buffers that do not override it. This is the same as `(default-value 'abbrev-mode)`.

35.2 Abbrev Tables

This section describes how to create and manipulate abbrev tables.

make-abbrev-table Function

This function creates and returns a new, empty abbrev table—an obarray containing no symbols. It is a vector filled with zeros.

clear-abbrev-table *table* Function
This function undefines all the abbrevs in abbrev table *table*, leaving it empty. The function returns `nil`.

define-abbrev-table *tabname definitions* Function
This function defines *tabname* (a symbol) as an abbrev table name, i.e., as a variable whose value is an abbrev table. It defines abbrevs in the table according to *definitions*, a list of elements of the form (*abbrevname expansion hook usecount*). The return value is always `nil`.

abbrev-table-name-list Variable
This is a list of symbols whose values are abbrev tables. `define-abbrev-table` adds the new abbrev table name to this list.

insert-abbrev-table-description *name* &optional *human* Function
This function inserts before point a description of the abbrev table named *name*. The argument *name* is a symbol whose value is an abbrev table. The return value is always `nil`.

If *human* is non-`nil`, the description is human-oriented. Otherwise the description is a Lisp expression—a call to `define-abbrev-table` that would define *name* exactly as it is currently defined.

35.3 Defining Abbrevs

These functions define an abbrev in a specified abbrev table. `define-abbrev` is the low-level basic function, while `add-abbrev` is used by commands that ask for information from the user.

add-abbrev *table type arg* Function
This function adds an abbreviation to abbrev table *table* based on information from the user. The argument *type* is a string describing in English the kind of abbrev this will be (typically, `"global"` or `"mode-specific"`); this is used in prompting the user. The argument *arg* is the number of words in the expansion.

The return value is the symbol that internally represents the new abbrev, or `nil` if the user declines to confirm redefining an existing abbrev.

define-abbrev *table name expansion hook* Function
This function defines an abbrev named *name*, in *table*, to expand to *expansion* and call *hook*. The return value is a symbol that represents the abbrev inside Emacs; its name is *name*.

The argument *name* should be a string. The argument *expansion* is normally the desired expansion (a string), or `nil` to undefine the abbrev.

If it is anything but a string or `nil`, then the abbreviation "expands" solely by running *hook*.

The argument *hook* is a function or `nil`. If *hook* is non-`nil`, then it is called with no arguments after the abbrev is replaced with *expansion*; point is located at the end of *expansion* when *hook* is called.

The use count of the abbrev is initialized to zero.

only-global-abbrevs User Option

If this variable is non-`nil`, it means that the user plans to use global abbrevs only. This tells the commands that define mode-specific abbrevs to define global ones instead. This variable does not alter the behavior of the functions in this section; it is examined by their callers.

35.4 Saving Abbrevs in Files

A file of saved abbrev definitions is actually a file of Lisp code. The abbrevs are saved in the form of a Lisp program to define the same abbrev tables with the same contents. Therefore, you can load the file with `load` (see Section 14.1 [How Programs Do Loading], page 211). However, the function `quietly-read-abbrev-file` is provided as a more convenient interface.

User-level facilities such as `save-some-buffers` can save abbrevs in a file automatically, under the control of variables described here.

abbrev-file-name User Option

This is the default file name for reading and saving abbrevs.

quietly-read-abbrev-file *filename* Function

This function reads abbrev definitions from a file named *filename*, previously written with `write-abbrev-file`. If *filename* is `nil`, the file specified in `abbrev-file-name` is used. `save-abbrevs` is set to `t` so that changes will be saved.

This function does not display any messages. It returns `nil`.

save-abbrevs User Option

A non-`nil` value for `save-abbrev` means that Emacs should save abbrevs when files are saved. `abbrev-file-name` specifies the file to save the abbrevs in.

abbrevs-changed Variable

This variable is set non-`nil` by defining or altering any abbrevs. This serves as a flag for various Emacs commands to offer to save your abbrevs.

write-abbrev-file *filename* Command

Save all abbrev definitions, in all abbrev tables, in the file *filename*, in the form of a Lisp program that when loaded will define the same abbrevs. This function returns `nil`.

35.5 Looking Up and Expanding Abbreviations

Abbrevs are usually expanded by certain interactive commands, including `self-insert-command`. This section describes the subroutines used in writing such commands, as well as the variables they use for communication.

abbrev-symbol *abbrev* &optional *table* Function
This function returns the symbol representing the abbrev named *abbrev*. The value returned is `nil` if that abbrev is not defined. The optional second argument *table* is the abbrev table to look it up in. If *table* is `nil`, this function tries first the current buffer's local abbrev table, and second the global abbrev table.

abbrev-expansion *abbrev* &optional *table* Function
This function returns the string that *abbrev* would expand into (as defined by the abbrev tables used for the current buffer). The optional argument *table* specifies the abbrev table to use, as in `abbrev-symbol`.

expand-abbrev Command
This command expands the abbrev before point, if any. If point does not follow an abbrev, this command does nothing. The command returns `t` if it did expansion, `nil` otherwise.

abbrev-prefix-mark &optional *arg* Command
Mark current point as the beginning of an abbrev. The next call to `expand-abbrev` will use the text from here to point (where it is then) as the abbrev to expand, rather than using the previous word as usual.

abbrev-all-caps User Option
When this is set non-`nil`, an abbrev entered entirely in upper case is expanded using all upper case. Otherwise, an abbrev entered entirely in upper case is expanded by capitalizing each word of the expansion.

abbrev-start-location Variable
This is the buffer position for `expand-abbrev` to use as the start of the next abbrev to be expanded. (`nil` means use the word before point instead.) `abbrev-start-location` is set to `nil` each time `expand-abbrev` is called. This variable is also set by `abbrev-prefix-mark`.

abbrev-start-location-buffer Variable
The value of this variable is the buffer for which `abbrev-start-location` has been set. Trying to expand an abbrev in any other buffer clears `abbrev-start-location`. This variable is set by `abbrev-prefix-mark`.

last-abbrev Variable

This is the `abbrev-symbol` of the most recent abbrev expanded. This information is left by `expand-abbrev` for the sake of the `unexpand-abbrev` command (see section "Expanding Abbrevs" in *The GNU Emacs Manual*).

last-abbrev-location Variable

This is the location of the most recent abbrev expanded. This contains information left by `expand-abbrev` for the sake of the `unexpand-abbrev` command.

last-abbrev-text Variable

This is the exact expansion text of the most recent abbrev expanded, after case conversion (if any). Its value is `nil` if the abbrev has already been unexpanded. This contains information left by `expand-abbrev` for the sake of the `unexpand-abbrev` command.

pre-abbrev-expand-hook Variable

This is a normal hook whose functions are executed, in sequence, just before any expansion of an abbrev. See Section 22.6 [Hooks], page 420. Since it is a normal hook, the hook functions receive no arguments. However, they can find the abbrev to be expanded by looking in the buffer before point.

The following sample code shows a simple use of `pre-abbrev-expand-hook`. If the user terminates an abbrev with a punctuation character, the hook function asks for confirmation. Thus, this hook allows the user to decide whether to expand the abbrev, and aborts expansion if it is not confirmed.

```
(add-hook 'pre-abbrev-expand-hook 'query-if-not-space)

;; This is the function invoked by pre-abbrev-expand-hook.

;; If the user terminated the abbrev with a space, the function does
;; nothing (that is, it returns so that the abbrev can expand).  If the
;; user entered some other character, this function asks whether
;; expansion should continue.

;; If the user answers the prompt with y, the function returns
;; nil (because of the not function), but that is
;; acceptable; the return value has no effect on expansion.

(defun query-if-not-space ()
  (if (/= ?\  (preceding-char))
      (if (not (y-or-n-p "Do you want to expand this abbrev? "))
          (error "Not expanding this abbrev"))))
```

35.6 Standard Abbrev Tables

Here we list the variables that hold the abbrev tables for the preloaded major modes of Emacs.

global-abbrev-table Variable
This is the abbrev table for mode-independent abbrevs. The abbrevs defined in it apply to all buffers. Each buffer may also have a local abbrev table, whose abbrev definitions take precedence over those in the global table.

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

fundamental-mode-abbrev-table Variable
This is the local abbrev table used in Fundamental mode; in other words, it is the local abbrev table in all buffers in Fundamental mode.

text-mode-abbrev-table Variable
This is the local abbrev table used in Text mode.

lisp-mode-abbrev-table Variable
This is the local abbrev table used in Lisp mode and Emacs Lisp mode.

36 Processes

In the terminology of operating systems, a *process* is a space in which a program can execute. Emacs runs in a process. Emacs Lisp programs can invoke other programs in processes of their own. These are called *subprocesses* or *child processes* of the Emacs process, which is their *parent process*.

A subprocess of Emacs may be *synchronous* or *asynchronous*, depending on how it is created. When you create a synchronous subprocess, the Lisp program waits for the subprocess to terminate before continuing execution. When you create an asynchronous subprocess, it can run in parallel with the Lisp program. This kind of subprocess is represented within Emacs by a Lisp object which is also called a "process". Lisp programs can use this object to communicate with the subprocess or to control it. For example, you can send signals, obtain status information, receive output from the process, or send input to it.

processp *object* Function

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

36.1 Functions that Create Subprocesses

There are three functions that create a new subprocess in which to run a program. One of them, `start-process`, creates an asynchronous process and returns a process object (see Section 36.4 [Asynchronous Processes], page 694). The other two, `call-process` and `call-process-region`, create a synchronous process and do not return a process object (see Section 36.3 [Synchronous Processes], page 691).

Synchronous and asynchronous processes are explained in following sections. Since the three functions are all called in a similar fashion, their common arguments are described here.

In all cases, the function's *program* argument specifies the program to be run. An error is signaled if the file is not found or cannot be executed. If the file name is relative, the variable `exec-path` contains a list of directories to search. Emacs initializes `exec-path` when it starts up, based on the value of the environment variable `PATH`. The standard file name constructs, '`~`', '`.`', and '`..`', are interpreted as usual in `exec-path`, but environment variable substitutions ('`$HOME`', etc.) are not recognized; use `substitute-in-file-name` to perform them (see Section 24.8.4 [File Name Expansion], page 454).

Each of the subprocess-creating functions has a *buffer-or-name* argument which specifies where the standard output from the program will go. It should be a buffer or a buffer name; if it is a buffer name, that will create the buffer if it does not already exist. It can also be `nil`, which says to discard the output unless a filter function handles it. (See Section 36.9.2 [Filter Functions], page 703, and Chapter 18 [Read and Print], page 283.)

Normally, you should avoid having multiple processes send output to the same buffer because their output would be intermixed randomly.

All three of the subprocess-creating functions have a `&rest` argument, *args*. The *args* must all be strings, and they are supplied to *program* as separate command line arguments. Wildcard characters and other shell constructs have no special meanings in these strings, since the whole strings are passed directly to the specified program.

Please note: The argument *program* contains only the name of the program; it may not contain any command-line arguments. You must use *args* to provide those.

The subprocess gets its current directory from the value of `default-directory` (see Section 24.8.4 [File Name Expansion], page 454).

The subprocess inherits its environment from Emacs, but you can specify overrides for it with `process-environment`. See Section 37.3 [System Environment], page 718.

exec-directory Variable

The value of this variable is the name of a directory (a string) that contains programs that come with GNU Emacs, programs intended for Emacs to invoke. The program `movemail` is an example of such a program; Rmail uses it to fetch new mail from an inbox.

exec-path User Option

The value of this variable is a list of directories to search for programs to run in subprocesses. Each element is either the name of a directory (i.e., a string), or `nil`, which stands for the default directory (which is the value of `default-directory`).

The value of `exec-path` is used by `call-process` and `start-process` when the *program* argument is not an absolute file name.

36.2 Shell Arguments

Lisp programs sometimes need to run a shell and give it a command which contains file names that were specified by the user. These programs ought to be able to support any valid file name. But the shell gives special treatment to certain characters, and if these characters occur in the file name, they will confuse the shell. To handle these characters, use the function `shell-quote-argument`:

shell-quote-argument *argument* Function

This function returns a string which represents, in shell syntax, an argument whose actual contents are *argument*. It should work reliably to concatenate the return value into a shell command and then pass it to a shell for execution.

Precisely what this function does depends on your operating system. The function is designed to work with the usual shell syntax; if you use an unusual shell, you will need to redefine this function. On MS-DOS, the function returns *argument* unchanged; while this is not really correct, it is the best one can do, since the MS-DOS shell has no quoting features.

```
;; This example shows the behavior on GNU and Unix systems.
(shell-quote-argument "foo > bar")
     ⇒ "foo\\ \\>\\ bar"
```

Here's an example of using `shell-quote-argument` to construct a shell command:

```
(concat "diff -c "
        (shell-quote-argument oldfile)
        " "
        (shell-quote-argument newfile))
```

36.3 Creating a Synchronous Process

After a *synchronous process* is created, Emacs waits for the process to terminate before continuing. Starting Dired is an example of this: it runs `ls` in a synchronous process, then modifies the output slightly. Because the process is synchronous, the entire directory listing arrives in the buffer before Emacs tries to do anything with it.

While Emacs waits for the synchronous subprocess to terminate, the user can quit by typing *C-g*. The first *C-g* tries to kill the subprocess with a `SIGINT` signal; but it waits until the subprocess actually terminates before quitting. If during that time the user types another *C-g*, that kills the subprocess instantly with `SIGKILL` and quits immediately. See Section 20.9 [Quitting], page 351.

The synchronous subprocess functions return an indication of how the process terminated.

The output from a synchronous subprocess is generally decoded using a coding system, much like text read from a file. The input sent to a subprocess by `call-process-region` is encoded using a coding system, much like text written into a file. See Section 32.10 [Coding Systems], page 636.

call-process *program* &optional *infile destination display* &rest *args* Function

This function calls *program* in a separate process and waits for it to finish.

The standard input for the process comes from file *infile* if *infile* is not `nil`, and from '`/dev/null`' otherwise. The argument *destination* says where to put the process output. Here are the possibilities:

a buffer
: Insert the output in that buffer, before point. This includes both the standard output stream and the standard error stream of the process.

a string
: Insert the output in a buffer with that name, before point.

`t`
: Insert the output in the current buffer, before point.

`nil`
: Discard the output.

0
: Discard the output, and return immediately without waiting for the subprocess to finish.

  In this case, the process is not truly synchronous, since it can run in parallel with Emacs; but you can think of it as synchronous in that Emacs is essentially finished with the subprocess as soon as this function returns.

(*real-destination error-destination*)
: Keep the standard output stream separate from the standard error stream; deal with the ordinary output as specified by *real-destination*, and dispose of the error output according to *error-destination*. If *error-destination* is `nil`, that means to discard the error output, `t` means mix it with the ordinary output, and a string specifies a file name to redirect error output into.

  You can't directly specify a buffer to put the error output in; that is too difficult to implement. But you can achieve this result by sending the error output to a temporary file and then inserting the file into a buffer.

If *display* is non-`nil`, then `call-process` redisplays the buffer as output is inserted. (However, if the coding system chosen for decoding output is `undecided`, meaning deduce the encoding from the actual data, then redisplay sometimes cannot continue once non-ASCII characters are encountered. There are fundamental reasons why it is hard to fix this.) Otherwise the function `call-process` does no redisplay, and the results become visible on the screen only when Emacs redisplays that buffer in the normal course of events.

The remaining arguments, *args*, are strings that specify command line arguments for the program.

The value returned by `call-process` (unless you told it not to wait) indicates the reason for process termination. A number gives the exit status of the subprocess; 0 means success, and any other value means failure. If the process terminated with a signal, `call-process` returns a string describing the signal.

In the examples below, the buffer '`foo`' is current.

```
(call-process "pwd" nil t)
       nil

---------- Buffer: foo ----------
/usr/user/lewis/manual
---------- Buffer: foo ----------
```

```
(call-process "grep" nil "bar" nil "lewis" "/etc/passwd")
        nil

---------- Buffer: bar ----------
lewis:5LTsHm66CSWKg:398:21:Bil Lewis:/user/lewis:/bin/csh

---------- Buffer: bar ----------
```

Here is a good example of the use of `call-process`, which used to be found in the definition of `insert-directory`:

```
(call-process insert-directory-program nil t nil switches
              (if full-directory-p
                  (concat (file-name-as-directory file) ".")
                file))
```

call-process-region *start end program* &optional *delete destination display* &rest *args* Function

This function sends the text between *start* to *end* as standard input to a process running *program*. It deletes the text sent if *delete* is non-`nil`; this is useful when *destination* is `t`, to insert the output in the current buffer in place of the input.

The arguments *destination* and *display* control what to do with the output from the subprocess, and whether to update the display as it comes in. For details, see the description of `call-process`, above. If *destination* is the integer 0, `call-process-region` discards the output and returns `nil` immediately, without waiting for the subprocess to finish.

The remaining arguments, *args*, are strings that specify command line arguments for the program.

The return value of `call-process-region` is just like that of `call-process`: `nil` if you told it to return without waiting; otherwise, a number or string which indicates how the subprocess terminated.

In the following example, we use `call-process-region` to run the `cat` utility, with standard input being the first five characters in buffer '`foo`' (the word '`input`'). `cat` copies its standard input into its standard output. Since the argument *destination* is `t`, this output is inserted in the current buffer.

```
---------- Buffer: foo ----------
input*
---------- Buffer: foo ----------
```

```
(call-process-region 1 6 "cat" nil t)
       nil

---------- Buffer: foo ----------
inputinput*
---------- Buffer: foo ----------
```

The `shell-command-on-region` command uses `call-process-region` like this:

```
(call-process-region
 start end
 shell-file-name      ; Name of program.
 nil                  ; Do not delete region.
 buffer               ; Send output to buffer.
 nil                  ; No redisplay during output.
 "-c" command)        ; Arguments for the shell.
```

shell-command-to-string *command* Function

This function executes *command* (a string) as a shell command, then returns the command's output as a string.

36.4 Creating an Asynchronous Process

After an *asynchronous process* is created, Emacs and the subprocess both continue running immediately. The process thereafter runs in parallel with Emacs, and the two can communicate with each other using the functions described in following sections. However, communication is only partially asynchronous: Emacs sends data to the process only when certain functions are called, and Emacs accepts data from the process only when Emacs is waiting for input or for a time delay.

Here we describe how to create an asynchronous process.

start-process *name buffer-or-name program* &rest *args* Function

This function creates a new asynchronous subprocess and starts the program *program* running in it. It returns a process object that stands for the new subprocess in Lisp. The argument *name* specifies the name for the process object; if a process with this name already exists, then *name* is modified (by appending '`<1>`', etc.) to be unique. The buffer *buffer-or-name* is the buffer to associate with the process.

The remaining arguments, *args*, are strings that specify command line arguments for the program.

In the example below, the first process is started and runs (rather, sleeps) for 100 seconds. Meanwhile, the second process is started, and given the name '`my-process<1>`' for the sake of uniqueness. It inserts the directory listing at the end of the buffer '`foo`', before the first process finishes. Then

it finishes, and a message to that effect is inserted in the buffer. Much later, the first process finishes, and another message is inserted in the buffer for it.

```
(start-process "my-process" "foo" "sleep" "100")
        #<process my-process>

(start-process "my-process" "foo" "ls" "-l" "/user/lewis/bin")
        #<process my-process<1>>

---------- Buffer: foo ----------
total 2
lrwxrwxrwx  1 lewis     14 Jul 22 10:12 gnuemacs --> /emacs
-rwxrwxrwx  1 lewis     19 Jul 30 21:02 lemon

Process my-process<1> finished

Process my-process finished
---------- Buffer: foo ----------
```

start-process-shell-command *name buffer-or-name command* &rest *command-args* Function

This function is like `start-process` except that it uses a shell to execute the specified command. The argument *command* is a shell command name, and *command-args* are the arguments for the shell command. The variable `shell-file-name` specifies which shell to use.

The point of running a program through the shell, rather than directly with `start-process`, is so that you can employ shell features such as wildcards in the arguments. It follows that if you include an arbitrary user-specified filename in the command, you should quote it with `shell-quote-argument` first, so that any special shell characters in the file name do *not* have their special shell meanings. See Section 36.2 [Shell Arguments], page 690.

process-connection-type Variable

This variable controls the type of device used to communicate with asynchronous subprocesses. If it is non-`nil`, then PTYs are used, when available. Otherwise, pipes are used.

PTYs are usually preferable for processes visible to the user, as in Shell mode, because they allow job control (`C-c`, `C-z`, etc.) to work between the process and its children, whereas pipes do not. For subprocesses used for internal purposes by programs, it is often better to use a pipe, because they are more efficient. In addition, the total number of PTYs is limited on many systems and it is good not to waste them.

The value `process-connection-type` is used when `start-process` is called. So you can specify how to communicate with one subprocess by binding the variable around the call to `start-process`.

```
(let ((process-connection-type nil))  ; Use a pipe.
  (start-process ...))
```

To determine whether a given subprocess actually got a pipe or a PTY, use the function `process-tty-name` (see Section 36.6 [Process Information], page 697).

36.5 Deleting Processes

Deleting a process disconnects Emacs immediately from the subprocess, and removes it from the list of active processes. It sends a signal to the subprocess to make the subprocess terminate, but this is not guaranteed to happen immediately. The process object itself continues to exist as long as other Lisp objects point to it. The process mark continues to point to the same place as before (usually into a buffer where output from the process was being inserted).

You can delete a process explicitly at any time. Processes are deleted automatically after they terminate, but not necessarily right away. If you delete a terminated process explicitly before it is deleted automatically, no harm results.

delete-exited-processes User Option

This variable controls automatic deletion of processes that have terminated (due to calling `exit` or to a signal). If it is `nil`, then they continue to exist until the user runs `list-processes`. Otherwise, they are deleted immediately after they exit.

delete-process *name* Function

This function deletes the process associated with *name*, killing it with a `SIGHUP` signal. The argument *name* may be a process, the name of a process, a buffer, or the name of a buffer.

```
(delete-process "*shell*")
       nil
```

process-kill-without-query *process* &optional *do-query* Function

This function specifies whether Emacs should query the user if *process* is still running when Emacs is exited. If *do-query* is `nil`, the process will be deleted silently. Otherwise, Emacs will query about killing it.

The value is `t` if the process was formerly set up to require query, `nil` otherwise. A newly-created process always requires query.

```
(process-kill-without-query (get-process "shell"))
       t
```

36.6 Process Information

Several functions return information about processes. `list-processes` is provided for interactive use.

list-processes Command
This command displays a listing of all living processes. In addition, it finally deletes any process whose status was '`Exited`' or '`Signaled`'. It returns `nil`.

process-list Function
This function returns a list of all processes that have not been deleted.

```
(process-list)
        (#<process display-time> #<process shell>)
```

get-process *name* Function
This function returns the process named *name*, or `nil` if there is none. An error is signaled if *name* is not a string.

```
(get-process "shell")
        #<process shell>
```

process-command *process* Function
This function returns the command that was executed to start *process*. This is a list of strings, the first string being the program executed and the rest of the strings being the arguments that were given to the program.

```
(process-command (get-process "shell"))
        ("/bin/csh" "-i")
```

process-id *process* Function
This function returns the PID of *process*. This is an integer that distinguishes the process *process* from all other processes running on the same computer at the current time. The PID of a process is chosen by the operating system kernel when the process is started and remains constant as long as the process exists.

process-name *process* Function
This function returns the name of *process*.

process-contact *process* Function
This function returns `t` for an ordinary child process, and (*hostname service*) for a net connection (see Section 36.12 [Network], page 708).

process-status *process-name* Function

This function returns the status of *process-name* as a symbol. The argument *process-name* must be a process, a buffer, a process name (string) or a buffer name (string).

The possible values for an actual subprocess are:

`run` for a process that is running.

`stop` for a process that is stopped but continuable.

`exit` for a process that has exited.

`signal` for a process that has received a fatal signal.

`open` for a network connection that is open.

`closed` for a network connection that is closed. Once a connection is closed, you cannot reopen it, though you might be able to open a new connection to the same place.

`nil` if *process-name* is not the name of an existing process.

```
(process-status "shell")
        run
(process-status (get-buffer "*shell*"))
        run
x
        #<process xx<1>>
(process-status x)
        exit
```

For a network connection, `process-status` returns one of the symbols `open` or `closed`. The latter means that the other side closed the connection, or Emacs did `delete-process`.

process-exit-status *process* Function

This function returns the exit status of *process* or the signal number that killed it. (Use the result of `process-status` to determine which of those it is.) If *process* has not yet terminated, the value is 0.

process-tty-name *process* Function

This function returns the terminal name that *process* is using for its communication with Emacs—or `nil` if it is using pipes instead of a terminal (see `process-connection-type` in Section 36.4 [Asynchronous Processes], page 694).

process-coding-system *process* Function

This function returns a cons cell describing the coding systems in use for decoding output from *process* and for encoding input to *process* (see Section 32.10 [Coding Systems], page 636). The value has this form:

(*coding-system-for-decoding* . *coding-system-for-encoding*)

set-process-coding-system *process decoding-system encoding-system* Function

This function specifies the coding systems to use for subsequent output from and input to *process*. It will use *decoding-system* to decode subprocess output, and *encoding-system* to encode subprocess input.

36.7 Sending Input to Processes

Asynchronous subprocesses receive input when it is sent to them by Emacs, which is done with the functions in this section. You must specify the process to send input to, and the input data to send. The data appears on the "standard input" of the subprocess.

Some operating systems have limited space for buffered input in a PTY. On these systems, Emacs sends an EOF periodically amidst the other characters, to force them through. For most programs, these EOFs do no harm.

Subprocess input is normally encoded using a coding system before the subprocess receives it, much like text written into a file. You can use `set-process-coding-system` to specify which coding system to use (see Section 36.6 [Process Information], page 697). Otherwise, the coding system comes from `coding-system-for-write`, if that is non-`nil`; or else from the defaulting mechanism (see Section 32.10.5 [Default Coding Systems], page 640).

process-send-string *process-name string* Function

This function sends *process-name* the contents of *string* as standard input. The argument *process-name* must be a process or the name of a process. If it is `nil`, the current buffer's process is used.

The function returns `nil`.

```
(process-send-string "shell<1>" "ls\n")
       nil


---------- Buffer: *shell* ----------
...
introduction.texi              syntax-tables.texi~
introduction.texi~             text.texi
introduction.txt               text.texi~
...
---------- Buffer: *shell* ----------
```

process-send-region *process-name start end* Command

This function sends the text in the region defined by *start* and *end* as standard input to *process-name*, which is a process or a process name. (If it is `nil`, the current buffer's process is used.)

An error is signaled unless both *start* and *end* are integers or markers that indicate positions in the current buffer. (It is unimportant which number is larger.)

process-send-eof &optional *process-name* Function

This function makes *process-name* see an end-of-file in its input. The EOF comes after any text already sent to it.

If *process-name* is not supplied, or if it is `nil`, then this function sends the EOF to the current buffer's process. An error is signaled if the current buffer has no process.

The function returns *process-name*.

```
(process-send-eof "shell")
        "shell"
```

36.8 Sending Signals to Processes

Sending a signal to a subprocess is a way of interrupting its activities. There are several different signals, each with its own meaning. The set of signals and their names is defined by the operating system. For example, the signal `SIGINT` means that the user has typed `C-c`, or that some analogous thing has happened.

Each signal has a standard effect on the subprocess. Most signals kill the subprocess, but some stop or resume execution instead. Most signals can optionally be handled by programs; if the program handles the signal, then we can say nothing in general about its effects.

You can send signals explicitly by calling the functions in this section. Emacs also sends signals automatically at certain times: killing a buffer sends a `SIGHUP` signal to all its associated processes; killing Emacs sends a `SIGHUP` signal to all remaining processes. (`SIGHUP` is a signal that usually indicates that the user hung up the phone.)

Each of the signal-sending functions takes two optional arguments: *process-name* and *current-group*.

