GNU Emacs Lisp Reference Manual

GNU Emacs Version 19
for Unix Users
Second Edition, June 1993

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

Second Edition
For Emacs Version 19,
Printed June, 1993.

ISBN 1-882114-40-X

Published by the Free Software Foundation
675 Massachusetts Avenue
Cambridge, MA 02139 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

1. 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.

2. 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.

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

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

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

c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

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

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

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

4. 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.

5. 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.
6. 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.
7. 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.
8. 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.

9. 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.

10. 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.

11. 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

12. 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.
13. 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) 19yy  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., 675 Mass Ave, Cambridge, MA 02139, 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) 19yy 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 describes Emacs Lisp, presuming 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 2.0.

1.1 Caveats

This manual has gone through numerous drafts. It is nearly complete but not flawless. There are a few sections which are not included, 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 references to VMS and all information relating Sunview. (The Free Software Foundation expends no effort on support for Sunview, since we believe users should use the free X window system rather than proprietary window systems.)

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. Also state the number of the edition which you are criticizing.

Please mail comments and corrections to

`bug-lisp-manual@prep.ai.mit.edu`

–Bil Lewis, Dan LaLiberte, Richard Stallman

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 superior 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 1960's at MIT's Project MAC. Eventually the implementors of the descendents of Maclisp came together and developed a standard for Lisp systems, called Common Lisp.

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

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" are used to refer to those routines in Lisp that convert textual representations of Lisp objects into actual 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 arguments or metasyntactic variables appear in this font or form: *first-number*.

1.3.2 `nil` and `t`

In Lisp, the symbol `nil` is overloaded with three 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 value `t`.

In Emacs Lisp, `nil` and `t` are special symbols that always evaluate to themselves. This is so that you do not need to quote them to use them as constants in a program. An attempt to change their values results in a `setting-constant` error. See Section 10.6 [Accessing Variables], page 138.

1.3.3 Evaluation Notation

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

```
(car '(1 2))
     ⇒ 1
```

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

When a form is a macro call, it expands into a new form for Lisp to evaluate. We show the result of the expansion with '↦'. We may or may not show the actual 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 which produces identical results. The exact equivalence of two forms is indicated with '≡'.

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

1.3.4 Printing Notation

Many of the examples in this manual print text when they are evaluated. If you execute the code from an example in a Lisp Interaction buffer (such as the buffer '`*scratch*`'), the printed text is inserted into the buffer. If the example is executed by other means (such as by evaluating the function `eval-region`), the text printed is usually displayed in the echo area. You should be aware that text displayed in the echo area is truncated to a single line.

In examples that print text, the printed text is indicated with '⊣', irrespective of how the form is executed. 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 cause errors to be signaled. In them, the error message (which always appears in the echo area) is shown on a line starting with '$\boxed{\text{error}}$'. Note that '$\boxed{\text{error}}$' itself does not appear in the echo area.

```
(+ 23 'x)
error  Wrong type argument: integer-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. In such cases, the entire contents of the buffer in question are included between two lines of dashes containing the buffer name. In addition, the location of point is shown as '⋆'. (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 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 parameters. The names used for the parameters are also used in the body of the description.

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

The keyword `&rest` (which will always be followed by a single parameter) indicates that any number of arguments can follow. The value of the single following parameter will be a list of all these 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 parameter 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. Parameters named *object* may be of any type. (See Chapter 2 [Types of Lisp Object], page 17, for a list of Emacs object types.) Parameters with other sorts of names (e.g., *new-file*) are discussed specifically in the description of the function. In some sections, features common to parameters of several functions are described at the beginning.

See Section 11.2 [Lambda Expressions], page 150, 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 parameters 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 1 (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, then *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 will 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 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, David Boyes, Alan Carroll, David A. Duff, Beverly Erlebacher, David Eckelkamp, Eirik Fuller, Eric Hanchrow, George Hartzell, Nathan Hess, Dan Jacobson, Jak Kirman, Bob Knighten, Frederick M. Korz, Joe Lammens, K. Richard Magill, Brian Marick, Roland McGrath, Skip Montanaro, John Gardiner Myers, Arnold D. Robbins, Raul Rockwell, Shinichirou Sugou, Kimmo Suominen, Edward Tharp, Bill Trost, 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*, and several special types, such as *buffer*, that are related to editing. (See Section 2.4 [Editing Types], page 30.)

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, it cannot be treated as a number because 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 remembers the type of any value you store in it.

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 `print`) 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. Most objects have more than one possible read syntax. Some types of object have no read syntax; 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, since it may not make sense to enter objects of these types directly in a Lisp program. For example, the buffer type does not have a read syntax. 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 a '`#<`' is encountered.

```
(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 103). 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 16.3 [Input Functions], page 224, 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 comment starts with a semicolon ('`;`') if it is not within a string or character constant, and continues to the end of line. Comments are discarded by the Lisp reader, and do not become part of the Lisp objects which represent the program within the Lisp system.

See Section A.4 [Comment Tips], page 619, 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 are provided in many Lisp implementations, in one form or another. The latter are unique to Emacs Lisp.

2.3.1 Integer Type

Integers are the only kind of number in GNU Emacs Lisp, version 18. The range of values for integers is -8388608 to 8388607 (24 bits; i.e., -2^{23}

to $2^{23} - 1$) on most machines, but is 25 or 26 bits on some systems. It is important to note that the Emacs Lisp arithmetic functions do not check for overflow. Thus `(1+ 8388607)` is -8388608 on 24-bit implementations.