The argument *process-name* must be either a process, the name of one, or `nil`. If it is `nil`, the process defaults to the process associated with the current buffer. An error is signaled if *process-name* does not identify a process.

The argument *current-group* is a flag that makes a difference when you are running a job-control shell as an Emacs subprocess. If it is non-`nil`, then the signal is sent to the current process-group of the terminal that Emacs uses to communicate with the subprocess. If the process is a job-control shell, this means the shell's current subjob. If it is `nil`, the signal is sent to the process group of the immediate subprocess of Emacs. If the subprocess is a job-control shell, this is the shell itself.

The flag *current-group* has no effect when a pipe is used to communicate with the subprocess, because the operating system does not support the distinction in the case of pipes. For the same reason, job-control shells won't work when a pipe is used. See `process-connection-type` in Section 36.4 [Asynchronous Processes], page 694.

interrupt-process &optional *process-name current-group* Function
This function interrupts the process *process-name* by sending the signal `SIGINT`. Outside of Emacs, typing the "interrupt character" (normally `C-c` on some systems, and `DEL` on others) sends this signal. When the argument *current-group* is non-`nil`, you can think of this function as "typing `C-c`" on the terminal by which Emacs talks to the subprocess.

kill-process &optional *process-name current-group* Function
This function kills the process *process-name* by sending the signal `SIGKILL`. This signal kills the subprocess immediately, and cannot be handled by the subprocess.

quit-process &optional *process-name current-group* Function
This function sends the signal `SIGQUIT` to the process *process-name*. This signal is the one sent by the "quit character" (usually `C-b` or `C-\`) when you are not inside Emacs.

stop-process &optional *process-name current-group* Function
This function stops the process *process-name* by sending the signal `SIGTSTP`. Use `continue-process` to resume its execution.

Outside of Emacs, on systems with job control, the "stop character" (usually `C-z`) normally sends this signal. When *current-group* is non-`nil`, you can think of this function as "typing `C-z`" on the terminal Emacs uses to communicate with the subprocess.

continue-process &optional *process-name current-group* Function
This function resumes execution of the process *process* by sending it the signal `SIGCONT`. This presumes that *process-name* was stopped previously.

signal-process *pid signal* Function
This function sends a signal to process *pid*, which need not be a child of Emacs. The argument *signal* specifies which signal to send; it should be an integer.

36.9 Receiving Output from Processes

There are two ways to receive the output that a subprocess writes to its standard output stream. The output can be inserted in a buffer, which is called the associated buffer of the process, or a function called the *filter function* can be called to act on the output. If the process has no buffer and no filter function, its output is discarded.

Output from a subprocess can arrive only while Emacs is waiting: when reading terminal input, in `sit-for` and `sleep-for` (see Section 20.8 [Waiting], page 350), and in `accept-process-output` (see Section 36.9.3 [Accepting Output], page 705). This minimizes the problem of timing errors that usually plague parallel programming. For example, you can safely create a process and only then specify its buffer or filter function; no output can arrive before you finish, if the code in between does not call any primitive that waits.

Subprocess output is normally decoded using a coding system before the buffer or filter function receives it, much like text read from a file. You can use `set-process-coding-system` to specify which coding system to use (see Section 36.6 [Process Information], page 697). Otherwise, the coding system comes from `coding-system-for-read`, if that is non-`nil`; or else from the defaulting mechanism (see Section 32.10.5 [Default Coding Systems], page 640).

Warning: Coding systems such as `undecided` which determine the coding system from the data do not work entirely reliably with asynchronous subprocess output. This is because Emacs has to process asynchronous subprocess output in batches, as it arrives. Emacs must try to detect the proper coding system from one batch at a time, and this does not always work. Therefore, if at all possible, use a coding system which determines both the character code conversion and the end of line conversion—that is, one like `latin-1-unix`, rather than `undecided` or `latin-1`.

36.9.1 Process Buffers

A process can (and usually does) have an *associated buffer*, which is an ordinary Emacs buffer that is used for two purposes: storing the output from the process, and deciding when to kill the process. You can also use the buffer to identify a process to operate on, since in normal practice only one process is associated with any given buffer. Many applications of processes also use the buffer for editing input to be sent to the process, but this is not built into Emacs Lisp.

Unless the process has a filter function (see Section 36.9.2 [Filter Functions], page 703), its output is inserted in the associated buffer. The position to insert the output is determined by the `process-mark`, which is then updated to point to the end of the text just inserted. Usually, but not always, the `process-mark` is at the end of the buffer.

process-buffer *process* Function

This function returns the associated buffer of the process *process*.

```
(process-buffer (get-process "shell"))
        #<buffer *shell*>
```

process-mark *process* Function

This function returns the process marker for *process*, which is the marker that says where to insert output from the process.

If *process* does not have a buffer, `process-mark` returns a marker that points nowhere.

Insertion of process output in a buffer uses this marker to decide where to insert, and updates it to point after the inserted text. That is why successive batches of output are inserted consecutively.

Filter functions normally should use this marker in the same fashion as is done by direct insertion of output in the buffer. A good example of a filter function that uses `process-mark` is found at the end of the following section.

When the user is expected to enter input in the process buffer for transmission to the process, the process marker separates the new input from previous output.

set-process-buffer *process buffer* Function

This function sets the buffer associated with *process* to *buffer*. If *buffer* is `nil`, the process becomes associated with no buffer.

get-buffer-process *buffer-or-name* Function

This function returns the process associated with *buffer-or-name*. If there are several processes associated with it, then one is chosen. (Currently, the one chosen is the one most recently created.) It is usually a bad idea to have more than one process associated with the same buffer.

```
(get-buffer-process "*shell*")
        #<process shell>
```

Killing the process's buffer deletes the process, which kills the subprocess with a `SIGHUP` signal (see Section 36.8 [Signals to Processes], page 700).

36.9.2 Process Filter Functions

A process *filter function* is a function that receives the standard output from the associated process. If a process has a filter, then *all* output from that process is passed to the filter. The process buffer is used directly for output from the process only when there is no filter.

The filter function can only be called when Emacs is waiting for something, because process output arrives only at such times. Emacs waits when

reading terminal input, in `sit-for` and `sleep-for` (see Section 20.8 [Waiting], page 350), and in `accept-process-output` (see Section 36.9.3 [Accepting Output], page 705).

A filter function must accept two arguments: the associated process and a string, which is output just received from it. The function is then free to do whatever it chooses with the output.

Quitting is normally inhibited within a filter function—otherwise, the effect of typing *C-g* at command level or to quit a user command would be unpredictable. If you want to permit quitting inside a filter function, bind `inhibit-quit` to `nil`. See Section 20.9 [Quitting], page 351.

If an error happens during execution of a filter function, it is caught automatically, so that it doesn't stop the execution of whatever program was running when the filter function was started. However, if `debug-on-error` is non-`nil`, the error-catching is turned off. This makes it possible to use the Lisp debugger to debug the filter function. See Section 17.1 [Debugger], page 247.

Many filter functions sometimes or always insert the text in the process's buffer, mimicking the actions of Emacs when there is no filter. Such filter functions need to use `set-buffer` in order to be sure to insert in that buffer. To avoid setting the current buffer semipermanently, these filter functions must save and restore the current buffer. They should also update the process marker, and in some cases update the value of point. Here is how to do these things:

```
(defun ordinary-insertion-filter (proc string)
  (with-current-buffer (process-buffer proc)
    (let ((moving (= (point) (process-mark proc))))
      (save-excursion
        ;; Insert the text, advancing the process marker.
        (goto-char (process-mark proc))
        (insert string)
        (set-marker (process-mark proc) (point)))
      (if moving (goto-char (process-mark proc))))))
```

The reason to use `with-current-buffer`, rather than using `save-excursion` to save and restore the current buffer, is so as to preserve the change in point made by the second call to `goto-char`.

To make the filter force the process buffer to be visible whenever new text arrives, insert the following line just before the `with-current-buffer` construct:

```
(display-buffer (process-buffer proc))
```

To force point to the end of the new output, no matter where it was previously, eliminate the variable `moving` and call `goto-char` unconditionally.

In earlier Emacs versions, every filter function that did regular expression searching or matching had to explicitly save and restore the match data. Now

Emacs does this automatically for filter functions; they never need to do it explicitly. See Section 33.6 [Match Data], page 660.

A filter function that writes the output into the buffer of the process should check whether the buffer is still alive. If it tries to insert into a dead buffer, it will get an error. The expression `(buffer-name (process-buffer` *process*`))` returns `nil` if the buffer is dead.

The output to the function may come in chunks of any size. A program that produces the same output twice in a row may send it as one batch of 200 characters one time, and five batches of 40 characters the next. If the filter looks for certain text strings in the subprocess output, make sure to handle the case where one of these strings is split across two or more batches of output.

set-process-filter *process filter* Function

This function gives *process* the filter function *filter*. If *filter* is `nil`, it gives the process no filter.

process-filter *process* Function

This function returns the filter function of *process*, or `nil` if it has none.

Here is an example of use of a filter function:

```
(defun keep-output (process output)
   (setq kept (cons output kept)))
        keep-output
(setq kept nil)
        nil
(set-process-filter (get-process "shell") 'keep-output)
        keep-output
(process-send-string "shell" "ls ~/other\n")
        nil
kept
        ("lewis@slug[8] % "
"FINAL-W87-SHORT.MSS    backup.otl              kolstad.mss~
address.txt             backup.psf              kolstad.psf
backup.bib~             david.mss               resume-Dec-86.mss~
backup.err              david.psf               resume-Dec.psf
backup.mss              dland                   syllabus.mss
"
"#backups.mss#          backup.mss~             kolstad.mss
")
```

36.9.3 Accepting Output from Processes

Output from asynchronous subprocesses normally arrives only while Emacs is waiting for some sort of external event, such as elapsed time or

terminal input. Occasionally it is useful in a Lisp program to explicitly permit output to arrive at a specific point, or even to wait until output arrives from a process.

accept-process-output &optional *process seconds millisec* Function

This function allows Emacs to read pending output from processes. The output is inserted in the associated buffers or given to their filter functions. If *process* is non-`nil` then this function does not return until some output has been received from *process*.

The arguments *seconds* and *millisec* let you specify timeout periods. The former specifies a period measured in seconds and the latter specifies one measured in milliseconds. The two time periods thus specified are added together, and `accept-process-output` returns after that much time whether or not there has been any subprocess output.

The argument *seconds* need not be an integer. If it is a floating point number, this function waits for a fractional number of seconds. Some systems support only a whole number of seconds; on these systems, *seconds* is rounded down.

Not all operating systems support waiting periods other than multiples of a second; on those that do not, you get an error if you specify nonzero *millisec*.

The function `accept-process-output` returns non-`nil` if it did get some output, or `nil` if the timeout expired before output arrived.

36.10 Sentinels: Detecting Process Status Changes

A *process sentinel* is a function that is called whenever the associated process changes status for any reason, including signals (whether sent by Emacs or caused by the process's own actions) that terminate, stop, or continue the process. The process sentinel is also called if the process exits. The sentinel receives two arguments: the process for which the event occurred, and a string describing the type of event.

The string describing the event looks like one of the following:

- "`finished\n`".
- "`exited abnormally with code` *exitcode*`\n`".
- "*name-of-signal*`\n`".
- "*name-of-signal* `(core dumped)\n`".

A sentinel runs only while Emacs is waiting (e.g., for terminal input, or for time to elapse, or for process output). This avoids the timing errors that could result from running them at random places in the middle of other Lisp programs. A program can wait, so that sentinels will run, by calling

`sit-for` or `sleep-for` (see Section 20.8 [Waiting], page 350), or `accept-process-output` (see Section 36.9.3 [Accepting Output], page 705). Emacs also allows sentinels to run when the command loop is reading input.

Quitting is normally inhibited within a sentinel—otherwise, the effect of typing `C-g` at command level or to quit a user command would be unpredictable. If you want to permit quitting inside a sentinel, bind `inhibit-quit` to `nil`. See Section 20.9 [Quitting], page 351.

A sentinel that writes the output into the buffer of the process should check whether the buffer is still alive. If it tries to insert into a dead buffer, it will get an error. If the buffer is dead, `(buffer-name (process-buffer` *process*`))` returns `nil`.

If an error happens during execution of a sentinel, it is caught automatically, so that it doesn't stop the execution of whatever programs was running when the sentinel was started. However, if `debug-on-error` is non-`nil`, the error-catching is turned off. This makes it possible to use the Lisp debugger to debug the sentinel. See Section 17.1 [Debugger], page 247.

In earlier Emacs versions, every sentinel that did regular expression searching or matching had to explicitly save and restore the match data. Now Emacs does this automatically for sentinels; they never need to do it explicitly. See Section 33.6 [Match Data], page 660.

set-process-sentinel *process sentinel* Function

This function associates *sentinel* with *process*. If *sentinel* is `nil`, then the process will have no sentinel. The default behavior when there is no sentinel is to insert a message in the process's buffer when the process status changes.

```
(defun msg-me (process event)
   (princ
     (format "Process: %s had the event `%s'" process event)))
(set-process-sentinel (get-process "shell") 'msg-me)
       msg-me
(kill-process (get-process "shell"))
       Process: #<process shell> had the event `killed'
       #<process shell>
```

process-sentinel *process* Function

This function returns the sentinel of *process*, or `nil` if it has none.

waiting-for-user-input-p Function

While a sentinel or filter function is running, this function returns non-`nil` if Emacs was waiting for keyboard input from the user at the time the sentinel or filter function was called, `nil` if it was not.

36.11 Transaction Queues

You can use a *transaction queue* to communicate with a subprocess using transactions. First use `tq-create` to create a transaction queue communicating with a specified process. Then you can call `tq-enqueue` to send a transaction.

tq-create *process* Function

This function creates and returns a transaction queue communicating with *process*. The argument *process* should be a subprocess capable of sending and receiving streams of bytes. It may be a child process, or it may be a TCP connection to a server, possibly on another machine.

tq-enqueue *queue question regexp closure fn* Function

This function sends a transaction to queue *queue*. Specifying the queue has the effect of specifying the subprocess to talk to.

The argument *question* is the outgoing message that starts the transaction. The argument *fn* is the function to call when the corresponding answer comes back; it is called with two arguments: *closure*, and the answer received.

The argument *regexp* is a regular expression that should match the entire answer, but nothing less; that's how `tq-enqueue` determines where the answer ends.

The return value of `tq-enqueue` itself is not meaningful.

tq-close *queue* Function

Shut down transaction queue *queue*, waiting for all pending transactions to complete, and then terminate the connection or child process.

Transaction queues are implemented by means of a filter function. See Section 36.9.2 [Filter Functions], page 703.

36.12 Network Connections

Emacs Lisp programs can open TCP network connections to other processes on the same machine or other machines. A network connection is handled by Lisp much like a subprocess, and is represented by a process object. However, the process you are communicating with is not a child of the Emacs process, so you can't kill it or send it signals. All you can do is send and receive data. `delete-process` closes the connection, but does not kill the process at the other end; that process must decide what to do about closure of the connection.

You can distinguish process objects representing network connections from those representing subprocesses with the `process-status` function. It always returns either `open` or `closed` for a network connection, and it

never returns either of those values for a real subprocess. See Section 36.6 [Process Information], page 697.

open-network-stream *name buffer-or-name host service* Function

This function opens a TCP connection for a service to a host. It returns a process object to represent the connection.

The *name* argument specifies the name for the process object. It is modified as necessary to make it unique.

The *buffer-or-name* argument is the buffer to associate with the connection. Output from the connection is inserted in the buffer, unless you specify a filter function to handle the output. If *buffer-or-name* is `nil`, it means that the connection is not associated with any buffer.

The arguments *host* and *service* specify where to connect to; *host* is the host name (a string), and *service* is the name of a defined network service (a string) or a port number (an integer).

37 Operating System Interface

This chapter is about starting and getting out of Emacs, access to values in the operating system environment, and terminal input, output, and flow control.

See Section B.1 [Building Emacs], page 793, for related information. See also Chapter 38 [Display], page 739, for additional operating system status information pertaining to the terminal and the screen.

37.1 Starting Up Emacs

This section describes what Emacs does when it is started, and how you can customize these actions.

37.1.1 Summary: Sequence of Actions at Start Up

The order of operations performed (in '`startup.el`') by Emacs when it is started up is as follows:

1. It adds subdirectories to `load-path`, by running the file named '`subdirs.el`' in each directory that is listed.
2. It sets the language environment and the terminal coding system, if requested by environment variables such as `LANG`.
3. It loads the initialization library for the window system, if you are using a window system. This library's name is '`term/`*windowsystem*`-win.el`'.
4. It processes the initial options. (Some of them are handled even earlier than this.)
5. It initializes the window frame and faces, if appropriate.
6. It runs the normal hook `before-init-hook`.
7. It loads the library '`site-start`', unless the option '`-no-site-file`' was specified. The library's file name is usually '`site-start.el`'.
8. It loads the file '`~/.emacs`', unless '`-q`' or '`-batch`' was specified on the command line. The '`-u`' option can specify another user name whose home directory should be used instead of '`~`'.
9. It loads the library '`default`', unless `inhibit-default-init` is non-`nil`. (This is not done in '`-batch`' mode or if '`-q`' was specified on the command line.) The library's file name is usually '`default.el`'.
10. It runs the normal hook `after-init-hook`.
11. It sets the major mode according to `initial-major-mode`, provided the buffer '`*scratch*`' is still current and still in Fundamental mode.
12. It loads the terminal-specific Lisp file, if any, except when in batch mode or using a window system.
13. It displays the initial echo area message, unless you have suppressed that with `inhibit-startup-echo-area-message`.

14. It processes the action arguments from the command line.
15. It runs `term-setup-hook`.
16. It calls `frame-notice-user-settings`, which modifies the parameters of the selected frame according to whatever the init files specify.
17. It runs `window-setup-hook`. See Section 38.16 [Window Systems], page 765.
18. It displays copyleft, nonwarranty, and basic use information, provided there were no remaining command line arguments (a few steps above), the value of `inhibit-startup-message` is `nil`, and the buffer is still empty.

inhibit-startup-message User Option

This variable inhibits the initial startup messages (the nonwarranty, etc.). If it is non-`nil`, then the messages are not printed.

This variable exists so you can set it in your personal init file, once you are familiar with the contents of the startup message. Do not set this variable in the init file of a new user, or in a way that affects more than one user, because that would prevent new users from receiving the information they are supposed to see.

inhibit-startup-echo-area-message User Option

This variable controls the display of the startup echo area message. You can suppress the startup echo area message by adding text with this form to your '`.emacs`' file:

```
(setq inhibit-startup-echo-area-message
      "your-login-name")
```

Emacs explicitly checks for an expression as shown above in your '`.emacs`' file; your login name must appear in the expression as a Lisp string constant. Other methods of setting `inhibit-startup-echo-area-message` to the same value do not inhibit the startup message.

This way, you can easily inhibit the message for yourself if you wish, but thoughtless copying of your '`.emacs`' file will not inhibit the message for someone else.

37.1.2 The Init File: '`.emacs`'

When you start Emacs, it normally attempts to load the file '`.emacs`' from your home directory. This file, if it exists, must contain Lisp code. It is called your *init file*. The command line switches '`-q`' and '`-u`' affect the use of the init file; '`-q`' says not to load an init file, and '`-u`' says to load a specified user's init file instead of yours. See section "Entering Emacs" in *The GNU Emacs Manual*.

A site may have a *default init file*, which is the library named '`default.el`'. Emacs finds the '`default.el`' file through the standard

search path for libraries (see Section 14.1 [How Programs Do Loading], page 211). The Emacs distribution does not come with this file; sites may provide one for local customizations. If the default init file exists, it is loaded whenever you start Emacs, except in batch mode or if '`-q`' is specified. But your own personal init file, if any, is loaded first; if it sets `inhibit-default-init` to a non-`nil` value, then Emacs does not subsequently load the '`default.el`' file.

Another file for site-customization is '`site-start.el`'. Emacs loads this *before* the user's init file. You can inhibit the loading of this file with the option '`-no-site-file`'.

site-run-file Variable

This variable specifies the site-customization file to load before the user's init file. Its normal value is `"site-start"`. The only way you can change it with real effect is to do so before dumping Emacs.

If there is a great deal of code in your '`.emacs`' file, you should move it into another file named '*something*`.el`', byte-compile it (see Chapter 15 [Byte Compilation], page 223), and make your '`.emacs`' file load the other file using `load` (see Chapter 14 [Loading], page 211).

See section "Init File Examples" in *The GNU Emacs Manual*, for examples of how to make various commonly desired customizations in your '`.emacs`' file.

inhibit-default-init User Option

This variable prevents Emacs from loading the default initialization library file for your session of Emacs. If its value is non-`nil`, then the default library is not loaded. The default value is `nil`.

before-init-hook Variable

This normal hook is run, once, just before loading all the init files (the user's init file, '`default.el`', and/or '`site-start.el`'). (The only way to change it with real effect is before dumping Emacs.)

after-init-hook Variable

This normal hook is run, once, just after loading all the init files (the user's init file, '`default.el`', and/or '`site-start.el`'), before the terminal-specific initialization.

37.1.3 Terminal-Specific Initialization

Each terminal type can have its own Lisp library that Emacs loads when run on that type of terminal. The library's name is constructed by concatenating the value of the variable `term-file-prefix` and the terminal type. Normally, `term-file-prefix` has the value `"term/"`; changing this is not

recommended. Emacs finds the file in the normal manner, by searching the `load-path` directories, and trying the '`.elc`' and '`.el`' suffixes.

The usual function of a terminal-specific library is to enable special keys to send sequences that Emacs can recognize. It may also need to set or add to `function-key-map` if the Termcap entry does not specify all the terminal's function keys. See Section 37.8 [Terminal Input], page 729.

When the name of the terminal type contains a hyphen, only the part of the name before the first hyphen is significant in choosing the library name. Thus, terminal types '`aaa-48`' and '`aaa-30-rv`' both use the '`term/aaa`' library. If necessary, the library can evaluate `(getenv "TERM")` to find the full name of the terminal type.

Your '`.emacs`' file can prevent the loading of the terminal-specific library by setting the variable `term-file-prefix` to `nil`. This feature is useful when experimenting with your own peculiar customizations.

You can also arrange to override some of the actions of the terminal-specific library by setting the variable `term-setup-hook`. This is a normal hook which Emacs runs using `run-hooks` at the end of Emacs initialization, after loading both your '`.emacs`' file and any terminal-specific libraries. You can use this variable to define initializations for terminals that do not have their own libraries. See Section 22.6 [Hooks], page 420.

term-file-prefix Variable

If the `term-file-prefix` variable is non-`nil`, Emacs loads a terminal-specific initialization file as follows:

```
(load (concat term-file-prefix (getenv "TERM")))
```

You may set the `term-file-prefix` variable to `nil` in your '`.emacs`' file if you do not wish to load the terminal-initialization file. To do this, put the following in your '`.emacs`' file: `(setq term-file-prefix nil)`.

term-setup-hook Variable

This variable is a normal hook that Emacs runs after loading your '`.emacs`' file, the default initialization file (if any) and the terminal-specific Lisp file.

You can use `term-setup-hook` to override the definitions made by a terminal-specific file.

See `window-setup-hook` in Section 38.16 [Window Systems], page 765, for a related feature.

37.1.4 Command Line Arguments

You can use command line arguments to request various actions when you start Emacs. Since you do not need to start Emacs more than once per day, and will often leave your Emacs session running longer than that, command line arguments are hardly ever used. As a practical matter, it is best to avoid

making the habit of using them, since this habit would encourage you to kill and restart Emacs unnecessarily often. These options exist for two reasons: to be compatible with other editors (for invocation by other programs) and to enable shell scripts to run specific Lisp programs.

This section describes how Emacs processes command line arguments, and how you can customize them.

command-line Function

This function parses the command line that Emacs was called with, processes it, loads the user's '`.emacs`' file and displays the startup messages.

command-line-processed Variable

The value of this variable is `t` once the command line has been processed.

If you redump Emacs by calling `dump-emacs`, you may wish to set this variable to `nil` first in order to cause the new dumped Emacs to process its new command line arguments.

command-switch-alist Variable

The value of this variable is an alist of user-defined command-line options and associated handler functions. This variable exists so you can add elements to it.

A *command line option* is an argument on the command line of the form:

`-`*option*

The elements of the `command-switch-alist` look like this:

(*option* `.` *handler-function*)

The *handler-function* is called to handle *option* and receives the option name as its sole argument.

In some cases, the option is followed in the command line by an argument. In these cases, the *handler-function* can find all the remaining command-line arguments in the variable `command-line-args-left`. (The entire list of command-line arguments is in `command-line-args`.)

The command line arguments are parsed by the `command-line-1` function in the '`startup.el`' file. See also section "Command Line Switches and Arguments" in *The GNU Emacs Manual*.

command-line-args Variable

The value of this variable is the list of command line arguments passed to Emacs.

command-line-functions Variable

This variable's value is a list of functions for handling an unrecognized command-line argument. Each time the next argument to be processed has no special meaning, the functions in this list are called, in order of appearance, until one of them returns a non-`nil` value.

These functions are called with no arguments. They can access the command-line argument under consideration through the variable `argi`, which is bound temporarily at this point. The remaining arguments (not including the current one) are in the variable `command-line-args-left`.

When a function recognizes and processes the argument in `argi`, it should return a non-`nil` value to say it has dealt with that argument. If it has also dealt with some of the following arguments, it can indicate that by deleting them from `command-line-args-left`.

If all of these functions return `nil`, then the argument is used as a file name to visit.

37.2 Getting Out of Emacs

There are two ways to get out of Emacs: you can kill the Emacs job, which exits permanently, or you can suspend it, which permits you to reenter the Emacs process later. As a practical matter, you seldom kill Emacs—only when you are about to log out. Suspending is much more common.

37.2.1 Killing Emacs

Killing Emacs means ending the execution of the Emacs process. The parent process normally resumes control. The low-level primitive for killing Emacs is `kill-emacs`.

kill-emacs &optional *exit-data* Function

This function exits the Emacs process and kills it.

If *exit-data* is an integer, then it is used as the exit status of the Emacs process. (This is useful primarily in batch operation; see Section 37.12 [Batch Mode], page 737.)

If *exit-data* is a string, its contents are stuffed into the terminal input buffer so that the shell (or whatever program next reads input) can read them.

All the information in the Emacs process, aside from files that have been saved, is lost when the Emacs is killed. Because killing Emacs inadvertently can lose a lot of work, Emacs queries for confirmation before actually terminating if you have buffers that need saving or subprocesses that are running. This is done in the function `save-buffers-kill-emacs`.

kill-emacs-query-functions Variable

After asking the standard questions, `save-buffers-kill-emacs` calls the functions in the list `kill-emacs-query-functions`, in order of appearance, with no arguments. These functions can ask for additional confirmation from the user. If any of them returns `nil`, Emacs is not killed.

kill-emacs-hook Variable

This variable is a normal hook; once `save-buffers-kill-emacs` is finished with all file saving and confirmation, it runs the functions in this hook.

37.2.2 Suspending Emacs

Suspending Emacs means stopping Emacs temporarily and returning control to its superior process, which is usually the shell. This allows you to resume editing later in the same Emacs process, with the same buffers, the same kill ring, the same undo history, and so on. To resume Emacs, use the appropriate command in the parent shell—most likely `fg`.

Some operating systems do not support suspension of jobs; on these systems, "suspension" actually creates a new shell temporarily as a subprocess of Emacs. Then you would exit the shell to return to Emacs.

Suspension is not useful with window systems, because the Emacs job may not have a parent that can resume it again, and in any case you can give input to some other job such as a shell merely by moving to a different window. Therefore, suspending is not allowed when Emacs is using a window system.

suspend-emacs *string* Function

This function stops Emacs and returns control to the superior process. If and when the superior process resumes Emacs, `suspend-emacs` returns `nil` to its caller in Lisp.

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

Before suspending, `suspend-emacs` runs the normal hook `suspend-hook`.

After the user resumes Emacs, `suspend-emacs` runs the normal hook `suspend-resume-hook`. See Section 22.6 [Hooks], page 420.

The next redisplay after resumption will redraw the entire screen, unless the variable `no-redraw-on-reenter` is non-`nil` (see Section 38.1 [Refresh Screen], page 739).

In the following example, note that '`pwd`' is not echoed after Emacs is suspended. But it is read and executed by the shell.

```
(suspend-emacs)
        nil
```

```
(add-hook 'suspend-hook
          (function (lambda ()
                      (or (y-or-n-p
                            "Really suspend? ")
                          (error "Suspend cancelled")))))
        (lambda nil
          (or (y-or-n-p "Really suspend? ")
              (error "Suspend cancelled")))
(add-hook 'suspend-resume-hook
          (function (lambda () (message "Resumed!"))))
        (lambda nil (message "Resumed!"))
(suspend-emacs "pwd")
        nil
---------- Buffer: Minibuffer ----------
Really suspend? y
---------- Buffer: Minibuffer ----------

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

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

suspend-hook Variable

This variable is a normal hook run before suspending.

suspend-resume-hook Variable

This variable is a normal hook run after suspending.

37.3 Operating System Environment

Emacs provides access to variables in the operating system environment through various functions. These variables include the name of the system, the user's UID, and so on.

system-configuration Variable

This variable holds the GNU configuration name for the hardware/software configuration of your system, as a string. The convenient way to test parts of this string is with `string-match`.

system-type Variable

The value of this variable is a symbol indicating the type of operating system Emacs is operating on. Here is a table of the possible values:

`alpha-vms`
: VMS on the Alpha.

`aix-v3`
: AIX.

`berkeley-unix`
: Berkeley BSD.

`dgux`
: Data General DGUX operating system.

`gnu`
: the GNU system (using the GNU kernel, which consists of the HURD and Mach).

`gnu/linux`
: A GNU/Linux system—that is, a variant GNU system, using the Linux kernel. (These systems are the ones people often call "Linux," but actually Linux is just the kernel, not the whole system.)

`hpux`
: Hewlett-Packard HPUX operating system.

`irix`
: Silicon Graphics Irix system.

`ms-dos`
: Microsoft MS-DOS "operating system."

`next-mach`
: NeXT Mach-based system.

`rtu`
: Masscomp RTU, UCB universe.

`unisoft-unix`
: UniSoft UniPlus.

`usg-unix-v`
: AT&T System V.

`vax-vms`
: VAX VMS.

`windows-nt`
: Microsoft windows NT.

`xenix`
: SCO Xenix 386.

We do not wish to add new symbols to make finer distinctions unless it is absolutely necessary! In fact, we hope to eliminate some of these alternatives in the future. We recommend using `system-configuration` to distinguish between different operating systems.

system-name Function

This function returns the name of the machine you are running on.

```
(system-name)
     ⇒ "www.gnu.org"
```

The symbol `system-name` is a variable as well as a function. In fact, the function returns whatever value the variable `system-name` currently holds. Thus, you can set the variable `system-name` in case Emacs is confused about the name of your system. The variable is also useful for constructing frame titles (see Section 28.4 [Frame Titles], page 534).

mail-host-address Variable

If this variable is non-`nil`, it is used instead of `system-name` for purposes of generating email addresses. For example, it is used when constructing the default value of `user-mail-address`. See Section 37.4 [User Identification], page 722. (Since this is done when Emacs starts up, the value actually used is the one saved when Emacs was dumped. See Section B.1 [Building Emacs], page 793.)

getenv *var* Function

This function returns the value of the environment variable *var*, as a string. Within Emacs, the environment variable values are kept in the Lisp variable `process-environment`.

```
(getenv "USER")
     ⇒ "lewis"

lewis@slug[10] % printenv
PATH=.:/user/lewis/bin:/usr/bin:/usr/local/bin
USER=lewis
TERM=ibmapa16
SHELL=/bin/csh
HOME=/user/lewis
```

setenv *variable value* Command

This command sets the value of the environment variable named *variable* to *value*. Both arguments should be strings. This function works by modifying `process-environment`; binding that variable with `let` is also reasonable practice.

process-environment Variable

This variable is a list of strings, each describing one environment variable. The functions `getenv` and `setenv` work by means of this variable.

```
process-environment
   ("l=/usr/stanford/lib/gnuemacs/lisp"
    "PATH=.:/user/lewis/bin:/usr/class:/nfsusr/local/bin"
    "USER=lewis"
    "TERM=ibmapa16"
    "SHELL=/bin/csh"
    "HOME=/user/lewis")
```

path-separator Variable
This variable holds a string which says which character separates directories in a search path (as found in an environment variable). Its value is ":" for Unix and GNU systems, and ";" for MS-DOS and Windows NT.

invocation-name Variable
This variable holds the program name under which Emacs was invoked. The value is a string, and does not include a directory name.

invocation-directory Variable
This variable holds the directory from which the Emacs executable was invoked, or perhaps `nil` if that directory cannot be determined.

installation-directory Variable
If non-`nil`, this is a directory within which to look for the '`lib-src`' and '`etc`' subdirectories. This is non-`nil` when Emacs can't find those directories in their standard installed locations, but can find them in a directory related somehow to the one containing the Emacs executable.

load-average &optional *use-float* Function
This function returns the current 1-minute, 5-minute, and 15-minute load averages, in a list.

By default, the values are integers that are 100 times the system load averages, which indicate the average number of processes trying to run. If *use-float* is non-`nil`, then they are returned as floating point numbers and without multiplying by 100.

```
(load-average)
     ⇒ (169 48 36)
(load-average t)
     ⇒ (1.69 0.48 0.36)

lewis@rocky[5] % uptime
 11:55am  up 1 day, 19:37,  3 users,
 load average: 1.69, 0.48, 0.36
```

emacs-pid Function
This function returns the process ID of the Emacs process.

tty-erase-char Variable
This variable holds the erase character that was selected in the system's terminal driver, before Emacs was started.

setprv *privilege-name* &optional *setp getprv* Function
This function sets or resets a VMS privilege. (It does not exist on Unix.) The first arg is the privilege name, as a string. The second argument,

setp, is `t` or `nil`, indicating whether the privilege is to be turned on or off. Its default is `nil`. The function returns `t` if successful, `nil` otherwise.

If the third argument, *getprv*, is non-`nil`, `setprv` does not change the privilege, but returns `t` or `nil` indicating whether the privilege is currently enabled.

37.4 User Identification

init-file-user Variable

This variable says which user's init files should be used by Emacs—or `nil` if none. The value reflects command line options such as '`-q`' or '`-u` *user*'.

Lisp packages that load files of customizations, or any other sort of user profile, should obey this variable in deciding where to find it. They should load the profile of the user name found in this variable. If `init-file-user` is `nil`, meaning that the '`-q`' option was used, then Lisp packages should not load any customization files or user profile.

user-mail-address Variable

This holds the nominal email address of the user who is using Emacs. Emacs normally sets this variable to a default value after reading your init files, but not if you have already set it. So you can set the variable to some other value in your '`~/.emacs`' file if you do not want to use the default value.

user-login-name &optional *uid* Function

If you don't specify *uid*, this function returns the name under which the user is logged in. If the environment variable `LOGNAME` is set, that value is used. Otherwise, if the environment variable `USER` is set, that value is used. Otherwise, the value is based on the effective UID, not the real UID.

If you specify *uid*, the value is the user name that corresponds to *uid* (which should be an integer).

```
(user-login-name)
     ⇒ "lewis"
```

user-real-login-name Function

This function returns the user name corresponding to Emacs's real UID. This ignores the effective UID and ignores the environment variables `LOGNAME` and `USER`.

user-full-name &optional *uid* Function

This function returns the full name of the logged-in user—or the value of the environment variables `NAME`, if that is set.

```
(user-full-name)
     ⇒ "Bil Lewis"
```

If *uid* is non-`nil`, then it should be an integer, a user-id, or a string, a login name. Then `user-full-name` returns the full name corresponding to that user-id or login name.

The symbols `user-login-name`, `user-real-login-name` and `user-full-name` are variables as well as functions. The functions return the same values that the variables hold. These variables allow you to "fake out" Emacs by telling the functions what to return. The variables are also useful for constructing frame titles (see Section 28.4 [Frame Titles], page 534).

user-real-uid Function

This function returns the real UID of the user.

```
(user-real-uid)
     ⇒ 19
```

user-uid Function

This function returns the effective UID of the user.

37.5 Time of Day

This section explains how to determine the current time and the time zone.

current-time-string &optional *time-value* Function

This function returns the current time and date as a human-readable string. The format of the string is unvarying; the number of characters used for each part is always the same, so you can reliably use `substring` to extract pieces of it. It is wise to count the characters from the beginning of the string rather than from the end, as additional information may some day be added at the end.

The argument *time-value*, if given, specifies a time to format instead of the current time. The argument should be a list whose first two elements are integers. Thus, you can use times obtained from `current-time` (see below) and from `file-attributes` (see Section 24.6.4 [File Attributes], page 446).

```
(current-time-string)
     ⇒ "Wed Oct 14 22:21:05 1987"
```

current-time Function

This function returns the system's time value as a list of three integers: (*high low microsec*). The integers *high* and *low* combine to give the number of seconds since 0:00 January 1, 1970, which is $high * 2^{16} + low$.

The third element, *microsec*, gives the microseconds since the start of the current second (or 0 for systems that return time only on the resolution of a second).

The first two elements can be compared with file time values such as you get with the function `file-attributes`. See Section 24.6.4 [File Attributes], page 446.

current-time-zone &optional *time-value* Function

This function returns a list describing the time zone that the user is in.

The value has the form `(`*offset name*`)`. Here *offset* is an integer giving the number of seconds ahead of UTC (east of Greenwich). A negative value means west of Greenwich. The second element, *name* is a string giving the name of the time zone. Both elements change when daylight savings time begins or ends; if the user has specified a time zone that does not use a seasonal time adjustment, then the value is constant through time.

If the operating system doesn't supply all the information necessary to compute the value, both elements of the list are `nil`.

The argument *time-value*, if given, specifies a time to analyze instead of the current time. The argument should be a cons cell containing two integers, or a list whose first two elements are integers. Thus, you can use times obtained from `current-time` (see above) and from `file-attributes` (see Section 24.6.4 [File Attributes], page 446).

37.6 Time Conversion

These functions convert time values (lists of two or three integers) to strings or to calendrical information. There is also a function to convert calendrical information to a time value. You can get time values from the functions `current-time` (see Section 37.5 [Time of Day], page 723) and `file-attributes` (see Section 24.6.4 [File Attributes], page 446).

Many operating systems are limited to time values that contain 32 bits of information; these systems typically handle only the times from 1901-12-13 20:45:52 UTC through 2038-01-19 03:14:07 UTC. However, some operating systems have larger time values, and can represent times far in the past or future.

Time conversion functions always use the Gregorian calendar, even for dates before the Gregorian calendar was introduced. Year numbers count the number of years since the year 1 B.C., and do not skip zero as traditional Gregorian years do; for example, the year number −37 represents the Gregorian year 38 B.C.

format-time-string *format-string time* Function

This function converts *time* to a string according to *format-string*. The argument *format-string* may contain '`%`'-sequences which say to substitute parts of the time. Here is a table of what the '`%`'-sequences mean:

'`%a`' This stands for the abbreviated name of the day of week.

'`%A`' This stands for the full name of the day of week.

'`%b`' This stands for the abbreviated name of the month.

'`%B`' This stands for the full name of the month.

'`%c`' This is a synonym for '`%x %X`'.

'`%C`' This has a locale-specific meaning. In the default locale (named C), it is equivalent to '`%A, %B %e, %Y`'.

'`%d`' This stands for the day of month, zero-padded.

'`%D`' This is a synonym for '`%m/%d/%y`'.

'`%e`' This stands for the day of month, blank-padded.

'`%h`' This is a synonym for '`%b`'.

'`%H`' This stands for the hour (00-23).

'`%I`' This stands for the hour (00-12).

'`%j`' This stands for the day of the year (001-366).

'`%k`' This stands for the hour (0-23), blank padded.

'`%l`' This stands for the hour (1-12), blank padded.

'`%m`' This stands for the month (01-12).

'`%M`' This stands for the minute (00-59).

'`%n`' This stands for a newline.

'`%p`' This stands for '`AM`' or '`PM`', as appropriate.

'`%r`' This is a synonym for '`%I:%M:%S %p`'.

'`%R`' This is a synonym for '`%H:%M`'.

'`%S`' This stands for the seconds (00-60).

'`%t`' This stands for a tab character.

'`%T`' This is a synonym for '`%H:%M:%S`'.

'`%U`' This stands for the week of the year (01-52), assuming that weeks start on Sunday.

'`%w`' This stands for the numeric day of week (0-6). Sunday is day 0.

'`%W`' This stands for the week of the year (01-52), assuming that weeks start on Monday.

'`%x`' This has a locale-specific meaning. In the default locale (named '`C`'), it is equivalent to '`%D`'.

'`%X`' This has a locale-specific meaning. In the default locale (named '`C`'), it is equivalent to '`%T`'.

'`%y`' This stands for the year without century (00-99).

'`%Y`' This stands for the year with century.

'`%Z`' This stands for the time zone abbreviation.

You can also specify the field width and type of padding for any of these '`%`'-sequences. This works as in `printf`: you write the field width as digits in the middle of a '`%`'-sequences. If you start the field width with '`0`', it means to pad with zeros. If you start the field width with '`_`', it means to pad with spaces.

For example, '`%S`' specifies the number of seconds since the minute; '`%03S`' means to pad this with zeros to 3 positions, '`%_3S`' to pad with spaces to 3 positions. Plain '`%3S`' pads with zeros, because that is how '`%S`' normally pads to two positions.

decode-time *time* Function

This function converts a time value into calendrical information. The return value is a list of nine elements, as follows:

(*seconds minutes hour day month year dow dst zone*)

Here is what the elements mean:

sec The number of seconds past the minute, as an integer between 0 and 59.

minute The number of minutes past the hour, as an integer between 0 and 59.

hour The hour of the day, as an integer between 0 and 23.

day The day of the month, as an integer between 1 and 31.

month The month of the year, as an integer between 1 and 12.

year The year, an integer typically greater than 1900.

dow The day of week, as an integer between 0 and 6, where 0 stands for Sunday.

dst `t` if daylight savings time is effect, otherwise `nil`.

zone An integer indicating the time zone, as the number of seconds east of Greenwich.

Common Lisp Note: Common Lisp has different meanings for *dow* and *zone*.

encode-time *seconds minutes hour day month year* &optional *...zone* Function

This function is the inverse of `decode-time`. It converts seven items of calendrical data into a time value. For the meanings of the arguments, see the table above under `decode-time`.

Year numbers less than 100 are treated just like other year numbers. If you want them to stand for years above 1900, you must alter them yourself before you call `encode-time`.

The optional argument *zone* defaults to the current time zone and its daylight savings time rules. If specified, it can be either a list (as you would get from `current-time-zone`), a string as in the `TZ` environment variable, or an integer (as you would get from `decode-time`). The specified zone is used without any further alteration for daylight savings time.

If you pass more than seven arguments to `encode-time`, the first six are used as *seconds* through *year*, the last argument is used as *zone*, and the arguments in between are ignored. This feature makes it possible to use the elements of a list returned by `decode-time` as the arguments to `encode-time`, like this:

```
(apply 'encode-time (decode-time ...))
```

You can perform simple date arithmetic by using out-of-range values for the *sec*, *minute*, *hour*, *day*, and *month* arguments; for example, day 0 means the day preceding the given month.

The operating system puts limits on the range of possible time values; if you try to encode a time that is out of range, an error results.

37.7 Timers for Delayed Execution

You can set up a *timer* to call a function at a specified future time or after a certain length of idleness.

Emacs cannot run timers at any arbitrary point in a Lisp program; it can run them only when Emacs could accept output from a subprocess: namely, while waiting or inside certain primitive functions such as `sit-for` or `read-event` which *can* wait. Therefore, a timer's execution may be delayed if Emacs is busy. However, the time of execution is very precise if Emacs is idle.

run-at-time *time repeat function* &rest *args* Function

This function arranges to call *function* with arguments *args* at time *time*. The argument *function* is a function to call later, and *args* are the arguments to give it when it is called. The time *time* is specified as a string.

Absolute times may be specified in a wide variety of formats; this function tries to accept all the commonly used date formats. Valid formats include these two,

year`-`*month*`-`*day* *hour*`:`*min*`:`*sec* *timezone*

hour`:`*min*`:`*sec* *timezone* *month*`/`*day*`/`*year*

where in both examples all fields are numbers; the format that `current-time-string` returns is also allowed, and many others as well.

To specify a relative time, use numbers followed by units. For example:

'`1 min`' denotes 1 minute from now.

'`1 min 5 sec`'
denotes 65 seconds from now.

'`1 min 2 sec 3 hour 4 day 5 week 6 fortnight 7 month 8 year`'
denotes exactly 103 months, 123 days, and 10862 seconds from now.

If *time* is a number (integer or floating point), that specifies a relative time measured in seconds.

The argument *repeat* specifies how often to repeat the call. If *repeat* is `nil`, there are no repetitions; *function* is called just once, at *time*. If *repeat* is a number, it specifies a repetition period measured in seconds.

In most cases, *repeat* has no effect on when *first* call takes place—*time* alone specifies that. There is one exception: if *time* is `t`, then the timer runs whenever the time is a multiple of *repeat* seconds after the epoch. This is useful for functions like `display-time`.

The function `run-at-time` returns a timer value that identifies the particular scheduled future action. You can use this value to call `cancel-timer` (see below).

with-timeout (*seconds timeout-forms...*) *body...* Macro

Execute *body*, but give up after *seconds* seconds. If *body* finishes before the time is up, `with-timeout` returns the value of the last form in *body*. If, however, the execution of *body* is cut short by the timeout, then `with-timeout` executes all the *timeout-forms* and returns the value of the last of them.

This macro works by setting a timer to run after *seconds* seconds. If *body* finishes before that time, it cancels the timer. If the timer actually runs, it terminates execution of *body*, then executes *timeout-forms*.

Since timers can run within a Lisp program only when the program calls a primitive that can wait, `with-timeout` cannot stop executing *body* while it is in the midst of a computation—only when it calls one of those primitives. So use `with-timeout` only with a *body* that waits for input, not one that does a long computation.

The function `y-or-n-p-with-timeout` provides a simple way to use a timer to avoid waiting too long for an answer. See Section 19.6 [Yes-or-No Queries], page 312.

run-with-idle-timer *secs repeat function* &rest *args* Function

Set up a timer which runs when Emacs has been idle for *secs* seconds. The value of *secs* may be an integer or a floating point number.

If *repeat* is `nil`, the timer runs just once, the first time Emacs remains idle for a long enough time. More often *repeat* is non-`nil`, which means to run the timer *each time* Emacs remains idle for *secs* seconds.

The function `run-with-idle-timer` returns a timer value which you can use in calling `cancel-timer` (see below).

Emacs becomes "idle" when it starts waiting for user input, and it remains idle until the user provides some input. If a timer is set for five seconds of idleness, it runs approximately five seconds after Emacs first became idle. Even if its *repeat* is true, this timer will not run again as long as Emacs remains idle, because the duration of idleness will continue to increase and will not go down to five seconds again.

Emacs can do various things while idle: garbage collect, autosave or handle data from a subprocess. But these interludes during idleness do not interfere with idle timers, because they do not reset the clock of idleness to zero. An idle timer set for 600 seconds will run when ten minutes have elapsed since the last user command was finished, even if subprocess output has been accepted thousands of times within those ten minutes, even if there have been garbage collections and autosaves.

When the user supplies input, Emacs becomes non-idle while executing the input. Then it becomes idle again, and all the idle timers that are set up to repeat will subsequently run another time, one by one.

cancel-timer *timer* Function

Cancel the requested action for *timer*, which should be a value previously returned by `run-at-time` or `run-with-idle-timer`. This cancels the effect of that call to `run-at-time`; the arrival of the specified time will not cause anything special to happen.

37.8 Terminal Input

This section describes functions and variables for recording or manipulating terminal input. See Chapter 38 [Display], page 739, for related functions.

37.8.1 Input Modes

set-input-mode *interrupt flow meta quit-char* Function

This function sets the mode for reading keyboard input. If *interrupt* is non-null, then Emacs uses input interrupts. If it is `nil`, then it uses CBREAK mode. The default setting is system dependent. Some systems always use CBREAK mode regardless of what is specified.

When Emacs communicates directly with X, it ignores this argument and uses interrupts if that is the way it knows how to communicate.

If *flow* is non-`nil`, then Emacs uses XON/XOFF (`C-q`, `C-s`) flow control for output to the terminal. This has no effect except in CBREAK mode. See Section 37.11 [Flow Control], page 736.

The argument *meta* controls support for input character codes above 127. If *meta* is `t`, Emacs converts characters with the 8th bit set into Meta characters. If *meta* is `nil`, Emacs disregards the 8th bit; this is necessary when the terminal uses it as a parity bit. If *meta* is neither `t` nor `nil`, Emacs uses all 8 bits of input unchanged. This is good for terminals that use 8-bit character sets.

If *quit-char* is non-`nil`, it specifies the character to use for quitting. Normally this character is `C-g`. See Section 20.9 [Quitting], page 351.

The `current-input-mode` function returns the input mode settings Emacs is currently using.

current-input-mode Function

This function returns current mode for reading keyboard input. It returns a list, corresponding to the arguments of `set-input-mode`, of the form (*interrupt flow meta quit*) in which:

interrupt is non-`nil` when Emacs is using interrupt-driven input. If `nil`, Emacs is using CBREAK mode.

flow is non-`nil` if Emacs uses XON/XOFF (`C-q`, `C-s`) flow control for output to the terminal. This value is meaningful only when *interrupt* is `nil`.

meta is `t` if Emacs treats the eighth bit of input characters as the meta bit; `nil` means Emacs clears the eighth bit of every input character; any other value means Emacs uses all eight bits as the basic character code.

quit is the character Emacs currently uses for quitting, usually `C-g`.

37.8.2 Translating Input Events

This section describes features for translating input events into other input events before they become part of key sequences. These features apply to each event in the order they are described here: each event is first modified according to `extra-keyboard-modifiers`, then translated through `keyboard-translate-table` (if applicable), and finally decoded with the specified keyboard coding system. If it is being read as part of a key sequence, it is then added to the sequence being read; then subsequences containing it are checked first with `function-key-map` and then with `key-translation-map`.

extra-keyboard-modifiers Variable

This variable lets Lisp programs "press" the modifier keys on the keyboard. The value is a bit mask:

1 The SHIFT key.

2 The LOCK key.

4 The CTL key.

8 The META key.

Each time the user types a keyboard key, it is altered as if the modifier keys specified in the bit mask were held down.

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.

keyboard-translate-table Variable

This variable is the translate table for keyboard characters. It lets you reshuffle the keys on the keyboard without changing any command bindings. Its value is normally a char-table, or else `nil`.

If `keyboard-translate-table` is a char-table, then each character read from the keyboard is looked up in this character. If the value found there is non-`nil`, then it is used instead of the actual input character.

In the example below, we set `keyboard-translate-table` to a char-table. Then we fill it in to swap the characters *C-s* and *C-* and the characters *C-q* and *C-^*. Subsequently, typing *C-* has all the usual effects of typing *C-s*, and vice versa. (See Section 37.11 [Flow Control], page 736 for more information on this subject.)

```
(defun evade-flow-control ()
  "Replace C-s with C-\ and C-q with C-^."
  (interactive)
  (setq keyboard-translate-table
        (make-char-table 'keyboard-translate-table nil))
  ;; Swap C-s and C-\.
  (aset keyboard-translate-table ?\034 ?\^s)
  (aset keyboard-translate-table ?\^s ?\034)
  ;; Swap C-q and C-^.
  (aset keyboard-translate-table ?\036 ?\^q)
  (aset keyboard-translate-table ?\^q ?\036))
```

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.

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.

The remaining translation features translate subsequences of key sequences being read. They are implemented in `read-key-sequence` and have no effect on input read with `read-event`.

function-key-map Variable

This variable holds a keymap that describes the character sequences sent by function keys on an ordinary character terminal. This keymap has the same structure as other keymaps, but is used differently: it specifies translations to make while reading key sequences, rather than bindings for key sequences.

If `function-key-map` "binds" a key sequence *k* to a vector *v*, then when *k* appears as a subsequence *anywhere* in a key sequence, it is replaced with the events in *v*.

For example, VT100 terminals send ESC *O P* when the keypad PF1 key is pressed. Therefore, we want Emacs to translate that sequence of events into the single event `pf1`. We accomplish this by "binding" ESC *O P* to `[pf1]` in `function-key-map`, when using a VT100.

Thus, typing *C-c* PF1 sends the character sequence *C-c* ESC *O P*; later the function `read-key-sequence` translates this back into *C-c* PF1, which it returns as the vector `[?\C-c pf1]`.

Entries in `function-key-map` are ignored if they conflict with bindings made in the minor mode, local, or global keymaps. The intent is that the character sequences that function keys send should not have command bindings in their own right—but if they do, the ordinary bindings take priority.

The value of `function-key-map` is usually set up automatically according to the terminal's Terminfo or Termcap entry, but sometimes those need help from terminal-specific Lisp files. Emacs comes with terminal-specific files for many common terminals; their main purpose is to make entries in `function-key-map` beyond those that can be deduced from Termcap and Terminfo. See Section 37.1.3 [Terminal-Specific], page 713.

key-translation-map Variable

This variable is another keymap used just like `function-key-map` to translate input events into other events. It differs from `function-key-map` in two ways:

- `key-translation-map` goes to work after `function-key-map` is finished; it receives the results of translation by `function-key-map`.
- `key-translation-map` overrides actual key bindings. For example, if *C-x f* has a binding in `key-translation-map`, that translation

takes effect even though *C-x f* also has a key binding in the global map.

The intent of `key-translation-map` is for users to map one character set to another, including ordinary characters normally bound to `self-insert-command`.

You can use `function-key-map` or `key-translation-map` for more than simple aliases, by using a function, instead of a key sequence, as the "translation" of a key. Then this function is called to compute the translation of that key.

The key translation function receives one argument, which is the prompt that was specified in `read-key-sequence`—or `nil` if the key sequence is being read by the editor command loop. In most cases you can ignore the prompt value.

If the function reads input itself, it can have the effect of altering the event that follows. For example, here's how to define *C-c h* to turn the character that follows into a Hyper character:

```
(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 function-key-map "\C-ch" 'hyperify)
```

Finally, if you have enabled keyboard character set decoding using `set-keyboard-coding-system`, decoding is done after the translations listed above. See Section 32.10.6 [Specifying Coding Systems], page 642. In future Emacs versions, character set decoding may be done before the other translations.

37.8.3 Recording Input

recent-keys Function

This function returns a vector containing the last 100 input events from the keyboard or mouse. All input events are included, whether or not they were used as parts of key sequences. Thus, you always get the last 100 input events, not counting events generated by keyboard macros. (These are excluded because they are less interesting for debugging; it should be enough to see the events that invoked the macros.)

open-dribble-file *filename* Command

This function opens a *dribble file* named *filename*. When a dribble file is open, each input event from the keyboard or mouse (but not those from keyboard macros) is written in that file. A non-character event is expressed using its printed representation surrounded by '`<...>`'.

You close the dribble file by calling this function with an argument of `nil`.

This function is normally used to record the input necessary to trigger an Emacs bug, for the sake of a bug report.

```
(open-dribble-file "~/dribble")
     ⇒ nil
```

See also the `open-termscript` function (see Section 37.9 [Terminal Output], page 734).

37.9 Terminal Output

The terminal output functions send output to the terminal or keep track of output sent to the terminal. The variable `baud-rate` tells you what Emacs thinks is the output speed of the terminal.

baud-rate Variable

This variable's value is the output speed of the terminal, as far as Emacs knows. Setting this variable does not change the speed of actual data transmission, but the value is used for calculations such as padding. It also affects decisions about whether to scroll part of the screen or repaint—even when using a window system. (We designed it this way despite the fact that a window system has no true "output speed", to give you a way to tune these decisions.)

The value is measured in baud.

If you are running across a network, and different parts of the network work at different baud rates, the value returned by Emacs may be different from the value used by your local terminal. Some network protocols communicate the local terminal speed to the remote machine, so that Emacs and other programs can get the proper value, but others do not. If Emacs has the wrong value, it makes decisions that are less than optimal. To fix the problem, set `baud-rate`.

baud-rate Function

This obsolete function returns the value of the variable `baud-rate`.

send-string-to-terminal *string* Function

This function sends *string* to the terminal without alteration. Control characters in *string* have terminal-dependent effects.

One use of this function is to define function keys on terminals that have downloadable function key definitions. For example, this is how on certain terminals to define function key 4 to move forward four characters (by transmitting the characters `C-u C-f` to the computer):

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

open-termscript *filename* Command

This function is used to open a *termscript file* that will record all the characters sent by Emacs to the terminal. It returns `nil`. Termscript files are useful for investigating problems where Emacs garbles the screen, problems that are due to incorrect Termcap entries or to undesirable settings of terminal options more often than to actual Emacs bugs. Once you are certain which characters were actually output, you can determine reliably whether they correspond to the Termcap specifications in use.

See also `open-dribble-file` in Section 37.8 [Terminal Input], page 729.

```
(open-termscript "../junk/termscript")
     ⇒ nil
```

37.10 System-Specific X11 Keysyms

To define system-specific X11 keysyms, set the variable `system-key-alist`.

system-key-alist Variable

This variable's value should be an alist with one element for each system-specific keysym. An element has this form: (*code* . *symbol*), where *code* is the numeric keysym code (not including the "vendor specific" bit, -2^{28}, and *symbol* is the name for the function key.

For example `(168 . mute-acute)` defines a system-specific key used by HP X servers whose numeric code is -2^{28} + 168.

It is not crucial to exclude from the alist the keysyms of other X servers; those do no harm, as long as they don't conflict with the ones used by the X server actually in use.

The variable is always local to the current terminal, and cannot be buffer-local. See Section 28.2 [Multiple Displays], page 526.

37.11 Flow Control

This section attempts to answer the question "Why does Emacs use flow-control characters in its command character set?" For a second view on this issue, read the comments on flow control in the '`emacs/INSTALL`' file from the distribution; for help with Termcap entries and DEC terminal concentrators, see '`emacs/etc/TERMS`'.

At one time, most terminals did not need flow control, and none used `C-s` and `C-q` for flow control. Therefore, the choice of `C-s` and `C-q` as command characters for searching and quoting was natural and uncontroversial. With so many commands needing key assignments, of course we assigned meanings to nearly all ASCII control characters.

Later, some terminals were introduced which required these characters for flow control. They were not very good terminals for full-screen editing, so Emacs maintainers ignored them. In later years, flow control with `C-s` and `C-q` became widespread among terminals, but by this time it was usually an option. And the majority of Emacs users, who can turn flow control off, did not want to switch to less mnemonic key bindings for the sake of flow control.

So which usage is "right"—Emacs's or that of some terminal and concentrator manufacturers? This question has no simple answer.

One reason why we are reluctant to cater to the problems caused by `C-s` and `C-q` is that they are gratuitous. There are other techniques (albeit less common in practice) for flow control that preserve transparency of the character stream. Note also that their use for flow control is not an official standard. Interestingly, on the model 33 teletype with a paper tape punch (around 1970), `C-s` and `C-q` were sent by the computer to turn the punch on and off!

As window systems and PC terminal emulators replace character-only terminals, the flow control problem is gradually disappearing. For the mean time, Emacs provides a convenient way of enabling flow control if you want it: call the function `enable-flow-control`.

enable-flow-control Command

This function enables use of `C-s` and `C-q` for output flow control, and provides the characters `C-\` and `C-^` as aliases for them using `keyboard-translate-table` (see Section 37.8.2 [Translating Input], page 730).

You can use the function `enable-flow-control-on` in your '`.emacs`' file to enable flow control automatically on certain terminal types.

enable-flow-control-on &rest *termtypes* Function

This function enables flow control, and the aliases `C-\` and `C-^`, if the terminal type is one of *termtypes*. For example:

```
(enable-flow-control-on "vt200" "vt300" "vt101" "vt131")
```

Here is how `enable-flow-control` does its job:

1. It sets CBREAK mode for terminal input, and tells the operating system to handle flow control, with `(set-input-mode nil t)`.
2. It sets up `keyboard-translate-table` to translate `C-\` and `C-^` into `C-s` and `C-q`. Except at its very lowest level, Emacs never knows that the characters typed were anything but `C-s` and `C-q`, so you can in effect type them as `C-\` and `C-^` even when they are input for other commands. See Section 37.8.2 [Translating Input], page 730.

If the terminal is the source of the flow control characters, then once you enable kernel flow control handling, you probably can make do with less padding than normal for that terminal. You can reduce the amount of padding by customizing the Termcap entry. You can also reduce it by setting `baud-rate` to a smaller value so that Emacs uses a smaller speed when calculating the padding needed. See Section 37.9 [Terminal Output], page 734.

37.12 Batch Mode

The command line option '`-batch`' causes Emacs to run noninteractively. In this mode, Emacs does not read commands from the terminal, it does not alter the terminal modes, and it does not expect to be outputting to an erasable screen. The idea is that you specify Lisp programs to run; when they are finished, Emacs should exit. The way to specify the programs to run is with '`-l` *file*', which loads the library named *file*, and '`-f` *function*', which calls *function* with no arguments.

Any Lisp program output that would normally go to the echo area, either using `message` or using `prin1`, etc., with `t` as the stream, goes instead to Emacs's standard error descriptor when in batch mode. Thus, Emacs behaves much like a noninteractive application program. (The echo area output that Emacs itself normally generates, such as command echoing, is suppressed entirely.)

noninteractive Variable

This variable is non-`nil` when Emacs is running in batch mode.

38 Emacs Display

This chapter describes a number of features related to the display that Emacs presents to the user.

38.1 Refreshing the Screen

The function `redraw-frame` redisplays the entire contents of a given frame (see Chapter 28 [Frames], page 525).

redraw-frame *frame* Function
This function clears and redisplays frame *frame*.

Even more powerful is `redraw-display`:

redraw-display Command
This function clears and redisplays all visible frames.

Processing user input takes absolute priority over redisplay. If you call these functions when input is available, they do nothing immediately, but a full redisplay does happen eventually—after all the input has been processed.

Normally, suspending and resuming Emacs also refreshes the screen. Some terminal emulators record separate contents for display-oriented programs such as Emacs and for ordinary sequential display. If you are using such a terminal, you might want to inhibit the redisplay on resumption.

no-redraw-on-reenter Variable
This variable controls whether Emacs redraws the entire screen after it has been suspended and resumed. Non-`nil` means there is no need to redraw, `nil` means redrawing is needed. The default is `nil`.

38.2 Truncation

When a line of text extends beyond the right edge of a window, the line can either be continued on the next screen line, or truncated to one screen line. The additional screen lines used to display a long text line are called *continuation* lines. Normally, a '`$`' in the rightmost column of the window indicates truncation; a '`\`' on the rightmost column indicates a line that "wraps" onto the next line, which is also called *continuing* the line. (The display table can specify alternative indicators; see Section 38.14 [Display Tables], page 762.)

Note that continuation is different from filling; continuation happens on the screen only, not in the buffer contents, and it breaks a line precisely at the right margin, not at a word boundary. See Section 31.11 [Filling], page 593.

truncate-lines User Option

This buffer-local variable controls how Emacs displays lines that extend beyond the right edge of the window. The default is `nil`, which specifies continuation. If the value is non-`nil`, then these lines are truncated.

If the variable `truncate-partial-width-windows` is non-`nil`, then truncation is always used for side-by-side windows (within one frame) regardless of the value of `truncate-lines`.

default-truncate-lines User Option

This variable is the default value for `truncate-lines`, for buffers that do not have buffer-local values for it.

truncate-partial-width-windows User Option

This variable controls display of lines that extend beyond the right edge of the window, in side-by-side windows (see Section 27.2 [Splitting Windows], page 496). If it is non-`nil`, these lines are truncated; otherwise, `truncate-lines` says what to do with them.

When horizontal scrolling (see Section 27.12 [Horizontal Scrolling], page 515) is in use in a window, that forces truncation.

You can override the glyphs that indicate continuation or truncation using the display table; see Section 38.14 [Display Tables], page 762.

If your buffer contains *very* long lines, and you use continuation to display them, just thinking about them can make Emacs redisplay slow. The column computation and indentation functions also become slow. Then you might find it advisable to set `cache-long-line-scans` to `t`.

cache-long-line-scans Variable

If this variable is non-`nil`, various indentation and motion functions, and Emacs redisplay, cache the results of scanning the buffer, and consult the cache to avoid rescanning regions of the buffer unless they are modified.

Turning on the cache slows down processing of short lines somewhat.

This variable is automatically buffer-local in every buffer.

38.3 The Echo Area

The *echo area* is used for displaying messages made with the `message` primitive, and for echoing keystrokes. It is not the same as the minibuffer, despite the fact that the minibuffer appears (when active) in the same place on the screen as the echo area. The *GNU Emacs Manual* specifies the rules for resolving conflicts between the echo area and the minibuffer for use of that screen space (see section "The Minibuffer" in *The GNU Emacs Manual*). Error messages appear in the echo area; see Section 9.5.3 [Errors], page 138.

You can write output in the echo area by using the Lisp printing functions with `t` as the stream (see Section 18.5 [Output Functions], page 289), or as follows:

message *string* &rest *arguments* Function

This function displays a one-line message in the echo area. The argument *string* is similar to a C language `printf` control string. See `format` in Section 4.6 [String Conversion], page 66, for the details on the conversion specifications. `message` returns the constructed string.

In batch mode, `message` prints the message text on the standard error stream, followed by a newline.

If *string* is `nil`, `message` clears the echo area. If the minibuffer is active, this brings the minibuffer contents back onto the screen immediately.

```
(message "Minibuffer depth is %d."
         (minibuffer-depth))
 ⊣ Minibuffer depth is 0.
⇒ "Minibuffer depth is 0."

---------- Echo Area ----------
Minibuffer depth is 0.
---------- Echo Area ----------
```

message-or-box *string* &rest *arguments* Function

This function displays a message like `message`, but may display it in a dialog box instead of the echo area. If this function is called in a command that was invoked using the mouse—more precisely, if `last-nonmenu-event` (see Section 20.4 [Command Loop Info], page 328) is either `nil` or a list—then it uses a dialog box or pop-up menu to display the message. Otherwise, it uses the echo area. (This is the same criterion that `y-or-n-p` uses to make a similar decision; see Section 19.6 [Yes-or-No Queries], page 312.)

You can force use of the mouse or of the echo area by binding `last-nonmenu-event` to a suitable value around the call.

message-box *string* &rest *arguments* Function

This function displays a message like `message`, but uses a dialog box (or a pop-up menu) whenever that is possible. If it is impossible to use a dialog box or pop-up menu, because the terminal does not support them, then `message-box` uses the echo area, like `message`.

current-message Function

This function returns the message currently being displayed in the echo area, or `nil` if there is none.

cursor-in-echo-area Variable

This variable controls where the cursor appears when a message is displayed in the echo area. If it is non-`nil`, then the cursor appears at the end of the message. Otherwise, the cursor appears at point—not in the echo area at all.

The value is normally `nil`; Lisp programs bind it to `t` for brief periods of time.

echo-area-clear-hook Variable

This normal hook is run whenever the echo area is cleared—either by `(message nil)` or for any other reason.

Almost all the messages displayed in the echo area are also recorded in the '`*Messages*`' buffer.

message-log-max User Option

This variable specifies how many lines to keep in the '`*Messages*`' buffer. The value `t` means there is no limit on how many lines to keep. The value `nil` disables message logging entirely. Here's how to display a message and prevent it from being logged:

```
(let (message-log-max)
  (message ...))
```

echo-keystrokes Variable

This variable determines how much time should elapse before command characters echo. Its value must be an integer, which specifies the number of seconds to wait before echoing. If the user types a prefix key (such as *C-x*) and then delays this many seconds before continuing, the prefix key is echoed in the echo area. (Once echoing begins in a key sequence, all subsequent characters in the same key sequence are echoed immediately.)

If the value is zero, then command input is not echoed.

38.4 Invisible Text

You can make characters *invisible*, so that they do not appear on the screen, with the `invisible` property. This can be either a text property (see Section 31.19 [Text Properties], page 610) or a property of an overlay (see Section 38.8 [Overlays], page 748).

In the simplest case, any non-`nil` `invisible` property makes a character invisible. This is the default case—if you don't alter the default value of `buffer-invisibility-spec`, this is how the `invisible` property works.

More generally, you can use the variable `buffer-invisibility-spec` to control which values of the `invisible` property make text invisible. This permits you to classify the text into different subsets in advance, by giving them different `invisible` values, and subsequently make various subsets visible or invisible by changing the value of `buffer-invisibility-spec`.

Controlling visibility with `buffer-invisibility-spec` is especially useful in a program to display the list of entries in a data base. It permits the implementation of convenient filtering commands to view just a part of the entries in the data base. Setting this variable is very fast, much faster than scanning all the text in the buffer looking for properties to change.

buffer-invisibility-spec Variable

This variable specifies which kinds of `invisible` properties actually make a character invisible.

`t` A character is invisible if its `invisible` property is non-`nil`. This is the default.

a list Each element of the list specifies a criterion for invisibility; if a character's `invisible` property fits any one of these criteria, the character is invisible. The list can have two kinds of elements:

atom A character is invisible if its `invisible` property value is *atom* or if it is a list with *atom* as a member.

(*atom* `. t`) A character is invisible if its `invisible` property value is *atom* or if it is a list with *atom* as a member. Moreover, if this character is at the end of a line and is followed by a visible newline, it displays an ellipsis.

Two functions are specifically provided for adding elements to `buffer-invisibility-spec` and removing elements from it.

add-to-invisibility-spec *element* Function

Add the element *element* to `buffer-invisibility-spec` (if it is not already present in that list).

remove-from-invisibility-spec *element* Function

Remove the element *element* from `buffer-invisibility-spec`.

One convention about the use of `buffer-invisibility-spec` is that a major mode should use the mode's own name as an element of `buffer-invisibility-spec` and as the value of the `invisible` property:

```
;; If you want to display an ellipsis:
(add-to-invisibility-spec '(my-symbol . t))
;; If you don't want ellipsis:
(add-to-invisibility-spec 'my-symbol)

(overlay-put (make-overlay beginning end)
             'invisible 'my-symbol)

;; When done with the overlays:
(remove-from-invisibility-spec '(my-symbol . t))
;; Or respectively:
(remove-from-invisibility-spec 'my-symbol)
```

Ordinarily, commands that operate on text or move point do not care whether the text is invisible. The user-level line motion commands explicitly ignore invisible newlines if `line-move-ignore-invisible` is non-`nil`, but only because they are explicitly programmed to do so.

Incremental search can make invisible overlays visible temporarily and/or permanently when a match includes invisible text. To enable this, the overlay should have a non-`nil` `isearch-open-invisible` property. The property value should be a function to be called with the overlay as an argument. This function should make the overlay visible permanently; it is used when the match overlaps the overlay on exit from the search.

During the search, such overlays are made temporarily visible by temporarily modifying their invisible and intangible properties. If you want this to be done differently for a certain overlay, give it an `isearch-open-invisible-temporary` property which is a function. The function is called with two arguments: the first is the overlay, and the second is `t` to make the overlay visible, or `nil` to make it invisible again.

38.5 Selective Display

Selective display refers to a pair of related features for hiding certain lines on the screen.

The first variant, explicit selective display, is designed for use in a Lisp program: it controls which lines are hidden by altering the text. The invisible text feature (see Section 38.4 [Invisible Text], page 742) has partially replaced this feature.

In the second variant, the choice of lines to hide is made automatically based on indentation. This variant is designed to be a user-level feature.

The way you control explicit selective display is by replacing a newline (control-j) with a carriage return (control-m). The text that was formerly a line following that newline is now invisible. Strictly speaking, it is temporarily no longer a line at all, since only newlines can separate lines; it is now part of the previous line.

Selective display does not directly affect editing commands. For example, *`C-f`* (`forward-char`) moves point unhesitatingly into invisible text. However, the replacement of newline characters with carriage return characters affects some editing commands. For example, `next-line` skips invisible lines, since it searches only for newlines. Modes that use selective display can also define commands that take account of the newlines, or that make parts of the text visible or invisible.

When you write a selectively displayed buffer into a file, all the control-m's are output as newlines. This means that when you next read in the file, it looks OK, with nothing invisible. The selective display effect is seen only within Emacs.

selective-display Variable

This buffer-local variable enables selective display. This means that lines, or portions of lines, may be made invisible.

- If the value of `selective-display` is `t`, then any portion of a line that follows a control-m is not displayed. This is explicit selective display.
- If the value of `selective-display` is a positive integer, then lines that start with more than that many columns of indentation are not displayed.

When some portion of a buffer is invisible, the vertical movement commands operate as if that portion did not exist, allowing a single `next-line` command to skip any number of invisible lines. However, character movement commands (such as `forward-char`) do not skip the invisible portion, and it is possible (if tricky) to insert or delete text in an invisible portion.

In the examples below, we show the *display appearance* of the buffer `foo`, which changes with the value of `selective-display`. The *contents* of the buffer do not change.

```
(setq selective-display nil)
     ⇒ nil

---------- Buffer: foo ----------
1 on this column
 2on this column
  3n this column
  3n this column
 2on this column
1 on this column
---------- Buffer: foo ----------

(setq selective-display 2)
     ⇒ 2

---------- Buffer: foo ----------
1 on this column
 2on this column
 2on this column
1 on this column
---------- Buffer: foo ----------
```

selective-display-ellipses Variable

If this buffer-local variable is non-`nil`, then Emacs displays '`...`' at the end of a line that is followed by invisible text. This example is a continuation of the previous one.

```
(setq selective-display-ellipses t)
     ⇒ t

---------- Buffer: foo ----------
1 on this column
 2on this column ...
 2on this column
1 on this column
---------- Buffer: foo ----------
```

You can use a display table to substitute other text for the ellipsis (‘...’). See Section 38.14 [Display Tables], page 762.

38.6 The Overlay Arrow

The *overlay arrow* is useful for directing the user's attention to a particular line in a buffer. For example, in the modes used for interface to debuggers, the overlay arrow indicates the line of code about to be executed.

overlay-arrow-string Variable

This variable holds the string to display to call attention to a particular line, or `nil` if the arrow feature is not in use.

overlay-arrow-position Variable

This variable holds a marker that indicates where to display the overlay arrow. It should point at the beginning of a line. The arrow text appears at the beginning of that line, overlaying any text that would otherwise appear. Since the arrow is usually short, and the line usually begins with indentation, normally nothing significant is overwritten.

The overlay string is displayed only in the buffer that this marker points into. Thus, only one buffer can have an overlay arrow at any given time.

You can do a similar job by creating an overlay with a `before-string` property. See Section 38.8.1 [Overlay Properties], page 749.

38.7 Temporary Displays

Temporary displays are used by Lisp programs to put output into a buffer and then present it to the user for perusal rather than for editing. Many help commands use this feature.

with-output-to-temp-buffer *buffer-name forms...* Special Form

This function executes *forms* while arranging to insert any output they print into the buffer named *buffer-name*. The buffer is then shown in some window for viewing, displayed but not selected.

The string *buffer-name* specifies the temporary buffer, which need not already exist. The argument must be a string, not a buffer. The buffer is erased initially (with no questions asked), and it is marked as unmodified after `with-output-to-temp-buffer` exits.

`with-output-to-temp-buffer` binds `standard-output` to the temporary buffer, then it evaluates the forms in *forms*. Output using the Lisp output functions within *forms* goes by default to that buffer (but screen display and messages in the echo area, although they are "output" in the general sense of the word, are not affected). See Section 18.5 [Output Functions], page 289.

The value of the last form in *forms* is returned.

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

(with-output-to-temp-buffer "foo"
    (print 20)
    (print standard-output))
⇒ #<buffer foo>

---------- Buffer: foo ----------
20

#<buffer foo>

---------- Buffer: foo ----------
```

temp-buffer-show-function Variable

If this variable is non-`nil`, `with-output-to-temp-buffer` calls it as a function to do the job of displaying a help buffer. The function gets one argument, which is the buffer it should display.

It is a good idea for this function to run `temp-buffer-show-hook` just as `with-output-to-temp-buffer` normally would, inside of `save-window-excursion` and with the chosen window and buffer selected.

temp-buffer-show-hook Variable

This normal hook is run by `with-output-to-temp-buffer` after displaying the help buffer. When the hook runs, the help buffer is current, and the window it was displayed in is selected.

momentary-string-display *string position* &optional *char message* Function

This function momentarily displays *string* in the current buffer at *position*. It has no effect on the undo list or on the buffer's modification status.

The momentary display remains until the next input event. If the next input event is *char*, `momentary-string-display` ignores it and returns. Otherwise, that event remains buffered for subsequent use as input. Thus, typing *char* will simply remove the string from the display, while typing (say) `C-f` will remove the string from the display and later (presumably) move point forward. The argument *char* is a space by default.

The return value of `momentary-string-display` is not meaningful.

If the string *string* does not contain control characters, you can do the same job in a more general way by creating (and then subsequently deleting) an overlay with a `before-string` property. See Section 38.8.1 [Overlay Properties], page 749.

If *message* is non-`nil`, it is displayed in the echo area while *string* is displayed in the buffer. If it is `nil`, a default message says to type *char* to continue.

In this example, point is initially located at the beginning of the second line:

```
---------- Buffer: foo ----------
This is the contents of foo.
*Second line.
---------- Buffer: foo ----------

(momentary-string-display
  "**** Important Message! ****"
  (point) ?\r
  "Type RET when done reading")
⇒ t

---------- Buffer: foo ----------
This is the contents of foo.
**** Important Message! ****Second line.
---------- Buffer: foo ----------

---------- Echo Area ----------
Type RET when done reading
---------- Echo Area ----------
```

38.8 Overlays

You can use *overlays* to alter the appearance of a buffer's text on the screen, for the sake of presentation features. An overlay is an object that belongs to a particular buffer, and has a specified beginning and end. It also has properties that you can examine and set; these affect the display of the text within the overlay.

38.8.1 Overlay Properties

Overlay properties are like text properties in that the properties that alter how a character is displayed can come from either source. But in most respects they are different. Text properties are considered a part of the text; overlays are specifically considered not to be part of the text. Thus, copying text between various buffers and strings preserves text properties, but does not try to preserve overlays. Changing a buffer's text properties marks the buffer as modified, while moving an overlay or changing its properties does not. Unlike text property changes, overlay changes are not recorded in the buffer's undo list. See Section 31.19 [Text Properties], page 610, for comparison.

`priority` This property's value (which should be a nonnegative number) determines the priority of the overlay. The priority matters when two or more overlays cover the same character and both specify a face for display; the one whose `priority` value is larger takes priority over the other, and its face attributes override the face attributes of the lower priority overlay.

Currently, all overlays take priority over text properties. Please avoid using negative priority values, as we have not yet decided just what they should mean.

`window` If the `window` property is non-`nil`, then the overlay applies only on that window.

`category` If an overlay has a `category` property, we call it the *category* of the overlay. It should be a symbol. The properties of the symbol serve as defaults for the properties of the overlay.

`face` This property controls the way text is displayed—for example, which font and which colors. Its value is a face name or a list of face names. See Section 38.10 [Faces], page 753, for more information.

If the property value is a list, elements may also have the form `(foreground-color .` *color-name*`)` or `(background-color .` *color-name*`)`. These elements specify just the foreground color or just the background color; therefore, there is no need to create a face for each color that you want to use.

`mouse-face`
This property is used instead of `face` when the mouse is within the range of the overlay.

`modification-hooks`
This property's value is a list of functions to be called if any character within the overlay is changed or if text is inserted strictly within the overlay.

The hook functions are called both before and after each change. If the functions save the information they receive, and compare notes between calls, they can determine exactly what change has been made in the buffer text.

When called before a change, each function receives four arguments: the overlay, `nil`, and the beginning and end of the text range to be modified.

When called after a change, each function receives five arguments: the overlay, `t`, the beginning and end of the text range just modified, and the length of the pre-change text replaced by that range. (For an insertion, the pre-change length is zero; for a deletion, that length is the number of characters deleted, and the post-change beginning and end are equal.)

`insert-in-front-hooks`
: This property's value is a list of functions to be called before and after inserting text right at the beginning of the overlay. The calling conventions are the same as for the `modification-hooks` functions.

`insert-behind-hooks`
: This property's value is a list of functions to be called before and after inserting text right at the end of the overlay. The calling conventions are the same as for the `modification-hooks` functions.

`invisible`
: The `invisible` property can make the text in the overlay invisible, which means that it does not appear on the screen. See Section 38.4 [Invisible Text], page 742, for details.

`intangible`
: The `intangible` property on an overlay works just like the `intangible` text property. See Section 31.19.4 [Special Properties], page 615, for details.

`isearch-open-invisible`
: This property tells incremental search how to make an invisible overlay visible, permanently, if the final match overlaps it. See Section 38.4 [Invisible Text], page 742.

`isearch-open-invisible-temporary`
: This property tells incremental search how to make an invisible overlay visible, temporarily, during the search. See Section 38.4 [Invisible Text], page 742.

`before-string`
: This property's value is a string to add to the display at the beginning of the overlay. The string does not appear in the buffer

in any sense—only on the screen. The string should contain only characters that display as a single column—control characters, including tabs or newlines, will give strange results.

`after-string`
: This property's value is a string to add to the display at the end of the overlay. The string does not appear in the buffer in any sense—only on the screen. The string should contain only characters that display as a single column—control characters, including tabs or newlines, will give strange results.

`evaporate`
: If this property is non-`nil`, the overlay is deleted automatically if it ever becomes empty (i.e., if it spans no characters).

`local-map`
: If this property is non-`nil`, it specifies a keymap for a portion of the text. The property's value replaces the buffer's local map, when the character after point is within the overlay. See Section 21.6 [Active Keymaps], page 367.

These are the functions for reading and writing the properties of an overlay.

overlay-get *overlay prop* Function
: This function returns the value of property *prop* recorded in *overlay*, if any. If *overlay* does not record any value for that property, but it does have a `category` property which is a symbol, that symbol's *prop* property is used. Otherwise, the value is `nil`.

overlay-put *overlay prop value* Function
: This function sets the value of property *prop* recorded in *overlay* to *value*. It returns *value*.

See also the function `get-char-property` which checks both overlay properties and text properties for a given character. See Section 31.19.1 [Examining Properties], page 610.

38.8.2 Managing Overlays

This section describes the functions to create, delete and move overlays, and to examine their contents.

make-overlay *start end* &optional *buffer front-advance rear-advance* Function
: This function creates and returns an overlay that belongs to *buffer* and ranges from *start* to *end*. Both *start* and *end* must specify buffer positions; they may be integers or markers. If *buffer* is omitted, the overlay is created in the current buffer.

The arguments *front-advance* and *rear-advance* specify the insertion type for the start of the overlay and for the end of the overlay. See Section 30.5 [Marker Insertion Types], page 569.

overlay-start *overlay* Function
This function returns the position at which *overlay* starts, as an integer.

overlay-end *overlay* Function
This function returns the position at which *overlay* ends, as an integer.

overlay-buffer *overlay* Function
This function returns the buffer that *overlay* belongs to.

delete-overlay *overlay* Function
This function deletes *overlay*. The overlay continues to exist as a Lisp object, but ceases to be attached to the buffer it belonged to, and ceases to have any effect on display.

A deleted overlay is not permanently useless. You can give it a new buffer position by calling `move-overlay`.

move-overlay *overlay start end* &optional *buffer* Function
This function moves *overlay* to *buffer*, and places its bounds at *start* and *end*. Both arguments *start* and *end* must specify buffer positions; they may be integers or markers. If *buffer* is omitted, the overlay stays in the same buffer.

The return value is *overlay*.

This is the only valid way to change the endpoints of an overlay. Do not try modifying the markers in the overlay by hand, as that fails to update other vital data structures and can cause some overlays to be "lost".

overlays-at *pos* Function
This function returns a list of all the overlays that contain position *pos* in the current buffer. The list is in no particular order. An overlay contains position *pos* if it begins at or before *pos*, and ends after *pos*.

overlays-in *beg end* Function
This function returns a list of the overlays that overlap the region *beg* through *end*. "Overlap" means that at least one character is contained within the overlay and also contained within the specified region; however, empty overlays are included in the result if they are located at *beg* or between *beg* and *end*.

next-overlay-change *pos* Function
This function returns the buffer position of the next beginning or end of an overlay, after *pos*.

previous-overlay-change *pos* Function
This function returns the buffer position of the previous beginning or end of an overlay, before *pos*.

38.9 Width

Since not all characters have the same width, these functions let you check the width of a character. See Section 31.17.1 [Primitive Indent], page 604, and Section 29.2.5 [Screen Lines], page 556, for related functions.

char-width *char* Function
This function returns the width in columns of the character *char*, if it were displayed in the current buffer and the selected window.

string-width *string* Function
This function returns the width in columns of the string *string*, if it were displayed in the current buffer and the selected window.

truncate-string-to-width *string width* &optional *start-column padding* Function
This function returns the part of *string* that fits within *width* columns, as a new string.

If *string* does not reach *width*, then the result ends where *string* ends. If one multi-column character in *string* extends across the column *width*, that character is not included in the result. Thus, the result can fall short of *width* but cannot go beyond it.

The optional argument *start-column* specifies the starting column. If this is non-`nil`, then the first *start-column* columns of the string are omitted from the value. If one multi-column character in *string* extends across the column *start-column*, that character is not included.

The optional argument *padding*, if non-`nil`, is a padding character added at the beginning and end of the result string, to extend it to exactly *width* columns. The padding character is used at the end of the result if it falls short of *width*. It is also used at the beginning of the result if one multi-column character in *string* extends across the column *start-column*.

```
(truncate-string-to-width "\tab\t" 12 4)
     ⇒ "ab"
(truncate-string-to-width "\tab\t" 12 4 ?\ )
     ⇒ "    ab  "
```

38.10 Faces

A *face* is a named collection of graphical attributes: font, foreground color, background color, and optional underlining. Faces control the display of text on the screen.

Each face has its own *face number*, which distinguishes faces at low levels within Emacs. However, for most purposes, you can refer to faces in Lisp programs by their names.

facep *object* Function
This function returns `t` if *object* is a face name symbol (or if it is a vector of the kind used internally to record face data). It returns `nil` otherwise.

Each face name is meaningful for all frames, and by default it has the same meaning in all frames. But you can arrange to give a particular face name a special meaning in one frame if you wish.

38.10.1 Standard Faces

This table lists all the standard faces and their uses.

`default` This face is used for ordinary text.

`modeline` This face is used for mode lines and menu bars.

`region` This face is used for highlighting the region in Transient Mark mode.

`secondary-selection`
This face is used to show any secondary selection you have made.

`highlight`
This face is meant to be used for highlighting for various purposes.

`underline`
This face underlines text.

`bold` This face uses a bold font, if possible. It uses the bold variant of the frame's font, if it has one. It's up to you to choose a default font that has a bold variant, if you want to use one.

`italic` This face uses the italic variant of the frame's font, if it has one.

`bold-italic`
This face uses the bold italic variant of the frame's font, if it has one.

38.10.2 Defining Faces

The way to define a new face is with `defface`. This creates a kind of customization item (see Chapter 13 [Customization], page 199) which the user can customize using the Customization buffer (see section "Easy Customization" in *The GNU Emacs Manual*).

defface *face spec doc* [*keyword value*]... Macro

Declare *face* as a customizable face that defaults according to *spec*. Do not quote the symbol *face*. The argument *doc* specifies the face documentation.

When `defface` executes, it defines the face according to *spec*, then uses any customizations that were read from the '`.emacs`' file to override that specification.

The purpose of *spec* is to specify how the face should appear on different kinds of terminals. It should be an alist whose elements have the form (*display atts*). The element's CAR, *display*, specifies a class of terminals. The CDR, *atts*, is a list of face attributes and their values; it specifies what the face should look like on that kind of terminal. The possible attributes are defined in the value of `custom-face-attributes`.

The *display* part of an element of *spec* determines which frames the element applies to. If more than one element of *spec* matches a given frame, the first matching element is the only one used for that frame. There are two possibilities for *display*:

`t` This element of *spec* matches all frames. Therefore, any subsequent elements of *spec* are never used. Normally `t` is used in the last (or only) element of *spec*.

a list If *display* is a list, each element should have the form (*characteristic value*...). Here *characteristic* specifies a way of classifying frames, and the *value*s are possible classifications which *display* should apply to. Here are the possible values of *characteristic*:

`type` The kind of window system the frame uses—either `x`, `pc` (for the MS-DOS console), `w32` (for MS Windows 9X/NT), or `tty`.

`class` What kinds of colors the frame supports—either `color`, `grayscale`, or `mono`.

`background`
The kind of background—either `light` or `dark`.

If an element of *display* specifies more than one *value* for a given *characteristic*, any of those values is acceptable. If *display* has more than one element, each element should specify a different *characteristic*; then *each* characteristic of the frame must match one of the *value*s specified for it in *display*.

Here's how the standard face `region` could be defined with `defface`:

```
(defface region
         ((((class color) (background dark))
           (:background "blue"))
          (t (:background "gray"))))
```

```
"Used for displaying the region.")
```

Internally, `defface` uses the symbol property `face-defface-spec` to record the face attributes specified in `defface`, `saved-face` for the attributes saved by the user with the customization buffer, and `face-documentation` for the documentation string.

frame-background-mode User Option

This option, if non-`nil`, specifies the background type to use for interpreting face definitions. If it is `dark`, then Emacs treats all frames as if they had a dark background, regardless of their actual background colors. If it is `light`, then Emacs treats all frames as if they had a light background.

38.10.3 Merging Faces for Display

Here are all the ways to specify which face to use for display of text:

- With defaults. Each frame has a *default face*, which is used for all text that doesn't somehow specify another face. (We may change this in a forthcoming Emacs version to serve as a default for all text.)
- With text properties. A character may have a `face` property; if so, it is displayed with that face. See Section 31.19.4 [Special Properties], page 615.

  If the character has a `mouse-face` property, that is used instead of the `face` property when the mouse is "near enough" to the character.
- With overlays. An overlay may have `face` and `mouse-face` properties too; they apply to all the text covered by the overlay.
- With a region that is active. In Transient Mark mode, the region is highlighted with a particular face (see `region-face`, below).
- With special glyphs. Each glyph can specify a particular face number. See Section 38.14.3 [Glyphs], page 764.

If these various sources together specify more than one face for a particular character, Emacs merges the attributes of the various faces specified. The attributes of the faces of special glyphs come first; then comes the face for region highlighting, if appropriate; then come attributes of faces from overlays, followed by those from text properties, and last the default face.

When multiple overlays cover one character, an overlay with higher priority overrides those with lower priority. See Section 38.8 [Overlays], page 748.

If an attribute such as the font or a color is not specified in any of the above ways, the frame's own font or color is used.

38.10.4 Functions for Working with Faces

The attributes a face can specify include the font, the foreground color, the background color, and underlining. The face can also leave these unspecified by giving the value `nil` for them.

Here are the primitives for creating and changing faces.

make-face *name* Function
This function defines a new face named *name*, initially with all attributes `nil`. It does nothing if there is already a face named *name*.

face-list Function
This function returns a list of all defined face names.

copy-face *old-face new-name* &optional *frame new-frame* Function
This function defines the face *new-name* as a copy of the existing face named *old-face*. It creates the face *new-name* if that doesn't already exist.

If the optional argument *frame* is given, this function applies only to that frame. Otherwise it applies to each frame individually, copying attributes from *old-face* in each frame to *new-face* in the same frame.

If the optional argument *new-frame* is given, then `copy-face` copies the attributes of *old-face* in *frame* to *new-name* in *new-frame*.

You can modify the attributes of an existing face with the following functions. If you specify *frame*, they affect just that frame; otherwise, they affect all frames as well as the defaults that apply to new frames.

set-face-foreground *face color* &optional *frame* Function
set-face-background *face color* &optional *frame* Function
These functions set the foreground (or background, respectively) color of face *face* to *color*. The argument *color* should be a string, the name of a color.

Certain shades of gray are implemented by stipple patterns on black-and-white screens.

set-face-stipple *face pattern* &optional *frame* Function
This function sets the background stipple pattern of face *face* to *pattern*. The argument *pattern* should be the name of a stipple pattern defined by the X server, or `nil` meaning don't use stipple.

Normally there is no need to pay attention to stipple patterns, because they are used automatically to handle certain shades of gray.

set-face-font *face font* &optional *frame* Function
This function sets the font of face *face*. The argument *font* should be a string, either a valid font name for your system or the name of an Emacs fontset (see Section 28.20 [Fontsets], page 546). Note that if you set the font explicitly, the bold and italic attributes cease to have any effect, because the precise font that you specified is always used.

set-face-bold-p *face bold-p* &optional *frame* Function
This function sets the bold attribute of face *face*. Non-`nil` means bold; `nil` means non-bold.

set-face-italic-p *face italic-p* &optional *frame* Function
This function sets the italic attribute of face *face*. Non-`nil` means italic; `nil` means non-italic.

set-face-underline-p *face underline-p* &optional *frame* Function
This function sets the underline attribute of face *face*. Non-`nil` means do underline; `nil` means don't.

invert-face *face* &optional *frame* Function
Swap the foreground and background colors of face *face*. If the face doesn't specify both foreground and background, then its foreground and background are set to the default background and foreground, respectively.

These functions examine the attributes of a face. If you don't specify *frame*, they refer to the default data for new frames.

face-foreground *face* &optional *frame* Function
face-background *face* &optional *frame* Function
These functions return the foreground color (or background color, respectively) of face *face*, as a string.

face-stipple *face* &optional *frame* Function
This function returns the name of the background stipple pattern of face *face*, or `nil` if it doesn't have one.

face-font *face* &optional *frame* Function
This function returns the name of the font of face *face*.

face-bold-p *face* &optional *frame* Function
This function returns the bold attribute of face *face*.

face-italic-p *face* &optional *frame* Function
This function returns the italic attribute of face *face*.

face-underline-p *face* &optional *frame* Function
This function returns the underline attribute of face *face*.

face-id *face* Function
This function returns the face number of face *face*.

face-documentation *face* Function
This function returns the documentation string of face *face*, or `nil` if none was specified for it.

face-equal *face1 face2* &optional *frame* Function
This returns `t` if the faces *face1* and *face2* have the same attributes for display.

face-differs-from-default-p *face* &optional *frame* Function
This returns `t` if the face *face* displays differently from the default face. A face is considered to be "the same" as the normal face if each attribute is either the same as that of the default face or `nil` (meaning to inherit from the default).

region-face Variable
This variable's value specifies the face number to use to display characters in the region when it is active (in Transient Mark mode only). The face thus specified takes precedence over all faces that come from text properties and overlays, for characters in the region. See Section 30.7 [The Mark], page 570, for more information about Transient Mark mode. Normally, the value is the face number of the face named `region`.

frame-update-face-colors *frame* Function
This function updates the way faces display on *frame*, for a change in *frame*'s foreground or background color.

38.11 Blinking Parentheses

This section describes the mechanism by which Emacs shows a matching open parenthesis when the user inserts a close parenthesis.

blink-paren-function Variable
The value of this variable should be a function (of no arguments) to be called whenever a character with close parenthesis syntax is inserted. The value of `blink-paren-function` may be `nil`, in which case nothing is done.

blink-matching-paren User Option
If this variable is `nil`, then `blink-matching-open` does nothing.

blink-matching-paren-distance User Option
This variable specifies the maximum distance to scan for a matching parenthesis before giving up.

blink-matching-delay User Option

This variable specifies the number of seconds for the cursor to remain at the matching parenthesis. A fraction of a second often gives good results, but the default is 1, which works on all systems.

blink-matching-open Command

This function is the default value of `blink-paren-function`. It assumes that point follows a character with close parenthesis syntax and moves the cursor momentarily to the matching opening character. If that character is not already on the screen, it displays the character's context in the echo area. To avoid long delays, this function does not search farther than `blink-matching-paren-distance` characters.

Here is an example of calling this function explicitly.

```
(defun interactive-blink-matching-open ()
  "Indicate momentarily the start of sexp before point."
  (interactive)
  (let ((blink-matching-paren-distance
         (buffer-size))
        (blink-matching-paren t))
    (blink-matching-open)))
```

38.12 Inverse Video

inverse-video User Option

This variable controls whether Emacs uses inverse video for all text on the screen. Non-`nil` means yes, `nil` means no. The default is `nil`.

mode-line-inverse-video User Option

This variable controls the use of inverse video for mode lines. If it is non-`nil`, then mode lines are displayed in inverse video. Otherwise, mode lines are displayed normally, just like text. The default is `t`.

For window frames, this displays mode lines using the face named `modeline`, which is normally the inverse of the default face unless you change it.

38.13 Usual Display Conventions

The usual display conventions define how to display each character code. You can override these conventions by setting up a display table (see Section 38.14 [Display Tables], page 762). Here are the usual display conventions:

- Character codes 32 through 126 map to glyph codes 32 through 126. Normally this means they display as themselves.

- Character code 9 is a horizontal tab. It displays as whitespace up to a position determined by `tab-width`.
- Character code 10 is a newline.
- All other codes in the range 0 through 31, and code 127, display in one of two ways according to the value of `ctl-arrow`. If it is non-`nil`, these codes map to sequences of two glyphs, where the first glyph is the ASCII code for '`^`'. (A display table can specify a glyph to use instead of '`^`'.) Otherwise, these codes map just like the codes in the range 128 to 255.
- Character codes 128 through 255 map to sequences of four glyphs, where the first glyph is the ASCII code for '`\`', and the others are digit characters representing the character code in octal. (A display table can specify a glyph to use instead of '`\`'.)
- Multibyte character codes above 256 are displayed as themselves, or as a question mark or empty box if the terminal cannot display that character.

The usual display conventions apply even when there is a display table, for any character whose entry in the active display table is `nil`. Thus, when you set up a display table, you need only specify the characters for which you want special behavior.

These variables affect the way certain characters are displayed on the screen. Since they change the number of columns the characters occupy, they also affect the indentation functions. These variables also affect how the mode line is displayed; if you want to force redisplay of the mode line using the new values, call the function `force-mode-line-update` (see Section 22.3 [Mode Line Format], page 405).

ctl-arrow User Option

This buffer-local variable controls how control characters are displayed. If it is non-`nil`, they are displayed as a caret followed by the character: '`^A`'. If it is `nil`, they are displayed as a backslash followed by three octal digits: '`\001`'.

default-ctl-arrow Variable

The value of this variable is the default value for `ctl-arrow` in buffers that do not override it. See Section 10.10.3 [Default Value], page 166.

tab-width User Option

The value of this variable is the spacing between tab stops used for displaying tab characters in Emacs buffers. The default is 8. Note that this feature is completely independent of the user-settable tab stops used by the command `tab-to-tab-stop`. See Section 31.17.5 [Indent Tabs], page 608.

38.14 Display Tables

You can use the *display table* feature to control how all possible character codes display on the screen. This is useful for displaying European languages that have letters not in the ASCII character set.

The display table maps each character code into a sequence of *glyphs*, each glyph being an image that takes up one character position on the screen. You can also define how to display each glyph on your terminal, using the *glyph table*.

Display tables affect how the mode line is displayed; if you want to force redisplay of the mode line using a new display table, call `force-mode-line-update` (see Section 22.3 [Mode Line Format], page 405).

38.14.1 Display Table Format

A display table is actually a char-table (see Section 6.6 [Char-Tables], page 104) with `display-table` as its subtype.

make-display-table Function

This creates and returns a display table. The table initially has `nil` in all elements.

The ordinary elements of the display table are indexed by character codes; the element at index *c* says how to display the character code *c*. The value should be `nil` or a vector of glyph values (see Section 38.14.3 [Glyphs], page 764). If an element is `nil`, it says to display that character according to the usual display conventions (see Section 38.13 [Usual Display], page 760).

If you use the display table to change the display of newline characters, the whole buffer will be displayed as one long "line."

The display table also has six "extra slots" which serve special purposes. Here is a table of their meanings; `nil` in any slot means to use the default for that slot, as stated below.

0 The glyph for the end of a truncated screen line (the default for this is '`$`'). See Section 38.14.3 [Glyphs], page 764.

1 The glyph for the end of a continued line (the default is '`\`').

2 The glyph for indicating a character displayed as an octal character code (the default is '`\`').

3 The glyph for indicating a control character (the default is '`^`').

4 A vector of glyphs for indicating the presence of invisible lines (the default is '`...`'). See Section 38.5 [Selective Display], page 744.

5 The glyph used to draw the border between side-by-side windows (the default is '`|`'). See Section 27.2 [Splitting Windows], page 496.

For example, here is how to construct a display table that mimics the effect of setting `ctl-arrow` to a non-`nil` value:

```
(setq disptab (make-display-table))
(let ((i 0))
  (while (< i 32)
    (or (= i ?\t) (= i ?\n)
        (aset disptab i (vector ?^ (+ i 64))))
    (setq i (1+ i)))
  (aset disptab 127 (vector ?^ ??)))
```

display-table-slot *display-table slot* Function

This function returns the value of the extra slot *slot* of *display-table*. The argument *slot* may be a number from 0 to 5 inclusive, or a slot name (symbol). Valid symbols are `truncation`, `wrap`, `escape`, `control`, `selective-display`, and `vertical-border`.

set-display-table-slot *display-table slot value* Function

This function stores *value* in the extra slot *slot* of *display-table*. The argument *slot* may be a number from 0 to 5 inclusive, or a slot name (symbol). Valid symbols are `truncation`, `wrap`, `escape`, `control`, `selective-display`, and `vertical-border`.

38.14.2 Active Display Table

Each window can specify a display table, and so can each buffer. When a buffer *b* is displayed in window *w*, display uses the display table for window *w* if it has one; otherwise, the display table for buffer *b* if it has one; otherwise, the standard display table if any. The display table chosen is called the *active* display table.

window-display-table *window* Function

This function returns *window*'s display table, or `nil` if *window* does not have an assigned display table.

set-window-display-table *window table* Function

This function sets the display table of *window* to *table*. The argument *table* should be either a display table or `nil`.

buffer-display-table Variable

This variable is automatically buffer-local in all buffers; its value in a particular buffer specifies the display table for that buffer. If it is `nil`, that means the buffer does not have an assigned display table.

standard-display-table Variable

This variable's value is the default display table, used whenever a window has no display table and neither does the buffer displayed in that window. This variable is `nil` by default.

If there is no display table to use for a particular window—that is, if the window specifies none, its buffer specifies none, and `standard-display-table` is `nil`—then Emacs uses the usual display conventions for all character codes in that window. See Section 38.13 [Usual Display], page 760.

38.14.3 Glyphs

A *glyph* is a generalization of a character; it stands for an image that takes up a single character position on the screen. Glyphs are represented in Lisp as integers, just as characters are.

The meaning of each integer, as a glyph, is defined by the glyph table, which is the value of the variable `glyph-table`.

glyph-table Variable

The value of this variable is the current glyph table. It should be a vector; the gth element defines glyph code g. If the value is `nil` instead of a vector, then all glyphs are simple (see below).

Here are the possible types of elements in the glyph table:

string Send the characters in *string* to the terminal to output this glyph. This alternative is available on character terminals, but not under a window system.

integer Define this glyph code as an alias for glyph code *integer*. You can use an alias to specify a face code for the glyph; see below.

`nil` This glyph is simple. On an ordinary terminal, the glyph code mod 524288 is the character to output. In a window system, the glyph code mod 524288 is the character to output, and the glyph code divided by 524288 specifies the face number (see Section 38.10.4 [Face Functions], page 756) to use while outputting it. (524288 is 2^{19}.) See Section 38.10 [Faces], page 753.

If a glyph code is greater than or equal to the length of the glyph table, that code is automatically simple.

38.15 Beeping

This section describes how to make Emacs ring the bell (or blink the screen) to attract the user's attention. Be conservative about how often you do this; frequent bells can become irritating. Also be careful not to use just beeping when signaling an error is more appropriate. (See Section 9.5.3 [Errors], page 138.)

ding &optional *do-not-terminate* Function
This function beeps, or flashes the screen (see `visible-bell` below). It also terminates any keyboard macro currently executing unless *do-not-terminate* is non-`nil`.

beep &optional *do-not-terminate* Function
This is a synonym for `ding`.

visible-bell User Option
This variable determines whether Emacs should flash the screen to represent a bell. Non-`nil` means yes, `nil` means no. This is effective on a window system, and on a character-only terminal provided the terminal's Termcap entry defines the visible bell capability (‘`vb`’).

ring-bell-function Variable
If this is non-`nil`, it specifies how Emacs should "ring the bell." Its value should be a function of no arguments.

38.16 Window Systems

Emacs works with several window systems, most notably the X Window System. Both Emacs and X use the term "window", but use it differently. An Emacs frame is a single window as far as X is concerned; the individual Emacs windows are not known to X at all.

window-system Variable
This variable tells Lisp programs what window system Emacs is running under. The possible values are

`x`	Emacs is displaying using X.
`pc`	Emacs is displaying using MSDOS.
`w32`	Emacs is displaying using Windows NT or Windows 95.
`nil`	Emacs is using a character-based terminal.

window-setup-hook Variable
This variable is a normal hook which Emacs runs after handling the initialization files. Emacs runs this hook after it has completed loading your ‘`.emacs`’ file, the default initialization file (if any), and the terminal-specific Lisp code, and running the hook `term-setup-hook`.

This hook is used for internal purposes: setting up communication with the window system, and creating the initial window. Users should not interfere with it.

39 Customizing the Calendar and Diary

There are many customizations that you can use to make the calendar and diary suit your personal tastes.

39.1 Customizing the Calendar

If you set the variable `view-diary-entries-initially` to `t`, calling up the calendar automatically displays the diary entries for the current date as well. The diary dates appear only if the current date is visible. If you add both of the following lines to your '`.emacs`' file:

```
(setq view-diary-entries-initially t)
(calendar)
```

this displays both the calendar and diary windows whenever you start Emacs.

Similarly, if you set the variable `view-calendar-holidays-initially` to `t`, entering the calendar automatically displays a list of holidays for the current three-month period. The holiday list appears in a separate window.

You can set the variable `mark-diary-entries-in-calendar` to `t` in order to mark any dates with diary entries. This takes effect whenever the calendar window contents are recomputed. There are two ways of marking these dates: by changing the face (see Section 38.10 [Faces], page 753), if the display supports that, or by placing a plus sign ('`+`') beside the date otherwise.

Similarly, setting the variable `mark-holidays-in-calendar` to `t` marks holiday dates, either with a change of face or with an asterisk ('`*`').

The variable `calendar-holiday-marker` specifies how to mark a date as being a holiday. Its value may be a character to insert next to the date, or a face name to use for displaying the date. Likewise, the variable `diary-entry-marker` specifies how to mark a date that has diary entries. The calendar creates faces named `holiday-face` and `diary-face` for these purposes; those symbols are the default values of these variables, when Emacs supports multiple faces on your terminal.

The variable `calendar-load-hook` is a normal hook run when the calendar package is first loaded (before actually starting to display the calendar).

Starting the calendar runs the normal hook `initial-calendar-window-hook`. Recomputation of the calendar display does not run this hook. But if you leave the calendar with the `q` command and reenter it, the hook runs again.

The variable `today-visible-calendar-hook` is a normal hook run after the calendar buffer has been prepared with the calendar when the current date is visible in the window. One use of this hook is to replace today's date with asterisks; to do that, use the hook function `calendar-star-date`.

```
(add-hook 'today-visible-calendar-hook 'calendar-star-date)
```

Another standard hook function marks the current date, either by changing its face or by adding an asterisk. Here's how to use it:

```
(add-hook 'today-visible-calendar-hook 'calendar-mark-today)
```

The variable `calendar-today-marker` specifies how to mark today's date. Its value should be a character to insert next to the date or a face name to use for displaying the date. A face named `calendar-today-face` is provided for this purpose; that symbol is the default for this variable when Emacs supports multiple faces on your terminal.

A similar normal hook, `today-invisible-calendar-hook` is run if the current date is *not* visible in the window.

39.2 Customizing the Holidays

Emacs knows about holidays defined by entries on one of several lists. You can customize these lists of holidays to your own needs, adding or deleting holidays. The lists of holidays that Emacs uses are for general holidays (`general-holidays`), local holidays (`local-holidays`), Christian holidays (`christian-holidays`), Hebrew (Jewish) holidays (`hebrew-holidays`), Islamic (Moslem) holidays (`islamic-holidays`), and other holidays (`other-holidays`).

The general holidays are, by default, holidays common throughout the United States. To eliminate these holidays, set `general-holidays` to `nil`.

There are no default local holidays (but sites may supply some). You can set the variable `local-holidays` to any list of holidays, as described below.

By default, Emacs does not include all the holidays of the religions that it knows, only those commonly found in secular calendars. For a more extensive collection of religious holidays, you can set any (or all) of the variables `all-christian-calendar-holidays`, `all-hebrew-calendar-holidays`, or `all-islamic-calendar-holidays` to `t`. If you want to eliminate the religious holidays, set any or all of the corresponding variables `christian-holidays`, `hebrew-holidays`, and `islamic-holidays` to `nil`.

You can set the variable `other-holidays` to any list of holidays. This list, normally empty, is intended for individual use.

Each of the lists (`general-holidays`, `local-holidays`, `christian-holidays`, `hebrew-holidays`, `islamic-holidays`, and `other-holidays`) is a list of *holiday forms*, each holiday form describing a holiday (or sometimes a list of holidays).

Here is a table of the possible kinds of holiday form. Day numbers and month numbers count starting from 1, but "dayname" numbers count Sunday as 0. The element *string* is always the name of the holiday, as a string.

`(holiday-fixed` *month day string*`)`
: A fixed date on the Gregorian calendar.

`(holiday-float` *month dayname k string*`)`
: The *k*th *dayname* in *month* on the Gregorian calendar (*dayname*=0 for Sunday, and so on); negative *k* means count back from the end of the month.

`(holiday-hebrew` *month day string*`)`
: A fixed date on the Hebrew calendar.

`(holiday-islamic` *month day string*`)`
: A fixed date on the Islamic calendar.

`(holiday-julian` *month day string*`)`
: A fixed date on the Julian calendar.

`(holiday-sexp` *sexp string*`)`
: A date calculated by the Lisp expression *sexp*. The expression should use the variable `year` to compute and return the date of a holiday, or `nil` if the holiday doesn't happen this year. The value of *sexp* must represent the date as a list of the form `(`*month day year*`)`.

`(if` *condition holiday-form*`)`
: A holiday that happens only if *condition* is true.

`(`*function* [*args*]`)`
: A list of dates calculated by the function *function*, called with arguments *args*.

For example, suppose you want to add Bastille Day, celebrated in France on July 14. You can do this as follows:

```
(setq other-holidays '((holiday-fixed 7 14 "Bastille Day")))
```

The holiday form `(holiday-fixed 7 14 "Bastille Day")` specifies the fourteenth day of the seventh month (July).

Many holidays occur on a specific day of the week, at a specific time of month. Here is a holiday form describing Hurricane Supplication Day, celebrated in the Virgin Islands on the fourth Monday in August:

```
(holiday-float 8 1 4 "Hurricane Supplication Day")
```

Here the 8 specifies August, the 1 specifies Monday (Sunday is 0, Tuesday is 2, and so on), and the 4 specifies the fourth occurrence in the month (1 specifies the first occurrence, 2 the second occurrence, -1 the last occurrence, -2 the second-to-last occurrence, and so on).

You can specify holidays that occur on fixed days of the Hebrew, Islamic, and Julian calendars too. For example,

```
(setq other-holidays
      '((holiday-hebrew 10 2 "Last day of Hanukkah")
        (holiday-islamic 3 12 "Mohammed's Birthday")
        (holiday-julian 4 2 "Jefferson's Birthday")))
```

adds the last day of Hanukkah (since the Hebrew months are numbered with 1 starting from Nisan), the Islamic feast celebrating Mohammed's birthday (since the Islamic months are numbered from 1 starting with Muharram), and Thomas Jefferson's birthday, which is 2 April 1743 on the Julian calendar.

To include a holiday conditionally, use either Emacs Lisp's `if` or the `holiday-sexp` form. For example, American presidential elections occur on the first Tuesday after the first Monday in November of years divisible by 4:

```
(holiday-sexp (if (= 0 (% year 4))
                  (calendar-gregorian-from-absolute
                   (1+ (calendar-dayname-on-or-before
                        1 (+ 6 (calendar-absolute-from-gregorian
                                (list 11 1 year))))))
              "US Presidential Election"))
```

or

```
(if (= 0 (% displayed-year 4))
    (fixed 11
           (extract-calendar-day
             (calendar-gregorian-from-absolute
               (1+ (calendar-dayname-on-or-before
                     1 (+ 6 (calendar-absolute-from-gregorian
                               (list 11 1 displayed-year)))))))
           "US Presidential Election"))
```

Some holidays just don't fit into any of these forms because special calculations are involved in their determination. In such cases you must write a Lisp function to do the calculation. To include eclipses, for example, add `(eclipses)` to `other-holidays` and write an Emacs Lisp function `eclipses` that returns a (possibly empty) list of the relevant Gregorian dates among the range visible in the calendar window, with descriptive strings, like this:

```
(((6 27 1991) "Lunar Eclipse") ((7 11 1991) "Solar Eclipse") ... )
```

39.3 Date Display Format

You can customize the manner of displaying dates in the diary, in mode lines, and in messages by setting `calendar-date-display-form`. This variable holds a list of expressions that can involve the variables `month`, `day`, and `year`, which are all numbers in string form, and `monthname` and `dayname`, which are both alphabetic strings. In the American style, the default value of this list is as follows:

```
((if dayname (concat dayname ", ")) monthname " " day ", " year)
```

while in the European style this value is the default:

```
((if dayname (concat dayname ", ")) day " " monthname " " year)
```

The ISO standard date representation is this:

```
(year "-" month "-" day)
```

This specifies a typical American format:

```
(month "/" day "/" (substring year -2))
```

39.4 Time Display Format

The calendar and diary by default display times of day in the conventional American style with the hours from 1 through 12, minutes, and either 'am' or 'pm'. If you prefer the European style, also known in the US as military, in which the hours go from 00 to 23, you can alter the variable `calendar-time-display-form`. This variable is a list of expressions that can involve the variables `12-hours`, `24-hours`, and `minutes`, which are all numbers in string form, and `am-pm` and `time-zone`, which are both alphabetic strings. The default value of `calendar-time-display-form` is as follows:

```
(12-hours ":" minutes am-pm
          (if time-zone " (") time-zone (if time-zone ")"))
```

Here is a value that provides European style times:

```
(24-hours ":" minutes
          (if time-zone " (") time-zone (if time-zone ")"))
```

39.5 Daylight Savings Time

Emacs understands the difference between standard time and daylight savings time—the times given for sunrise, sunset, solstices, equinoxes, and the phases of the moon take that into account. The rules for daylight savings time vary from place to place and have also varied historically from year to year. To do the job properly, Emacs needs to know which rules to use.

Some operating systems keep track of the rules that apply to the place where you are; on these systems, Emacs gets the information it needs from the system automatically. If some or all of this information is missing, Emacs fills in the gaps with the rules currently used in Cambridge, Massachusetts, which is the center of GNU's world.

If the default choice of rules is not appropriate for your location, you can tell Emacs the rules to use by setting the variables `calendar-daylight-savings-starts` and `calendar-daylight-savings-ends`. Their values should be Lisp expressions that refer to the variable `year`, and evaluate to the Gregorian date on which daylight savings time starts or (respectively) ends, in the form of a list (*month day year*). The values should be `nil` if your area does not use daylight savings time.

Emacs uses these expressions to determine the start and end dates of daylight savings time as holidays and for correcting times of day in the solar and lunar calculations.

The values for Cambridge, Massachusetts are as follows:

```
(calendar-nth-named-day 1 0 4 year)
(calendar-nth-named-day -1 0 10 year)
```

i.e., the first 0th day (Sunday) of the fourth month (April) in the year specified by `year`, and the last Sunday of the tenth month (October) of that

year. If daylight savings time were changed to start on October 1, you would set `calendar-daylight-savings-starts` to this:

```
(list 10 1 year)
```

For a more complex example, suppose daylight savings time begins on the first of Nisan on the Hebrew calendar. You should set `calendar-daylight-savings-starts` to this value:

```
(calendar-gregorian-from-absolute
  (calendar-absolute-from-hebrew
    (list 1 1 (+ year 3760))))
```

because Nisan is the first month in the Hebrew calendar and the Hebrew year differs from the Gregorian year by 3760 at Nisan.

If there is no daylight savings time at your location, or if you want all times in standard time, set `calendar-daylight-savings-starts` and `calendar-daylight-savings-ends` to `nil`.

The variable `calendar-daylight-time-offset` specifies the difference between daylight savings time and standard time, measured in minutes. The value for Cambridge is 60.

The variable `calendar-daylight-savings-starts-time` and the variable `calendar-daylight-savings-ends-time` specify the number of minutes after midnight local time when the transition to and from daylight savings time should occur. For Cambridge, both variables' values are 120.

39.6 Customizing the Diary

Ordinarily, the mode line of the diary buffer window indicates any holidays that fall on the date of the diary entries. The process of checking for holidays can take several seconds, so including holiday information delays the display of the diary buffer noticeably. If you'd prefer to have a faster display of the diary buffer but without the holiday information, set the variable `holidays-in-diary-buffer` to `nil`.

The variable `number-of-diary-entries` controls the number of days of diary entries to be displayed at one time. It affects the initial display when `view-diary-entries-initially` is `t`, as well as the command *`M-x diary`*. For example, the default value is 1, which says to display only the current day's diary entries. If the value is 2, both the current day's and the next day's entries are displayed. The value can also be a vector of seven elements: for example, if the value is `[0 2 2 2 2 4 1]` then no diary entries appear on Sunday, the current date's and the next day's diary entries appear Monday through Thursday, Friday through Monday's entries appear on Friday, while on Saturday only that day's entries appear.

The variable `print-diary-entries-hook` is a normal hook run after preparation of a temporary buffer containing just the diary entries currently visible in the diary buffer. (The other, irrelevant diary entries are really absent from the temporary buffer; in the diary buffer, they are merely hid-

den.) The default value of this hook does the printing with the command `lpr-buffer`. If you want to use a different command to do the printing, just change the value of this hook. Other uses might include, for example, rearranging the lines into order by day and time.

You can customize the form of dates in your diary file, if neither the standard American nor European styles suits your needs, by setting the variable `diary-date-forms`. This variable is a list of patterns for recognizing a date. Each date pattern is a list whose elements may be regular expressions (see Section 33.2 [Regular Expressions], page 649) or the symbols `month`, `day`, `year`, `monthname`, and `dayname`. All these elements serve as patterns that match certain kinds of text in the diary file. In order for the date pattern, as a whole, to match, all of its elements must match consecutively.

A regular expression in a date pattern matches in its usual fashion, using the standard syntax table altered so that '`*`' is a word constituent.

The symbols `month`, `day`, `year`, `monthname`, and `dayname` match the month number, day number, year number, month name, and day name of the date being considered. The symbols that match numbers allow leading zeros; those that match names allow three-letter abbreviations and capitalization. All the symbols can match '`*`'; since '`*`' in a diary entry means "any day", "any month", and so on, it should match regardless of the date being considered.

The default value of `diary-date-forms` in the American style is this:

```
((month "/" day "[^/0-9]")
 (month "/" day "/" year "[^0-9]")
 (monthname " *" day "[^,0-9]")
 (monthname " *" day ", *" year "[^0-9]")
 (dayname "\\W"))
```

The date patterns in the list must be *mutually exclusive* and must not match any portion of the diary entry itself, just the date and one character of whitespace. If, to be mutually exclusive, the pattern must match a portion of the diary entry text—beyond the whitespace that ends the date—then the first element of the date pattern *must* be `backup`. This causes the date recognizer to back up to the beginning of the current word of the diary entry, after finishing the match. Even if you use `backup`, the date pattern must absolutely not match more than a portion of the first word of the diary entry. The default value of `diary-date-forms` in the European style is this list:

```
((day "/" month "[^/0-9]")
 (day "/" month "/" year "[^0-9]")
 (backup day " *" monthname "\\W+\\<[^*0-9]")
 (day " *" monthname " *" year "[^0-9]")
 (dayname "\\W"))
```

Notice the use of `backup` in the third pattern, because it needs to match part of a word beyond the date itself to distinguish it from the fourth pattern.

39.7 Hebrew- and Islamic-Date Diary Entries

Your diary file can have entries based on Hebrew or Islamic dates, as well as entries based on the world-standard Gregorian calendar. However, because recognition of such entries is time-consuming and most people don't use them, you must explicitly enable their use. If you want the diary to recognize Hebrew-date diary entries, for example, you must do this:

```
(add-hook 'nongregorian-diary-listing-hook 'list-hebrew-diary-entries)
(add-hook 'nongregorian-diary-marking-hook 'mark-hebrew-diary-entries)
```

If you want Islamic-date entries, do this:

```
(add-hook 'nongregorian-diary-listing-hook 'list-islamic-diary-entries)
(add-hook 'nongregorian-diary-marking-hook 'mark-islamic-diary-entries)
```

Hebrew- and Islamic-date diary entries have the same formats as Gregorian-date diary entries, except that 'H' precedes a Hebrew date and 'I' precedes an Islamic date. Moreover, because the Hebrew and Islamic month names are not uniquely specified by the first three letters, you may not abbreviate them. For example, a diary entry for the Hebrew date Heshvan 25 could look like this:

```
HHeshvan 25 Happy Hebrew birthday!
```

and would appear in the diary for any date that corresponds to Heshvan 25 on the Hebrew calendar. And here is an Islamic-date diary entry that matches Dhu al-Qada 25:

```
IDhu al-Qada 25 Happy Islamic birthday!
```

As with Gregorian-date diary entries, Hebrew- and Islamic-date entries are nonmarking if they are preceded with an ampersand ('&').

Here is a table of commands used in the calendar to create diary entries that match the selected date and other dates that are similar in the Hebrew or Islamic calendar:

i h d Add a diary entry for the Hebrew date corresponding to the selected date (`insert-hebrew-diary-entry`).

i h m Add a diary entry for the day of the Hebrew month corresponding to the selected date (`insert-monthly-hebrew-diary-entry`). This diary entry matches any date that has the same Hebrew day-within-month as the selected date.

i h y Add a diary entry for the day of the Hebrew year corresponding to the selected date (`insert-yearly-hebrew-diary-entry`). This diary entry matches any date which has the same Hebrew month and day-within-month as the selected date.

i i d Add a diary entry for the Islamic date corresponding to the selected date (`insert-islamic-diary-entry`).

i i m Add a diary entry for the day of the Islamic month corresponding to the selected date (`insert-monthly-islamic-diary-entry`).

`i i y` Add a diary entry for the day of the Islamic year corresponding to the selected date (`insert-yearly-islamic-diary-entry`).

These commands work much like the corresponding commands for ordinary diary entries: they apply to the date that point is on in the calendar window, and what they do is insert just the date portion of a diary entry at the end of your diary file. You must then insert the rest of the diary entry.

39.8 Fancy Diary Display

Diary display works by preparing the diary buffer and then running the hook `diary-display-hook`. The default value of this hook (`simple-diary-display`) hides the irrelevant diary entries and then displays the buffer. However, if you specify the hook as follows,

```
(add-hook 'diary-display-hook 'fancy-diary-display)
```

this enables fancy diary display. It displays diary entries and holidays by copying them into a special buffer that exists only for the sake of display. Copying to a separate buffer provides an opportunity to change the displayed text to make it prettier—for example, to sort the entries by the dates they apply to.

As with simple diary display, you can print a hard copy of the buffer with `print-diary-entries`. To print a hard copy of a day-by-day diary for a week by positioning point on Sunday of that week, type *`7 d`* and then do *`M-x print-diary-entries`*. As usual, the inclusion of the holidays slows down the display slightly; you can speed things up by setting the variable `holidays-in-diary-buffer` to `nil`.

Ordinarily, the fancy diary buffer does not show days for which there are no diary entries, even if that day is a holiday. If you want such days to be shown in the fancy diary buffer, set the variable `diary-list-include-blanks` to `t`.

If you use the fancy diary display, you can use the normal hook `list-diary-entries-hook` to sort each day's diary entries by their time of day. Here's how:

```
(add-hook 'list-diary-entries-hook 'sort-diary-entries t)
```

For each day, this sorts diary entries that begin with a recognizable time of day according to their times. Diary entries without times come first within each day.

Fancy diary display also has the ability to process included diary files. This permits a group of people to share a diary file for events that apply to all of them. Lines in the diary file of this form:

```
#include "filename"
```

includes the diary entries from the file *filename* in the fancy diary buffer. The include mechanism is recursive, so that included files can include other

files, and so on; you must be careful not to have a cycle of inclusions, of course. Here is how to enable the include facility:

```
(add-hook 'list-diary-entries-hook 'include-other-diary-files)
(add-hook 'mark-diary-entries-hook 'mark-included-diary-files)
```

The include mechanism works only with the fancy diary display, because ordinary diary display shows the entries directly from your diary file.

39.9 Sexp Entries and the Fancy Diary Display

Sexp diary entries allow you to do more than just have complicated conditions under which a diary entry applies. If you use the fancy diary display, sexp entries can generate the text of the entry depending on the date itself. For example, an anniversary diary entry can insert the number of years since the anniversary date into the text of the diary entry. Thus the '`%d`' in this dairy entry:

```
%%(diary-anniversary 10 31 1948) Arthur's birthday (%d years old)
```

gets replaced by the age, so on October 31, 1990 the entry appears in the fancy diary buffer like this:

```
Arthur's birthday (42 years old)
```

If the diary file instead contains this entry:

```
%%(diary-anniversary 10 31 1948) Arthur's %d%s birthday
```

the entry in the fancy diary buffer for October 31, 1990 appears like this:

```
Arthur's 42nd birthday
```

Similarly, cyclic diary entries can interpolate the number of repetitions that have occurred:

```
%%(diary-cyclic 50 1 1 1990) Renew medication (%d%s time)
```

looks like this:

```
Renew medication (5th time)
```

in the fancy diary display on September 8, 1990.

There is an early reminder diary sexp that includes its entry in the diary not only on the date of occurrence, but also on earlier dates. For example, if you want a reminder a week before your anniversary, you can use

```
%%(diary-remind '(diary-anniversary 12 22 1968) 7) Ed's anniversary
```

and the fancy diary will show

```
Ruth & Ed's anniversary
```

both on December 15 and on December 22.

The function `diary-date` applies to dates described by a month, day, year combination, each of which can be an integer, a list of integers, or `t`. The value `t` means all values. For example,

```
%%(diary-date '(10 11 12) 22 t) Rake leaves
```

causes the fancy diary to show

```
Rake leaves
```

on October 22, November 22, and December 22 of every year.

The function `diary-float` allows you to describe diary entries that apply to dates like the third Friday of November, or the last Tuesday in April. The parameters are the *month*, *dayname*, and an index *n*. The entry appears on the *n*th *dayname* of *month*, where *dayname*=0 means Sunday, 1 means Monday, and so on. If *n* is negative it counts backward from the end of *month*. The value of *month* can be a list of months, a single month, or `t` to specify all months. You can also use an optional parameter *day* to specify the *n*th *dayname* of *month* on or after/before *day*; the value of *day* defaults to 1 if *n* is positive and to the last day of *month* if *n* is negative. For example,

```
%%(diary-float t 1 -1) Pay rent
```

causes the fancy diary to show

```
Pay rent
```

on the last Monday of every month.

The generality of sexp diary entries lets you specify any diary entry that you can describe algorithmically. A sexp diary entry contains an expression that computes whether the entry applies to any given date. If its value is non-`nil`, the entry applies to that date; otherwise, it does not. The expression can use the variable `date` to find the date being considered; its value is a list (*month day year*) that refers to the Gregorian calendar.

Suppose you get paid on the 21st of the month if it is a weekday, and on the Friday before if the 21st is on a weekend. Here is how to write a sexp diary entry that matches those dates:

```
&%%(let ((dayname (calendar-day-of-week date))
         (day (car (cdr date))))
       (or (and (= day 21) (memq dayname '(1 2 3 4 5)))
           (and (memq day '(19 20)) (= dayname 5)))
       ) Pay check deposited
```

The following sexp diary entries take advantage of the ability (in the fancy diary display) to concoct diary entries whose text varies based on the date:

`%%(diary-sunrise-sunset)`
: Make a diary entry for the local times of today's sunrise and sunset.

`%%(diary-phases-of-moon)`
: Make a diary entry for the phases (quarters) of the moon.

`%%(diary-day-of-year)`
: Make a diary entry with today's day number in the current year and the number of days remaining in the current year.

`%%(diary-iso-date)`
: Make a diary entry with today's equivalent ISO commercial date.

`%%(diary-julian-date)`
: Make a diary entry with today's equivalent date on the Julian calendar.

`%%(diary-astro-day-number)`
: Make a diary entry with today's equivalent astronomical (Julian) day number.

`%%(diary-hebrew-date)`
: Make a diary entry with today's equivalent date on the Hebrew calendar.

`%%(diary-islamic-date)`
: Make a diary entry with today's equivalent date on the Islamic calendar.

`%%(diary-french-date)`
: Make a diary entry with today's equivalent date on the French Revolutionary calendar.

`%%(diary-mayan-date)`
: Make a diary entry with today's equivalent date on the Mayan calendar.

Thus including the diary entry

```
&%%(diary-hebrew-date)
```

causes every day's diary display to contain the equivalent date on the Hebrew calendar, if you are using the fancy diary display. (With simple diary display, the line '`&%%(diary-hebrew-date)`' appears in the diary for any date, but does nothing particularly useful.)

These functions can be used to construct sexp diary entries based on the Hebrew calendar in certain standard ways:

`%%(diary-rosh-hodesh)`
: Make a diary entry that tells the occurrence and ritual announcement of each new Hebrew month.

`%%(diary-parasha)`
: Make a Saturday diary entry that tells the weekly synagogue scripture reading.

`%%(diary-sabbath-candles)`
: Make a Friday diary entry that tells the *local time* of Sabbath candle lighting.

`%%(diary-omer)`
: Make a diary entry that gives the omer count, when appropriate.

`%%(diary-yahrzeit` *month day year*`)` *name*
: Make a diary entry marking the anniversary of a date of death. The date is the *Gregorian* (civil) date of death. The diary entry

appears on the proper Hebrew calendar anniversary and on the day before. (In the European style, the order of the parameters is changed to *day*, *month*, *year*.)

39.10 Customizing Appointment Reminders

You can specify exactly how Emacs reminds you of an appointment, and how far in advance it begins doing so, by setting these variables:

`appt-message-warning-time`
: The time in minutes before an appointment that the reminder begins. The default is 10 minutes.

`appt-audible`
: If this is non-`nil`, Emacs rings the terminal bell for appointment reminders. The default is `t`.

`appt-visible`
: If this is non-`nil`, Emacs displays the appointment message in the echo area. The default is `t`.

`appt-display-mode-line`
: If this is non-`nil`, Emacs displays the number of minutes to the appointment on the mode line. The default is `t`.

`appt-msg-window`
: If this is non-`nil`, Emacs displays the appointment message in another window. The default is `t`.

`appt-disp-window-function`
: This variable holds a function to use to create the other window for the appointment message.

`appt-delete-window-function`
: This variable holds a function to use to get rid of the appointment message window, when its time is up.

`appt-display-duration`
: The number of seconds to display an appointment message. The default is 5 seconds.

Appendix A Tips and Conventions

This chapter describes no additional features of Emacs Lisp. Instead it gives advice on making effective use of the features described in the previous chapters, and describes conventions Emacs Lisp programmers should follow.

A.1 Emacs Lisp Coding Conventions

Here are conventions that you should follow when writing Emacs Lisp code intended for widespread use:

- Since all global variables share the same name space, and all functions share another name space, you should choose a short word to distinguish your program from other Lisp programs. Then take care to begin the names of all global variables, constants, and functions with the chosen prefix. This helps avoid name conflicts.

  This recommendation applies even to names for traditional Lisp primitives that are not primitives in Emacs Lisp—even to `copy-list`. Believe it or not, there is more than one plausible way to define `copy-list`. Play it safe; append your name prefix to produce a name like `foo-copy-list` or `mylib-copy-list` instead.

  If you write a function that you think ought to be added to Emacs under a certain name, such as `twiddle-files`, don't call it by that name in your program. Call it `mylib-twiddle-files` in your program, and send mail to '`bug-gnu-emacs@gnu.org`' suggesting we add it to Emacs. If and when we do, we can change the name easily enough.

  If one prefix is insufficient, your package may use two or three alternative common prefixes, so long as they make sense.

  Separate the prefix from the rest of the symbol name with a hyphen, '`-`'. This will be consistent with Emacs itself and with most Emacs Lisp programs.
- It is often useful to put a call to `provide` in each separate library program, at least if there is more than one entry point to the program.
- If a file requires certain other library programs to be loaded beforehand, then the comments at the beginning of the file should say so. Also, use `require` to make sure they are loaded.
- If one file *foo* uses a macro defined in another file *bar*, *foo* should contain this expression before the first use of the macro:

  ```
  (eval-when-compile (require 'bar))
  ```

  (And the library *bar* should contain `(provide 'bar)`, to make the `require` work.) This will cause *bar* to be loaded when you byte-compile *foo*. Otherwise, you risk compiling *foo* without the necessary macro loaded, and that would produce compiled code that won't work right. See Section 12.3 [Compiling Macros], page 190.

Using `eval-when-compile` avoids loading *bar* when the compiled version of *foo* is *used*.

- When defining a major mode, please follow the major mode conventions. See Section 22.1.1 [Major Mode Conventions], page 392.
- When defining a minor mode, please follow the minor mode conventions. See Section 22.2.1 [Minor Mode Conventions], page 402.
- If the purpose of a function is to tell you whether a certain condition is true or false, give the function a name that ends in '`p`'. If the name is one word, add just '`p`'; if the name is multiple words, add '`-p`'. Examples are `framep` and `frame-live-p`.
- If a user option variable records a true-or-false condition, give it a name that ends in '`-flag`'.
- Please do not define `C-c` *letter* as a key in your major modes. These sequences are reserved for users; they are the **only** sequences reserved for users, so do not block them.

  Instead, define sequences consisting of `C-c` followed by a control character, a digit, or certain punctuation characters. These sequences are reserved for major modes.

  Changing all the Emacs major modes to follow this convention was a lot of work. Abandoning this convention would make that work go to waste, and inconvenience users.
- Sequences consisting of `C-c` followed by `{`, `}`, `<`, `>`, `:` or `;` are also reserved for major modes.
- Sequences consisting of `C-c` followed by any other punctuation character are allocated for minor modes. Using them in a major mode is not absolutely prohibited, but if you do that, the major mode binding may be shadowed from time to time by minor modes.
- Function keys F5 through F9 without modifier keys are reserved for users to define.
- Do not bind `C-h` following any prefix character (including `C-c`). If you don't bind `C-h`, it is automatically available as a help character for listing the subcommands of the prefix character.
- Do not bind a key sequence ending in ESC except following another ESC. (That is, it is OK to bind a sequence ending in ESC ESC.)

  The reason for this rule is that a non-prefix binding for ESC in any context prevents recognition of escape sequences as function keys in that context.
- Anything which acts like a temporary mode or state which the user can enter and leave should define ESC ESC of ESC ESC ESC as a way to escape.

  For a state which accepts ordinary Emacs commands, or more generally any kind of state in which ESC followed by a function key or arrow

key is potentially meaningful, then you must not define ESC ESC, since that would preclude recognizing an escape sequence after ESC. In these states, you should define ESC ESC ESC as the way to escape. Otherwise, define ESC ESC instead.

- Applications should not bind mouse events based on button 1 with the shift key held down. These events include *S-mouse-1*, *M-S-mouse-1*, *C-S-mouse-1*, and so on. They are reserved for users.
- Special major modes used for read-only text should usually redefine *mouse-2* and RET to trace some sort of reference in the text. Modes such as Dired, Info, Compilation, and Occur redefine it in this way.
- When a package provides a modification of ordinary Emacs behavior, it is good to include a command to enable and disable the feature, Provide a command named *whatever*`-mode` which turns the feature on or off, and make it autoload (see Section 14.4 [Autoload], page 215). Design the package so that simply loading it has no visible effect—that should not enable the feature. Users will request the feature by invoking the command.
- It is a bad idea to define aliases for the Emacs primitives. Use the standard names instead.
- Redefining (or advising) an Emacs primitive is discouraged. It may do the right thing for a particular program, but there is no telling what other programs might break as a result.
- If a file does replace any of the functions or library programs of standard Emacs, prominent comments at the beginning of the file should say which functions are replaced, and how the behavior of the replacements differs from that of the originals.
- Please keep the names of your Emacs Lisp source files to 13 characters or less. This way, if the files are compiled, the compiled files' names will be 14 characters or less, which is short enough to fit on all kinds of Unix systems.
- Don't use `next-line` or `previous-line` in programs; nearly always, `forward-line` is more convenient as well as more predictable and robust. See Section 29.2.4 [Text Lines], page 554.
- Don't call functions that set the mark, unless setting the mark is one of the intended features of your program. The mark is a user-level feature, so it is incorrect to change the mark except to supply a value for the user's benefit. See Section 30.7 [The Mark], page 570.

  In particular, don't use any of these functions:

  - `beginning-of-buffer`, `end-of-buffer`
  - `replace-string`, `replace-regexp`

  If you just want to move point, or replace a certain string, without any of the other features intended for interactive users, you can replace these functions with one or two lines of simple Lisp code.

- Use lists rather than vectors, except when there is a particular reason to use a vector. Lisp has more facilities for manipulating lists than for vectors, and working with lists is usually more convenient.

  Vectors are advantageous for tables that are substantial in size and are accessed in random order (not searched front to back), provided there is no need to insert or delete elements (only lists allow that).
- The recommended way to print a message in the echo area is with the `message` function, not `princ`. See Section 38.3 [The Echo Area], page 740.
- When you encounter an error condition, call the function `error` (or `signal`). The function `error` does not return. See Section 9.5.3.1 [Signaling Errors], page 138.

  Do not use `message`, `throw`, `sleep-for`, or `beep` to report errors.
- An error message should start with a capital letter but should not end with a period.
- Many commands that take a long time to execute display a message that says '`Operating...`' when they start, and change it to '`Operating...done`' when they finish. Please keep the style of these messages uniform: *no* space around the ellipsis, and *no* period at the end.
- Try to avoid using recursive edits. Instead, do what the Rmail `e` command does: use a new local keymap that contains one command defined to switch back to the old local keymap. Or do what the `edit-options` command does: switch to another buffer and let the user switch back at will. See Section 20.11 [Recursive Editing], page 355.
- In some other systems there is a convention of choosing variable names that begin and end with '`*`'. We don't use that convention in Emacs Lisp, so please don't use it in your programs. (Emacs uses such names only for special-purpose buffers.) The users will find Emacs more coherent if all libraries use the same conventions.
- Try to avoid compiler warnings about undefined free variables, by adding `defvar` definitions for these variables.

  If you bind a variable in one function, and use it or set it in another function, the compiler warns about the latter function unless the variable has a definition. But often these variables have short names, and it is not clean for Lisp packages to define such variable names. Therefore, you should rename the variable to start with the name prefix used for the other functions and variables in your package.
- Indent each function with *C-M-q* (`indent-sexp`) using the default indentation parameters.
- Don't make a habit of putting close-parentheses on lines by themselves; Lisp programmers find this disconcerting. Once in a while, when there is a sequence of many consecutive close-parentheses, it may make sense to split the sequence in one or two significant places.

- Please put a copyright notice on the file if you give copies to anyone. Use a message like this one:

```
;; Copyright (C) year name

;; This program is free software; you can redistribute it and/or
;; modify it under the terms of the GNU General Public License as
;; published by the Free Software Foundation; either version 2 of
;; the License, or (at your option) any later version.

;; This program is distributed in the hope that it will be
;; useful, but WITHOUT ANY WARRANTY; without even the implied
;; warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR
;; PURPOSE.  See the GNU General Public License for more details.

;; You should have received a copy of the GNU General Public
;; License along with this program; if not, write to the Free
;; Software Foundation, Inc., 59 Temple Place, Suite 330, Boston,
;; MA 02111-1307 USA
```

  If you have signed papers to assign the copyright to the Foundation, then use '`Free Software Foundation, Inc.`' as *name*. Otherwise, use your name.

A.2 Tips for Making Compiled Code Fast

Here are ways of improving the execution speed of byte-compiled Lisp programs.

- Profile your program with the '`profile`' library or the '`elp`' library. See the files '`profile.el`' and '`elp.el`' for instructions.
- Use iteration rather than recursion whenever possible. Function calls are slow in Emacs Lisp even when a compiled function is calling another compiled function.
- Using the primitive list-searching functions `memq`, `member`, `assq`, or `assoc` is even faster than explicit iteration. It can be worth rearranging a data structure so that one of these primitive search functions can be used.
- Certain built-in functions are handled specially in byte-compiled code, avoiding the need for an ordinary function call. It is a good idea to use these functions rather than alternatives. To see whether a function is handled specially by the compiler, examine its `byte-compile` property. If the property is non-`nil`, then the function is handled specially.

  For example, the following input will show you that `aref` is compiled specially (see Section 6.3 [Array Functions], page 100):

```
(get 'aref 'byte-compile)
     ⇒ byte-compile-two-args
```

- If calling a small function accounts for a substantial part of your program's running time, make the function inline. This eliminates the function call overhead. Since making a function inline reduces the flexibility of changing the program, don't do it unless it gives a noticeable speedup in something slow enough that users care about the speed. See Section 11.9 [Inline Functions], page 186.

A.3 Tips for Documentation Strings

Here are some tips and conventions for the writing of documentation strings. You can check many of these conventions by running the command *M-x checkdoc-minor-mode*.

- Every command, function, or variable intended for users to know about should have a documentation string.
- An internal variable or subroutine of a Lisp program might as well have a documentation string. In earlier Emacs versions, you could save space by using a comment instead of a documentation string, but that is no longer the case.
- The first line of the documentation string should consist of one or two complete sentences that stand on their own as a summary. *M-x apropos* displays just the first line, and if it doesn't stand on its own, the result looks bad. In particular, start the first line with a capital letter and end with a period.

  The documentation string can have additional lines that expand on the details of how to use the function or variable. The additional lines should be made up of complete sentences also, but they may be filled if that looks good.
- For consistency, phrase the verb in the first sentence of a function's documentation string as an infinitive with "to" omitted. For instance, use "Return the cons of A and B." in preference to "Returns the cons of A and B." Usually it looks good to do likewise for the rest of the first paragraph. Subsequent paragraphs usually look better if they have proper subjects.
- Write documentation strings in the active voice, not the passive, and in the present tense, not the future. For instance, use "Return a list containing A and B." instead of "A list containing A and B will be returned."
- Avoid using the word "cause" (or its equivalents) unnecessarily. Instead of, "Cause Emacs to display text in boldface," write just "Display text in boldface."
- Do not start or end a documentation string with whitespace.

- Format the documentation string so that it fits in an Emacs window on an 80-column screen. It is a good idea for most lines to be no wider than 60 characters. The first line can be wider if necessary to fit the information that ought to be there.

  However, rather than simply filling the entire documentation string, you can make it much more readable by choosing line breaks with care. Use blank lines between topics if the documentation string is long.

- **Do not** indent subsequent lines of a documentation string so that the text is lined up in the source code with the text of the first line. This looks nice in the source code, but looks bizarre when users view the documentation. Remember that the indentation before the starting double-quote is not part of the string!

- When the user tries to use a disabled command, Emacs displays just the first paragraph of its documentation string—everything through the first blank line. If you wish, you can choose which information to include before the first blank line so as to make this display useful.

- A variable's documentation string should start with '`*`' if the variable is one that users would often want to set interactively. If the value is a long list, or a function, or if the variable would be set only in init files, then don't start the documentation string with '`*`'. See Section 10.5 [Defining Variables], page 152.

- The documentation string for a variable that is a yes-or-no flag should start with words such as "Non-nil means...", to make it clear that all non-`nil` values are equivalent and indicate explicitly what `nil` and non-`nil` mean.

- When a function's documentation string mentions the value of an argument of the function, use the argument name in capital letters as if it were a name for that value. Thus, the documentation string of the function `/` refers to its second argument as '`DIVISOR`', because the actual argument name is `divisor`.

  Also use all caps for meta-syntactic variables, such as when you show the decomposition of a list or vector into subunits, some of which may vary.

- When a documentation string refers to a Lisp symbol, write it as it would be printed (which usually means in lower case), with single-quotes around it. For example: '`‘lambda’`'. There are two exceptions: write `t` and `nil` without single-quotes.

  Help mode automatically creates a hyperlink when a documentation string uses a symbol name inside single quotes, if the symbol has either a function or a variable definition. You do not need to do anything special to make use of this feature. However, when a symbol has both a function definition and a variable definition, and you want to refer to just one of them, you can specify which one by writing one of the words '`variable`', '`option`', '`function`', or '`command`', immediately before the

symbol name. (Case makes no difference in recognizing these indicator words.) For example, if you write

```
This function sets the variable `buffer-file-name'.
```

then the hyperlink will refer only to the variable documentation of `buffer-file-name`, and not to its function documentation.

If a symbol has a function definition and/or a variable definition, but those are irrelevant to the use of the symbol that you are documenting, you can write the word '`symbol`' before the symbol name to prevent making any hyperlink. For example,

```
If the argument KIND-OF-RESULT is the symbol `list',
this function returns a list of all the objects
that satisfy the criterion.
```

does not make a hyperlink to the documentation, irrelevant here, of the function `list`.

- Don't write key sequences directly in documentation strings. Instead, use the '`\\[...]`' construct to stand for them. For example, instead of writing '`C-f`', write the construct '`\\[forward-char]`'. When Emacs displays the documentation string, it substitutes whatever key is currently bound to `forward-char`. (This is normally '`C-f`', but it may be some other character if the user has moved key bindings.) See Section 23.3 [Keys in Documentation], page 427.
- In documentation strings for a major mode, you will want to refer to the key bindings of that mode's local map, rather than global ones. Therefore, use the construct '`\\<...>`' once in the documentation string to specify which key map to use. Do this before the first use of '`\\[...]`'. The text inside the '`\\<...>`' should be the name of the variable containing the local keymap for the major mode.

  It is not practical to use '`\\[...]`' very many times, because display of the documentation string will become slow. So use this to describe the most important commands in your major mode, and then use '`\\{...}`' to display the rest of the mode's keymap.

A.4 Tips on Writing Comments

We recommend these conventions for where to put comments and how to indent them:

'`;`' Comments that start with a single semicolon, '`;`', should all be aligned to the same column on the right of the source code. Such comments usually explain how the code on the same line does its job. In Lisp mode and related modes, the *M-;* (`indent-for-comment`) command automatically inserts such a '`;`' in the right place, or aligns such a comment if it is already present.

This and following examples are taken from the Emacs sources.

```
(setq base-version-list                 ; there was a base
      (assoc (substring fn 0 start-vn)  ; version to which
             file-version-assoc-list))  ; this looks like
                                        ; a subversion
```

';;' Comments that start with two semicolons, ';;', should be aligned to the same level of indentation as the code. Such comments usually describe the purpose of the following lines or the state of the program at that point. For example:

```
(prog1 (setq auto-fill-function
             ...
             ...
  ;; update mode line
  (force-mode-line-update)))
```

Every function that has no documentation string (presumably one that is used only internally within the package it belongs to), should have instead a two-semicolon comment right before the function, explaining what the function does and how to call it properly. Explain precisely what each argument means and how the function interprets its possible values.

';;;' Comments that start with three semicolons, ';;;', should start at the left margin. Such comments are used outside function definitions to make general statements explaining the design principles of the program. For example:

```
;;; This Lisp code is run in Emacs
;;; when it is to operate as a server
;;; for other processes.
```

Another use for triple-semicolon comments is for commenting out lines within a function. We use triple-semicolons for this precisely so that they remain at the left margin.

```
(defun foo (a)
;;; This is no longer necessary.
;;;  (force-mode-line-update)
  (message "Finished with %s" a))
```

';;;;' Comments that start with four semicolons, ';;;;', should be aligned to the left margin and are used for headings of major sections of a program. For example:

```
;;;; The kill ring
```

The indentation commands of the Lisp modes in Emacs, such as *M-;* (`indent-for-comment`) and TAB (`lisp-indent-line`), automatically indent comments according to these conventions, depending on the number of semicolons. See section "Manipulating Comments" in *The GNU Emacs Manual*.

A.5 Conventional Headers for Emacs Libraries

Emacs has conventions for using special comments in Lisp libraries to divide them into sections and give information such as who wrote them. This section explains these conventions. First, an example:

```
;;; lisp-mnt.el --- minor mode for Emacs Lisp maintainers

;; Copyright (C) 1992 Free Software Foundation, Inc.

;; Author: Eric S. Raymond <esr@snark.thyrsus.com>
;; Maintainer: Eric S. Raymond <esr@snark.thyrsus.com>
;; Created: 14 Jul 1992
;; Version: 1.2
;; Keywords: docs

;; This file is part of GNU Emacs.
...
;; Free Software Foundation, Inc., 59 Temple Place - Suite 330,
;; Boston, MA 02111-1307, USA.
```

The very first line should have this format:

;;; *filename* --- *description*

The description should be complete in one line.

After the copyright notice come several *header comment* lines, each beginning with '`;;` *header-name*`:`'. Here is a table of the conventional possibilities for *header-name*:

'`Author`' This line states the name and net address of at least the principal author of the library.

If there are multiple authors, you can list them on continuation lines led by `;;` and a tab character, like this:

```
;; Author: Ashwin Ram <Ram-Ashwin@cs.yale.edu>
;;      Dave Sill <de5@ornl.gov>
;;      Dave Brennan <brennan@hal.com>
;;      Eric Raymond <esr@snark.thyrsus.com>
```

'`Maintainer`'
This line should contain a single name/address as in the Author line, or an address only, or the string '`FSF`'. If there is no maintainer line, the person(s) in the Author field are presumed to be the maintainers. The example above is mildly bogus because the maintainer line is redundant.

The idea behind the '`Author`' and '`Maintainer`' lines is to make possible a Lisp function to "send mail to the maintainer" without having to mine the name out by hand.

Be sure to surround the network address with '`<...>`' if you include the person's full name as well as the network address.

'`Created`' This optional line gives the original creation date of the file. For historical interest only.

'`Version`' If you wish to record version numbers for the individual Lisp program, put them in this line.

'`Adapted-By`'
In this header line, place the name of the person who adapted the library for installation (to make it fit the style conventions, for example).

'`Keywords`'
This line lists keywords for the `finder-by-keyword` help command. Please use that command to see a list of the meaningful keywords.

This field is important; it's how people will find your package when they're looking for things by topic area. To separate the keywords, you can use spaces, commas, or both.

Just about every Lisp library ought to have the '`Author`' and '`Keywords`' header comment lines. Use the others if they are appropriate. You can also put in header lines with other header names—they have no standard meanings, so they can't do any harm.

We use additional stylized comments to subdivide the contents of the library file. Here is a table of them:

'`;;; Commentary:`'
This begins introductory comments that explain how the library works. It should come right after the copying permissions, terminated by a '`Change Log`', '`History`' or '`Code`' comment line. This text is used by the Finder package, so it should make sense in that context.

'`;;; Documentation`'
This has been used in some files in place of '`;;; Commentary:`', but '`;;; Commentary:`' is preferred.

'`;;; Change Log:`'
This begins change log information stored in the library file (if you store the change history there). For most of the Lisp files distributed with Emacs, the change history is kept in the file '`ChangeLog`' and not in the source file at all; these files do not have a '`;;; Change Log:`' line.

'`;;; Code:`'
This begins the actual code of the program.

‘`;;;` *filename* `ends here`’
: This is the *footer line*; it appears at the very end of the file. Its purpose is to enable people to detect truncated versions of the file from the lack of a footer line.

Appendix B GNU Emacs Internals

This chapter describes how the runnable Emacs executable is dumped with the preloaded Lisp libraries in it, how storage is allocated, and some internal aspects of GNU Emacs that may be of interest to C programmers.

B.1 Building Emacs

This section explains the steps involved in building the Emacs executable. You don't have to know this material to build and install Emacs, since the makefiles do all these things automatically. This information is pertinent to Emacs maintenance.

Compilation of the C source files in the '`src`' directory produces an executable file called '`temacs`', also called a *bare impure Emacs*. It contains the Emacs Lisp interpreter and I/O routines, but not the editing commands.

The command '`temacs -l loadup`' uses '`temacs`' to create the real runnable Emacs executable. These arguments direct '`temacs`' to evaluate the Lisp files specified in the file '`loadup.el`'. These files set up the normal Emacs editing environment, resulting in an Emacs that is still impure but no longer bare.

It takes a substantial time to load the standard Lisp files. Luckily, you don't have to do this each time you run Emacs; '`temacs`' can dump out an executable program called '`emacs`' that has these files preloaded. '`emacs`' starts more quickly because it does not need to load the files. This is the Emacs executable that is normally installed.

To create '`emacs`', use the command '`temacs -batch -l loadup dump`'. The purpose of '`-batch`' here is to prevent '`temacs`' from trying to initialize any of its data on the terminal; this ensures that the tables of terminal information are empty in the dumped Emacs. The argument '`dump`' tells '`loadup.el`' to dump a new executable named '`emacs`'.

Some operating systems don't support dumping. On those systems, you must start Emacs with the '`temacs -l loadup`' command each time you use it. This takes a substantial time, but since you need to start Emacs once a day at most—or once a week if you never log out—the extra time is not too severe a problem.

You can specify additional files to preload by writing a library named '`site-load.el`' that loads them. You may need to increase the value of `PURESIZE`, in '`src/puresize.h`', to make room for the additional data. (Try adding increments of 20000 until it is big enough.) However, the advantage of preloading additional files decreases as machines get faster. On modern machines, it is usually not advisable.

After '`loadup.el`' reads '`site-load.el`', it finds the documentation strings for primitive and preloaded functions (and variables) in the file '`etc/DOC`' where they are stored, by calling `Snarf-documentation` (see Section 23.2 [Accessing Documentation], page 424).

You can specify other Lisp expressions to execute just before dumping by putting them in a library named '`site-init.el`'. This file is executed after the documentation strings are found.

If you want to preload function or variable definitions, there are three ways you can do this and make their documentation strings accessible when you subsequently run Emacs:

- Arrange to scan these files when producing the '`etc/DOC`' file, and load them with '`site-load.el`'.
- Load the files with '`site-init.el`', then copy the files into the installation directory for Lisp files when you install Emacs.
- Specify a non-`nil` value for `byte-compile-dynamic-docstrings` as a local variable in each these files, and load them with either '`site-load.el`' or '`site-init.el`'. (This method has the drawback that the documentation strings take up space in Emacs all the time.)

It is not advisable to put anything in '`site-load.el`' or '`site-init.el`' that would alter any of the features that users expect in an ordinary unmodified Emacs. If you feel you must override normal features for your site, do it with '`default.el`', so that users can override your changes if they wish. See Section 37.1.1 [Start-up Summary], page 711.

dump-emacs *to-file from-file* Function

This function dumps the current state of Emacs into an executable file *to-file*. It takes symbols from *from-file* (this is normally the executable file '`temacs`').

If you want to use this function in an Emacs that was already dumped, you must run Emacs with '`-batch`'.

B.2 Pure Storage

Emacs Lisp uses two kinds of storage for user-created Lisp objects: *normal storage* and *pure storage*. Normal storage is where all the new data created during an Emacs session is kept; see the following section for information on normal storage. Pure storage is used for certain data in the preloaded standard Lisp files—data that should never change during actual use of Emacs.

Pure storage is allocated only while '`temacs`' is loading the standard preloaded Lisp libraries. In the file '`emacs`', it is marked as read-only (on operating systems that permit this), so that the memory space can be shared by all the Emacs jobs running on the machine at once. Pure storage is not expandable; a fixed amount is allocated when Emacs is compiled, and if that is not sufficient for the preloaded libraries, '`temacs`' crashes. If that happens, you must increase the compilation parameter `PURESIZE` in the file '`src/puresize.h`'. This normally won't happen unless you try to preload additional libraries or add features to the standard ones.

purecopy *object* Function

This function makes a copy of *object* in pure storage and returns it. It copies strings by simply making a new string with the same characters in pure storage. It recursively copies the contents of vectors and cons cells. It does not make copies of other objects such as symbols, but just returns them unchanged. It signals an error if asked to copy markers.

This function is a no-op except while Emacs is being built and dumped; it is usually called only in the file '`emacs/lisp/loaddefs.el`', but a few packages call it just in case you decide to preload them.

pure-bytes-used Variable

The value of this variable is the number of bytes of pure storage allocated so far. Typically, in a dumped Emacs, this number is very close to the total amount of pure storage available—if it were not, we would preallocate less.

purify-flag Variable

This variable determines whether `defun` should make a copy of the function definition in pure storage. If it is non-`nil`, then the function definition is copied into pure storage.

This flag is `t` while loading all of the basic functions for building Emacs initially (allowing those functions to be sharable and non-collectible). Dumping Emacs as an executable always writes `nil` in this variable, regardless of the value it actually has before and after dumping.

You should not change this flag in a running Emacs.

B.3 Garbage Collection

When a program creates a list or the user defines a new function (such as by loading a library), that data is placed in normal storage. If normal storage runs low, then Emacs asks the operating system to allocate more memory in blocks of 1k bytes. Each block is used for one type of Lisp object, so symbols, cons cells, markers, etc., are segregated in distinct blocks in memory. (Vectors, long strings, buffers and certain other editing types, which are fairly large, are allocated in individual blocks, one per object, while small strings are packed into blocks of 8k bytes.)

It is quite common to use some storage for a while, then release it by (for example) killing a buffer or deleting the last pointer to an object. Emacs provides a *garbage collector* to reclaim this abandoned storage. (This name is traditional, but "garbage recycler" might be a more intuitive metaphor for this facility.)

The garbage collector operates by finding and marking all Lisp objects that are still accessible to Lisp programs. To begin with, it assumes all the symbols, their values and associated function definitions, and any data

presently on the stack, are accessible. Any objects that can be reached indirectly through other accessible objects are also accessible.

When marking is finished, all objects still unmarked are garbage. No matter what the Lisp program or the user does, it is impossible to refer to them, since there is no longer a way to reach them. Their space might as well be reused, since no one will miss them. The second ("sweep") phase of the garbage collector arranges to reuse them.

The sweep phase puts unused cons cells onto a *free list* for future allocation; likewise for symbols and markers. It compacts the accessible strings so they occupy fewer 8k blocks; then it frees the other 8k blocks. Vectors, buffers, windows, and other large objects are individually allocated and freed using `malloc` and `free`.

> **Common Lisp note:** Unlike other Lisps, GNU Emacs Lisp does not call the garbage collector when the free list is empty. Instead, it simply requests the operating system to allocate more storage, and processing continues until `gc-cons-threshold` bytes have been used.
>
> This means that you can make sure that the garbage collector will not run during a certain portion of a Lisp program by calling the garbage collector explicitly just before it (provided that portion of the program does not use so much space as to force a second garbage collection).

garbage-collect Command

This command runs a garbage collection, and returns information on the amount of space in use. (Garbage collection can also occur spontaneously if you use more than `gc-cons-threshold` bytes of Lisp data since the previous garbage collection.)

`garbage-collect` returns a list containing the following information:

```
((used-conses . free-conses)
 (used-syms . free-syms)
 (used-miscs . free-miscs)
 used-string-chars
 used-vector-slots
 (used-floats . free-floats)
 (used-intervals . free-intervals))
```

Here is an example:

```
(garbage-collect)
     ⇒ ((106886 . 13184) (9769 . 0)
                (7731 . 4651) 347543 121628
                (31 . 94) (1273 . 168))
```

Here is a table explaining each element:

used-conses
: The number of cons cells in use.

free-conses
: The number of cons cells for which space has been obtained from the operating system, but that are not currently being used.

used-syms
: The number of symbols in use.

free-syms
: The number of symbols for which space has been obtained from the operating system, but that are not currently being used.

used-miscs
: The number of miscellaneous objects in use. These include markers and overlays, plus certain objects not visible to users.

free-miscs
: The number of miscellaneous objects for which space has been obtained from the operating system, but that are not currently being used.

used-string-chars
: The total size of all strings, in characters.

used-vector-slots
: The total number of elements of existing vectors.

used-floats
: The number of floats in use.

free-floats
: The number of floats for which space has been obtained from the operating system, but that are not currently being used.

used-intervals
: The number of intervals in use. Intervals are an internal data structure used for representing text properties.

free-intervals
: The number of intervals for which space has been obtained from the operating system, but that are not currently being used.

garbage-collection-messages User Option

If this variable is non-`nil`, Emacs displays a message at the beginning and end of garbage collection. The default value is `nil`, meaning there are no such messages.

gc-cons-threshold User Option

The value of this variable is the number of bytes of storage that must be allocated for Lisp objects after one garbage collection in order to trigger

another garbage collection. A cons cell counts as eight bytes, a string as one byte per character plus a few bytes of overhead, and so on; space allocated to the contents of buffers does not count. Note that the subsequent garbage collection does not happen immediately when the threshold is exhausted, but only the next time the Lisp evaluator is called.

The initial threshold value is 400,000. If you specify a larger value, garbage collection will happen less often. This reduces the amount of time spent garbage collecting, but increases total memory use. You may want to do this when running a program that creates lots of Lisp data.

You can make collections more frequent by specifying a smaller value, down to 10,000. A value less than 10,000 will remain in effect only until the subsequent garbage collection, at which time `garbage-collect` will set the threshold back to 10,000.

The value return by `garbage-collect` describes the amount of memory used by Lisp data, broken down by data type. By contrast, the function `memory-limit` provides information on the total amount of memory Emacs is currently using.

memory-limit Function

This function returns the address of the last byte Emacs has allocated, divided by 1024. We divide the value by 1024 to make sure it fits in a Lisp integer.

You can use this to get a general idea of how your actions affect the memory usage.

B.4 Memory Usage

These functions and variables give information about the total amount of memory allocation that Emacs has done, broken down by data type. Note the difference between these and the values returned by `(garbage-collect)`; those count objects that currently exist, but these count the number or size of all allocations, including those for objects that have since been freed.

cons-cells-consed Variable

The total number of cons cells that have been allocated so far in this Emacs session.

floats-consed Variable

The total number of floats that have been allocated so far in this Emacs session.

vector-cells-consed Variable

The total number of vector cells that have been allocated so far in this Emacs session.

symbols-consed Variable
The total number of symbols that have been allocated so far in this Emacs session.

string-chars-consed Variable
The total number of string characters that have been allocated so far in this Emacs session.

misc-objects-consed Variable
The total number of miscellaneous objects that have been allocated so far in this Emacs session. These include markers and overlays, plus certain objects not visible to users.

intervals-consed Variable
The total number of intervals that have been allocated so far in this Emacs session.

B.5 Writing Emacs Primitives

Lisp primitives are Lisp functions implemented in C. The details of interfacing the C function so that Lisp can call it are handled by a few C macros. The only way to really understand how to write new C code is to read the source, but we can explain some things here.

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

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

  if (NULL (args))
    return Qnil;

  args_left = args;
  GCPRO1 (args_left);
```

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

    UNGCPRO;
    return val;
  }
```

Let's start with a precise explanation of the arguments to the `DEFUN` macro. Here is a template for them:

`DEFUN` (*lname*, *fname*, *sname*, *min*, *max*, *interactive*, *doc*)

lname This is the name of the Lisp symbol to define as the function name; in the example above, it is `or`.

fname This is the C function name for this function. This is the name that is used in C code for calling the function. The name is, by convention, '`F`' prepended to the Lisp name, with all dashes ('`-`') in the Lisp name changed to underscores. Thus, to call this function from C code, call `For`. Remember that the arguments must be of type `Lisp_Object`; various macros and functions for creating values of type `Lisp_Object` are declared in the file '`lisp.h`'.

sname This is a C variable name to use for a structure that holds the data for the subr object that represents the function in Lisp. This structure conveys the Lisp symbol name to the initialization routine that will create the symbol and store the subr object as its definition. By convention, this name is always *fname* with '`F`' replaced with '`S`'.

min This is the minimum number of arguments that the function requires. The function `or` allows a minimum of zero arguments.

max This is the maximum number of arguments that the function accepts, if there is a fixed maximum. Alternatively, it can be `UNEVALLED`, indicating a special form that receives unevaluated arguments, or `MANY`, indicating an unlimited number of evaluated arguments (the equivalent of `&rest`). Both `UNEVALLED` and `MANY` are macros. If *max* is a number, it may not be less than *min* and it may not be greater than seven.

interactive
This is an interactive specification, a string such as might be used as the argument of `interactive` in a Lisp function. In

the case of `or`, it is 0 (a null pointer), indicating that `or` cannot be called interactively. A value of "" indicates a function that should receive no arguments when called interactively.

doc This is the documentation string. It is written just like a documentation string for a function defined in Lisp, except you must write '`\n\`' at the end of each line. In particular, the first line should be a single sentence.

After the call to the `DEFUN` macro, you must write the argument name list that every C function must have, followed by ordinary C declarations for the arguments. For a function with a fixed maximum number of arguments, declare a C argument for each Lisp argument, and give them all type `Lisp_Object`. When a Lisp function has no upper limit on the number of arguments, its implementation in C actually receives exactly two arguments: the first is the number of Lisp arguments, and the second is the address of a block containing their values. They have types `int` and `Lisp_Object *`.

Within the function `For` itself, note the use of the macros `GCPRO1` and `UNGCPRO`. `GCPRO1` is used to "protect" a variable from garbage collection—to inform the garbage collector that it must look in that variable and regard its contents as an accessible object. This is necessary whenever you call `Feval` or anything that can directly or indirectly call `Feval`. At such a time, any Lisp object that you intend to refer to again must be protected somehow. `UNGCPRO` cancels the protection of the variables that are protected in the current function. It is necessary to do this explicitly.

For most data types, it suffices to protect at least one pointer to the object; as long as the object is not recycled, all pointers to it remain valid. This is not so for strings, because the garbage collector can move them. When the garbage collector moves a string, it relocates all the pointers it knows about; any other pointers become invalid. Therefore, you must protect all pointers to strings across any point where garbage collection may be possible.

The macro `GCPRO1` protects just one local variable. If you want to protect two, use `GCPRO2` instead; repeating `GCPRO1` will not work. Macros `GCPRO3` and `GCPRO4` also exist.

These macros implicitly use local variables such as `gcpro1`; you must declare these explicitly, with type `struct gcpro`. Thus, if you use `GCPRO2`, you must declare `gcpro1` and `gcpro2`. Alas, we can't explain all the tricky details here.

You must not use C initializers for static or global variables unless they are never written once Emacs is dumped. These variables with initializers are allocated in an area of memory that becomes read-only (on certain operating systems) as a result of dumping Emacs. See Section B.2 [Pure Storage], page 794.

Do not use static variables within functions—place all static variables at top level in the file. This is necessary because Emacs on some operating

systems defines the keyword `static` as a null macro. (This definition is used because those systems put all variables declared static in a place that becomes read-only after dumping, whether they have initializers or not.)

Defining the C function is not enough to make a Lisp primitive available; you must also create the Lisp symbol for the primitive and store a suitable subr object in its function cell. The code looks like this:

```
defsubr (&subr-structure-name);
```

Here *subr-structure-name* is the name you used as the third argument to `DEFUN`.

If you add a new primitive to a file that already has Lisp primitives defined in it, find the function (near the end of the file) named `syms_of_`*something*, and add the call to `defsubr` there. If the file doesn't have this function, or if you create a new file, add to it a `syms_of_`*filename* (e.g., `syms_of_myfile`). Then find the spot in '`emacs.c`' where all of these functions are called, and add a call to `syms_of_`*filename* there.

The function `syms_of_`*filename* is also the place to define any C variables that are to be visible as Lisp variables. `DEFVAR_LISP` makes a C variable of type `Lisp_Object` visible in Lisp. `DEFVAR_INT` makes a C variable of type `int` visible in Lisp with a value that is always an integer. `DEFVAR_BOOL` makes a C variable of type `int` visible in Lisp with a value that is either `t` or `nil`.

If you define a file-scope C variable of type `Lisp_Object`, you must protect it for garbage-collection by calling `staticpro` in `syms_of_`*filename*, like this:

```
staticpro (&variable);
```

Here is another example function, with more complicated arguments. This comes from the code in '`window.c`', and it demonstrates the use of macros and functions to manipulate Lisp objects.

```
DEFUN ("coordinates-in-window-p", Fcoordinates_in_window_p,
  Scoordinates_in_window_p, 2, 2,
  "xSpecify coordinate pair: \nXExpression which evals to window: ",
  "Return non-nil if COORDINATES is in WINDOW.\n\
COORDINATES is a cons of the form (X . Y), X and Y being distances\n\
...
If they are on the border between WINDOW and its right sibling,\n\
   `vertical-line' is returned.")
  (coordinates, window)
     register Lisp_Object coordinates, window;
{
  int x, y;
```

```
    CHECK_LIVE_WINDOW (window, 0);
    CHECK_CONS (coordinates, 1);
    x = XINT (Fcar (coordinates));
    y = XINT (Fcdr (coordinates));

    switch (coordinates_in_window (XWINDOW (window), &x, &y))
      {
      case 0:  /* NOT in window at all. */
        return Qnil;

      case 1:  /* In text part of window. */
        return Fcons (make_number (x), make_number (y));

      case 2:  /* In mode line of window. */
        return Qmode_line;

      case 3:  /* On right border of window.  */
        return Qvertical_line;

      default:
        abort ();
      }
  }
```

Note that C code cannot call functions by name unless they are defined in C. The way to call a function written in Lisp is to use `Ffuncall`, which embodies the Lisp function `funcall`. Since the Lisp function `funcall` accepts an unlimited number of arguments, in C it takes two: the number of Lisp-level arguments, and a one-dimensional array containing their values. The first Lisp-level argument is the Lisp function to call, and the rest are the arguments to pass to it. Since `Ffuncall` can call the evaluator, you must protect pointers from garbage collection around the call to `Ffuncall`.

The C functions `call0`, `call1`, `call2`, and so on, provide handy ways to call a Lisp function conveniently with a fixed number of arguments. They work by calling `Ffuncall`.

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

B.6 Object Internals

GNU Emacs Lisp manipulates many different types of data. The actual data are stored in a heap and the only access that programs have to it is through pointers. Pointers are thirty-two bits wide in most implementations. Depending on the operating system and type of machine for which you compile Emacs, twenty-eight bits are used to address the object, and the remaining four bits are used for a GC mark bit and the tag that identifies the object's type.

Because Lisp objects are represented as tagged pointers, it is always possible to determine the Lisp data type of any object. The C data type `Lisp_Object` can hold any Lisp object of any data type. Ordinary variables have type `Lisp_Object`, which means they can hold any type of Lisp value; you can determine the actual data type only at run time. The same is true for function arguments; if you want a function to accept only a certain type of argument, you must check the type explicitly using a suitable predicate (see Section 2.5 [Type Predicates], page 36).

B.6.1 Buffer Internals

Buffers contain fields not directly accessible by the Lisp programmer. We describe them here, naming them by the names used in the C code. Many are accessible indirectly in Lisp programs via Lisp primitives.

`name` The buffer name is a string that names the buffer. It is guaranteed to be unique. See Section 26.3 [Buffer Names], page 482.

`save_modified`
This field contains the time when the buffer was last saved, as an integer. See Section 26.5 [Buffer Modification], page 485.

`modtime` This field contains the modification time of the visited file. It is set when the file is written or read. Every time the buffer is written to the file, this field is compared to the modification time of the file. See Section 26.5 [Buffer Modification], page 485.

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

`last_window_start`
This field contains the `window-start` position in the buffer as of the last time the buffer was displayed in a window.

`undo_list`
This field points to the buffer's undo list. See Section 31.9 [Undo], page 590.

`syntax_table_v`
This field contains the syntax table for the buffer. See Chapter 34 [Syntax Tables], page 669.

`downcase_table`
This field contains the conversion table for converting text to lower case. See Section 4.9 [Case Tables], page 72.

`upcase_table`
This field contains the conversion table for converting text to upper case. See Section 4.9 [Case Tables], page 72.

`case_canon_table`
: This field contains the conversion table for canonicalizing text for case-folding search. See Section 4.9 [Case Tables], page 72.

`case_eqv_table`
: This field contains the equivalence table for case-folding search. See Section 4.9 [Case Tables], page 72.

`display_table`
: This field contains the buffer's display table, or `nil` if it doesn't have one. See Section 38.14 [Display Tables], page 762.

`markers`
: This field contains the chain of all markers that currently point into the buffer. Deletion of text in the buffer, and motion of the buffer's gap, must check each of these markers and perhaps update it. See Chapter 30 [Markers], page 565.

`backed_up`
: This field is a flag that tells whether a backup file has been made for the visited file of this buffer.

`mark`
: This field contains the mark for the buffer. The mark is a marker, hence it is also included on the list `markers`. See Section 30.7 [The Mark], page 570.

`mark_active`
: This field is non-`nil` if the buffer's mark is active.

`local_var_alist`
: This field contains the association list describing the buffer-local variable bindings of this buffer, not including the built-in buffer-local bindings that have special slots in the buffer object. (Those slots are omitted from this table.) See Section 10.10 [Buffer-Local Variables], page 161.

`base_buffer`
: This field holds the buffer's base buffer (if it is an indirect buffer), or `nil`.

`keymap`
: This field holds the buffer's local keymap. See Chapter 21 [Keymaps], page 361.

`overlay_center`
: This field holds the current overlay center position. See Section 38.8 [Overlays], page 748.

`overlays_before`
: This field holds a list of the overlays in this buffer that end at or before the current overlay center position. They are sorted in order of decreasing end position.

`overlays_after`
: This field holds a list of the overlays in this buffer that end after the current overlay center position. They are sorted in order of increasing beginning position.

`enable_multibyte_characters`
: This field holds the buffer's local value of `enable-multibyte-characters`—either `t` or `nil`.

B.6.2 Window Internals

Windows have the following accessible fields:

`frame`
: The frame that this window is on.

`mini_p`
: Non-`nil` if this window is a minibuffer window.

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

`dedicated`
: Non-`nil` if this window is dedicated to its buffer.

`pointm`
: This is the value of point in the current buffer when this window is selected; when it is not selected, it retains its previous value.

`start`
: The position in the buffer that is the first character to be displayed in the window.

`force_start`
: If this flag is non-`nil`, it says that the window has been scrolled explicitly by the Lisp program. This affects what the next redisplay does if point is off the screen: instead of scrolling the window to show the text around point, it moves point to a location that is on the screen.

`last_modified`
: The `modified` field of the window's buffer, as of the last time a redisplay completed in this window.

`last_point`
: The buffer's value of point, as of the last time a redisplay completed in this window.

`left`
: This is the left-hand edge of the window, measured in columns. (The leftmost column on the screen is column 0.)

`top`
: This is the top edge of the window, measured in lines. (The top line on the screen is line 0.)

`height`
: The height of the window, measured in lines.

`width`
: The width of the window, measured in columns.

`next` This is the window that is the next in the chain of siblings. It is `nil` in a window that is the rightmost or bottommost of a group of siblings.

`prev` This is the window that is the previous in the chain of siblings. It is `nil` in a window that is the leftmost or topmost of a group of siblings.

`parent` Internally, Emacs arranges windows in a tree; each group of siblings has a parent window whose area includes all the siblings. This field points to a window's parent.

Parent windows do not display buffers, and play little role in display except to shape their child windows. Emacs Lisp programs usually have no access to the parent windows; they operate on the windows at the leaves of the tree, which actually display buffers.

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

`use_time` This is the last time that the window was selected. The function `get-lru-window` uses this field.

`display_table`
The window's display table, or `nil` if none is specified for it.

`update_mode_line`
Non-`nil` means this window's mode line needs to be updated.

`base_line_number`
The line number of a certain position in the buffer, or `nil`. This is used for displaying the line number of point in the mode line.

`base_line_pos`
The position in the buffer for which the line number is known, or `nil` meaning none is known.

`region_showing`
If the region (or part of it) is highlighted in this window, this field holds the mark position that made one end of that region. Otherwise, this field is `nil`.

B.6.3 Process Internals

The fields of a process are:

`name` A string, the name of the process.

`command` A list containing the command arguments that were used to start this process.

`filter` A function used to accept output from the process instead of a buffer, or `nil`.

`sentinel` A function called whenever the process receives a signal, or `nil`.

`buffer` The associated buffer of the process.

`pid` An integer, the Unix process ID.

`childp` A flag, non-`nil` if this is really a child process. It is `nil` for a network connection.

`mark` A marker indicating the position of the end of the last output from this process inserted into the buffer. This is often but not always the end of the buffer.

`kill_without_query`
: If this is non-`nil`, killing Emacs while this process is still running does not ask for confirmation about killing the process.

`raw_status_low`
`raw_status_high`
: These two fields record 16 bits each of the process status returned by the `wait` system call.

`status` The process status, as `process-status` should return it.

`tick`
`update_tick`
: If these two fields are not equal, a change in the status of the process needs to be reported, either by running the sentinel or by inserting a message in the process buffer.

`pty_flag` Non-`nil` if communication with the subprocess uses a PTY; `nil` if it uses a pipe.

`infd` The file descriptor for input from the process.

`outfd` The file descriptor for output to the process.

`subtty` The file descriptor for the terminal that the subprocess is using. (On some systems, there is no need to record this, so the value is `nil`.)

`tty_name` The name of the terminal that the subprocess is using, or `nil` if it is using pipes.

Appendix C Standard Errors

Here is the complete list of the error symbols in standard Emacs, grouped by concept. The list includes each symbol's message (on the `error-message` property of the symbol) and a cross reference to a description of how the error can occur.

Each error symbol has an `error-conditions` property that is a list of symbols. Normally this list includes the error symbol itself and the symbol `error`. Occasionally it includes additional symbols, which are intermediate classifications, narrower than `error` but broader than a single error symbol. For example, all the errors in accessing files have the condition `file-error`. If we do not say here that a certain error symbol has additional error conditions, that means it has none.

As a special exception, the error symbol `quit` does not have the condition `error`, because quitting is not considered an error.

See Section 9.5.3 [Errors], page 138, for an explanation of how errors are generated and handled.

symbol *string*; *reference.*

`error` `"error"`
See Section 9.5.3 [Errors], page 138.

`quit` `"Quit"`
See Section 20.9 [Quitting], page 351.

`args-out-of-range`
`"Args out of range"`
See Chapter 6 [Sequences Arrays Vectors], page 97.

`arith-error`
`"Arithmetic error"`
See `/` and `%` in Chapter 3 [Numbers], page 43.

`beginning-of-buffer`
`"Beginning of buffer"`
See Section 29.2 [Motion], page 552.

`buffer-read-only`
`"Buffer is read-only"`
See Section 26.7 [Read Only Buffers], page 487.

`cyclic-function-indirection`
`"Symbol's chain of function indirections\`
`contains a loop"`
See Section 8.1.4 [Function Indirection], page 121.

`end-of-buffer`
`"End of buffer"`
See Section 29.2 [Motion], page 552.

`end-of-file`
: `"End of file during parsing"`
Note that this is not a `file-error` because it pertains to the Lisp reader, not to file I/O. See Section 18.3 [Input Functions], page 286.

`file-already-exists`
: This is a `file-error`.
See Section 24.4 [Writing to Files], page 440.

`file-date-error`
: This is a subcategory of `file-error`. It occurs when `copy-file` tries and fails to set the last-modification time of the output file.
See Section 24.7 [Changing Files], page 448.

`file-error`
: This error and its subcategories do not have error-strings, because the error message is constructed from the data items alone when the error condition `file-error` is present.
See Chapter 24 [Files], page 433.

`file-locked`
: This is a `file-error`.
See Section 24.5 [File Locks], page 441.

`file-supersession`
: This is a `file-error`.
See Section 26.6 [Modification Time], page 486.

`invalid-function`
: `"Invalid function"`
See Section 8.1.3 [Classifying Lists], page 121.

`invalid-read-syntax`
: `"Invalid read syntax"`
See Section 18.3 [Input Functions], page 286.

`invalid-regexp`
: `"Invalid regexp"`
See Section 33.2 [Regular Expressions], page 649.

`mark-inactive`
: `"Mark inactive"`
See Section 30.7 [The Mark], page 570.

`no-catch` `"No catch for tag"`
See Section 9.5.1 [Catch and Throw], page 135.

`scan-error`
: `"Scan error"`
This happens when certain syntax-parsing functions find invalid syntax or mismatched parentheses.

See Section 29.2.6 [List Motion], page 558, and Section 34.6 [Parsing Expressions], page 677.

`search-failed`
: `"Search failed"`
See Chapter 33 [Searching and Matching], page 647.

`setting-constant`
: `"Attempt to set a constant symbol"`
The values of the symbols `nil` and `t`, and any symbols that start with ':', may not be changed.
See Section 10.2 [Variables that Never Change], page 148.

`undefined-color`
: `"Undefined color"`
See Section 28.21 [Color Names], page 547.

`void-function`
: `"Symbol's function definition is void"`
See Section 11.8 [Function Cells], page 183.

`void-variable`
: `"Symbol's value as variable is void"`
See Section 10.7 [Accessing Variables], page 155.

`wrong-number-of-arguments`
: `"Wrong number of arguments"`
See Section 8.1.3 [Classifying Lists], page 121.

`wrong-type-argument`
: `"Wrong type argument"`
See Section 2.5 [Type Predicates], page 36.

These kinds of error, which are classified as special cases of `arith-error`, can occur on certain systems for invalid use of mathematical functions.

`domain-error`
: `"Arithmetic domain error"`
See Section 3.9 [Math Functions], page 56.

`overflow-error`
: `"Arithmetic overflow error"`
See Section 3.9 [Math Functions], page 56.

`range-error`
: `"Arithmetic range error"`
See Section 3.9 [Math Functions], page 56.

`singularity-error`
: `"Arithmetic singularity error"`
See Section 3.9 [Math Functions], page 56.

`underflow-error`
: `"Arithmetic underflow error"`
See Section 3.9 [Math Functions], page 56.

Appendix D Buffer-Local Variables

The table below lists the general-purpose Emacs variables that automatically become buffer-local in each buffer. Most become buffer-local only when set; a few of them are always local in every buffer. Many Lisp packages define such variables for their internal use, but we don't try to list them all here.

`abbrev-mode`
: See Chapter 35 [Abbrevs], page 683.

`auto-fill-function`
: See Section 31.14 [Auto Filling], page 598.

`buffer-auto-save-file-name`
: See Section 25.2 [Auto-Saving], page 472.

`buffer-backed-up`
: See Section 25.1 [Backup Files], page 467.

`buffer-display-count`
: See Section 27.7 [Displaying Buffers], page 504.

`buffer-display-table`
: See Section 38.14 [Display Tables], page 762.

`buffer-file-format`
: See Section 24.12 [Format Conversion], page 463.

`buffer-file-name`
: See Section 26.4 [Buffer File Name], page 483.

`buffer-file-number`
: See Section 26.4 [Buffer File Name], page 483.

`buffer-file-truename`
: See Section 26.4 [Buffer File Name], page 483.

`buffer-file-type`
: See Section 32.10.9 [MS-DOS File Types], page 644.

`buffer-invisibility-spec`
: See Section 38.4 [Invisible Text], page 742.

`buffer-offer-save`
: See Section 24.2 [Saving Buffers], page 436.

`buffer-read-only`
: See Section 26.7 [Read Only Buffers], page 487.

`buffer-saved-size`
: See Section 29.1 [Point], page 551.

`buffer-undo-list`
: See Section 31.9 [Undo], page 590.

`cache-long-line-scans`
: See Section 29.2.4 [Text Lines], page 554.

`case-fold-search`
: See Section 33.7 [Searching and Case], page 665.

`ctl-arrow`
: See Section 38.13 [Usual Display], page 760.

`comment-column`
: See section "Comments" in *The GNU Emacs Manual*.

`default-directory`
: See Section 37.3 [System Environment], page 718.

`defun-prompt-regexp`
: See Section 29.2.6 [List Motion], page 558.

`enable-multibyte-characters`
: Chapter 32 [Non-ASCII Characters], page 629.

`fill-column`
: See Section 31.14 [Auto Filling], page 598.

`goal-column`
: See section "Moving Point" in *The GNU Emacs Manual*.

`left-margin`
: See Section 31.17 [Indentation], page 604.

`local-abbrev-table`
: See Chapter 35 [Abbrevs], page 683.

`local-write-file-hooks`
: See Section 24.2 [Saving Buffers], page 436.

`major-mode`
: See Section 22.1.4 [Mode Help], page 401.

`mark-active`
: See Section 30.7 [The Mark], page 570.

`mark-ring`
: See Section 30.7 [The Mark], page 570.

`minor-modes`
: See Section 22.2 [Minor Modes], page 402.

`mode-line-buffer-identification`
: See Section 22.3.2 [Mode Line Variables], page 408.

`mode-line-format`
: See Section 22.3.1 [Mode Line Data], page 406.

`mode-line-modified`
: See Section 22.3.2 [Mode Line Variables], page 408.

`mode-line-process`
: See Section 22.3.2 [Mode Line Variables], page 408.

`mode-name`
: See Section 22.3.2 [Mode Line Variables], page 408.

`overwrite-mode`
: See Section 31.4 [Insertion], page 578.

`paragraph-separate`
: See Section 33.8 [Standard Regexps], page 666.

`paragraph-start`
: See Section 33.8 [Standard Regexps], page 666.

`point-before-scroll`
: Ued for communication between mouse commands and scroll-bar commands..

`require-final-newline`
: See Section 31.4 [Insertion], page 578.

`selective-display`
: See Section 38.5 [Selective Display], page 744.

`selective-display-ellipses`
: See Section 38.5 [Selective Display], page 744.

`tab-width`
: See Section 38.13 [Usual Display], page 760.

`truncate-lines`
: See Section 38.2 [Truncation], page 739.

`vc-mode` See Section 22.3.2 [Mode Line Variables], page 408.

Appendix E Standard Keymaps

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

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

`Buffer-menu-mode-map`
: A full keymap used by Buffer Menu mode.

`c-mode-map`
: A sparse keymap used by C mode.

`command-history-map`
: A full keymap used by Command History mode.

`ctl-x-4-map`
: A sparse keymap for subcommands of the prefix `C-x 4`.

`ctl-x-5-map`
: A sparse keymap for subcommands of the prefix `C-x 5`.

`ctl-x-map`
: A full keymap for `C-x` commands.

`debugger-mode-map`
: A full keymap used by Debugger mode.

`dired-mode-map`
: A full keymap for `dired-mode` buffers.

`edit-abbrevs-map`
: A sparse keymap used in `edit-abbrevs`.

`edit-tab-stops-map`
: A sparse keymap used in `edit-tab-stops`.

`electric-buffer-menu-mode-map`
: A full keymap used by Electric Buffer Menu mode.

`electric-history-map`
: A full keymap used by Electric Command History mode.

`emacs-lisp-mode-map`
: A sparse keymap used by Emacs Lisp mode.

`facemenu-menu`
: The keymap that displays the Text Properties menu.

`facemenu-background-menu`
: The keymap that displays the Background Color submenu of the Text Properties menu.

`facemenu-face-menu`
: The keymap that displays the Face submenu of the Text Properties menu.

`facemenu-foreground-menu`
: The keymap that displays the Foreground Color submenu of the Text Properties menu.

`facemenu-indentation-menu`
: The keymap that displays the Indentation submenu of the Text Properties menu.

`facemenu-justification-menu`
: The keymap that displays the Justification submenu of the Text Properties menu.

`facemenu-special-menu`
: The keymap that displays the Special Props submenu of the Text Properties menu.

`function-key-map`
: The keymap for translating keypad and function keys.
If there are none, then it contains an empty sparse keymap. See Section 37.8.2 [Translating Input], page 730.

`fundamental-mode-map`
: The local keymap for Fundamental mode.
It is empty and should not be changed.

`Helper-help-map`
: A full keymap used by the help utility package.
It has the same keymap in its value cell and in its function cell.

`Info-edit-map`
: A sparse keymap used by the `e` command of Info.

`Info-mode-map`
: A sparse keymap containing Info commands.

`isearch-mode-map`
: A keymap that defines the characters you can type within incremental search.

`key-translation-map`
: A keymap for translating keys. This one overrides ordinary key bindings, unlike `function-key-map`. See Section 37.8.2 [Translating Input], page 730.

`lisp-interaction-mode-map`
: A sparse keymap used by Lisp mode.

`lisp-mode-map`
: A sparse keymap used by Lisp mode.

`menu-bar-edit-menu`
: The keymap which displays the Edit menu in the menu bar.

`menu-bar-files-menu`
: The keymap which displays the Files menu in the menu bar.

`menu-bar-help-menu`
: The keymap which displays the Help menu in the menu bar.

`menu-bar-mule-menu`
: The keymap which displays the Mule menu in the menu bar.

`menu-bar-search-menu`
: The keymap which displays the Search menu in the menu bar.

`menu-bar-tools-menu`
: The keymap which displays the Tools menu in the menu bar.

`mode-specific-map`
: The keymap for characters following *C-c*. Note, this is in the global map. This map is not actually mode specific: its name was chosen to be informative for the user in *C-h b* (`display-bindings`), where it describes the main use of the *C-c* prefix key.

`occur-mode-map`
: A local keymap used by Occur mode.

`query-replace-map`
: A local keymap used for responses in `query-replace` and related commands; also for `y-or-n-p` and `map-y-or-n-p`. The functions that use this map do not support prefix keys; they look up one event at a time.

`text-mode-map`
: A sparse keymap used by Text mode.

`view-mode-map`
: A full keymap used by View mode.

Appendix F Standard Hooks

The following is a list of hook variables that let you provide functions to be called from within Emacs on suitable occasions.

Most of these variables have names ending with '`-hook`'. They are *normal hooks*, run by means of `run-hooks`. The value of such a hook is a list of functions; the functions are called with no arguments and their values are completely ignored. The recommended way to put a new function on such a hook is to call `add-hook`. See Section 22.6 [Hooks], page 420, for more information about using hooks.

The variables whose names end in '`-hooks`' or '`-functions`' are usually *abnormal hooks*; their values are lists of functions, but these functions are called in a special way (they are passed arguments, or their values are used). A few of these variables are actually normal hooks which were named before we established the convention that normal hooks' names should end in '`-hook`'.

The variables whose names end in '`-function`' have single functions as their values. (In older Emacs versions, some of these variables had names ending in '`-hook`' even though they were not normal hooks; however, we have renamed all of those.)

`activate-mark-hook`

`after-change-function`

`after-change-functions`

`after-init-hook`

`after-insert-file-functions`

`after-make-frame-hook`

`after-revert-hook`

`after-save-hook`

`auto-fill-function`

`auto-save-hook`

`before-change-function`

`before-change-functions`

`before-init-hook`

`before-make-frame-hook`

`before-revert-hook`

`blink-paren-function`

`buffer-access-fontify-functions`

`c-mode-hook`

`calendar-load-hook`

`change-major-mode-hook`

command-history-hook
command-line-functions
comment-indent-function
deactivate-mark-hook
diary-display-hook
diary-hook
dired-mode-hook
disabled-command-hook
echo-area-clear-hook
edit-picture-hook
electric-buffer-menu-mode-hook
electric-command-history-hook
electric-help-mode-hook
emacs-lisp-mode-hook
find-file-hooks
find-file-not-found-hooks
first-change-hook
fortran-comment-hook
fortran-mode-hook
ftp-setup-write-file-hooks
ftp-write-file-hook
indent-mim-hook
initial-calendar-window-hook
kill-buffer-hook
kill-buffer-query-functions
kill-emacs-hook
kill-emacs-query-functions
LaTeX-mode-hook
ledit-mode-hook
lisp-indent-function
lisp-interaction-mode-hook
lisp-mode-hook
list-diary-entries-hook
local-write-file-hooks
m2-mode-hook
mail-mode-hook
mail-setup-hook

```
mark-diary-entries-hook
medit-mode-hook
menu-bar-update-hook
minibuffer-setup-hook
minibuffer-exit-hook
news-mode-hook
news-reply-mode-hook
news-setup-hook
nongregorian-diary-listing-hook
nongregorian-diary-marking-hook
nroff-mode-hook
outline-mode-hook
plain-TeX-mode-hook
post-command-hook
pre-abbrev-expand-hook
pre-command-hook
print-diary-entries-hook
prolog-mode-hook
protect-innocence-hook
redisplay-end-trigger-functions
rmail-edit-mode-hook
rmail-mode-hook
rmail-summary-mode-hook
scheme-indent-hook
scheme-mode-hook
scribe-mode-hook
shell-mode-hook
shell-set-directory-error-hook
suspend-hook
suspend-resume-hook
temp-buffer-show-function
term-setup-hook
terminal-mode-hook
terminal-mode-break-hook
TeX-mode-hook
text-mode-hook
today-visible-calendar-hook
```

today-invisible-calendar-hook
vi-mode-hook
view-hook
window-configuration-change-hook
window-scroll-functions
window-setup-hook
window-size-change-functions
write-contents-hooks
write-file-hooks
write-region-annotate-functions

New Symbols Since the Previous Edition

A

B

C

D

E

F

H

I

K

L

M

N

O

P

R

S

T

W

Short Contents

Table of Contents