The read syntax for numbers is a sequence of (base ten) digits with an optional sign. The printed representation produced by the Lisp interpreter never has a leading '+'.

```
-1               ; The integer -1.
1                ; The integer 1.
+1               ; Also the integer 1.
16777217         ; Also the integer 1!
                 ;   (on a 24-bit or 25-bit implementation)
```

See Chapter 3 [Numbers], page 39, for more information.

2.3.2 Floating Point Type

Emacs version 19 supports floating point numbers, if compiled with the macro `LISP_FLOAT_TYPE` defined. 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 39, 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 2.3.7 [String Type], page 26.

Characters in strings, buffers, and files are currently limited to the range of 0 to 255. If an arbitrary integer is used as a character for those purposes, only the lower eight bits are significant. Characters that represent keyboard input have a much wider range.

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 '\' to prevent Lisp mode from getting 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. Those values are 7, 8, 9, 10, 11, 12, 13, and 27 in decimal. Thus,

```
?\a ⇒ 7                 ; C-g
?\b ⇒ 8                 ; backspace, BS, C-h
?\t ⇒ 9                 ; tab, TAB, C-i
?\n ⇒ 10                ; newline, LFD, 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, but they have 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, either '?\^I' or '?\^i' may be used as the 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
```

For use in strings and buffers, you are limited to the control characters 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 characters include the 2**22 bit as well as the code for the non-control character. Ordinary terminals have no way of generating non-ASCII control characters, but you can generate them straightforwardly using an X terminal.

The DEL key can be considered and written as `Control-?`:

```
?\^? ⇒ 127

?\C-? ⇒ 127
```

When you represent control characters to be found in files or strings, we recommend the '`^`' syntax; but when you refer to keyboard input, we prefer the '`C-`' syntax. This 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 key. The integer that represents such a character has the 2**23 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 indicates a meta character, so 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 codes, '`\C-`', or 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 shift modifier is used in indicating the case of a character in special circumstances. The case of an ordinary letter is indicated by its character code as part of ASCII, but ASCII has no way to represent whether a control character is upper case or lower case. Emacs uses the 2**21 bit to indicate that the shift key was used for typing a control character. This distinction is possible only when you use X terminals or other special terminals; ordinary terminals do not indicate 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-`'. Thus, '`?\H-\M-\A-x`' represents `Alt-Hyper-Meta-x`. Numerically, the bit values are 2**18 for alt, 2**19 for super and 2**20 for hyper.

Finally, the most general read syntax consists of a question mark followed by a backslash and the character code in octal (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
```

A backslash is allowed, and harmless, preceding any character without a special escape meaning; thus, ‘?\A’ is equivalent to ‘?A’. There is no reason to use a backslash before most such characters. However, any of the characters ‘()\|;'`"#.,’ should be preceded by a backslash to avoid confusing the Emacs commands for editing Lisp code. Whitespace characters such as space, tab, newline and formfeed should also be preceded by a backslash. However, it is cleaner to use one of the easily readable escape sequences, such as ‘\t’, instead of an actual control character such as a tab.

2.3.4 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 and vectors. Vectors can hold elements of any type, but string elements must be characters in the range from 0 to 255. However, the characters in a string can have text properties; vectors do not support text properties even when their elements happen to be characters.

Lists, strings and vectors 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. Also, several functions, called sequence functions, accept any kind of sequence. For example, the function `elt` can be used to extract an element of a sequence, given its index. See Chapter 6 [Sequences Arrays Vectors], page 87.

It is impossible to read the same sequence twice, in the sense of `eq` (see Section 2.6 [Equality Predicates], page 36), since sequences are always created anew upon reading. There is one exception: the empty list `()` always stands for the same object, `nil`.

2.3.5 List Type

A *list* is a series of cons cells, linked together. A *cons cell* is an object comprising two pointers named the CAR and the CDR. Each of them can point to any Lisp object, but when the cons cell is part of a list, the CDR points either to another cons cell or to the empty list. See Chapter 5 [Lists], page 67, for functions that work on lists.

The names CAR and CDR have only historical meaning now. 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, any object inside the parentheses is made into an element of the list. That is, a cons cell is made for each element. The CAR of the cons cell points to the element, and its CDR points to the next cons cell which holds the next element in the list. The CDR 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. (The Lisp reader cannot read such an illustration; unlike the textual notation, which can be understood both humans and computers, the box illustrations can only be understood by humans.) The following represents the three-element list `(rose violet buttercup)`:

```
 --- ---      --- ---      --- ---
|___|___|--> |___|___|--> |___|___|--> nil
  |            |            |
  |            |            |
   --> rose     --> violet   --> buttercup
```

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

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

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

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

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

Here 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:

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

2.3.5.1 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, the two notations produce the same result, but list notation is usually clearer and more convenient when it is applicable. When printing a list, the dotted pair notation is only used if the CDR of a cell is not a list.

Box notation can also be used to illustrate what dotted pairs look like. For example, (rose . violet) is diagrammed as follows:

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

Dotted pair notation can be combined with list notation to represent a chain of cons cells with a non-nil final CDR. For example, (rose violet . buttercup) is equivalent to (rose . (violet . buttercup)). The object looks like this:

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

These diagrams make it evident that `(rose . violet . buttercup)` must have an invalid syntax since it would require that a cons cell have three parts rather than two.

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

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

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

2.3.5.2 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 to implement stacks, since new associations may easily be added to or removed from 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 83, for a further explanation of alists and for functions that work on alists.

2.3.6 Array Type

An *array* is composed of an arbitrary number of other Lisp objects, arranged in a contiguous block of memory. Any element of an array may be accessed in constant 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 two types of array, strings and vectors. A string is an array of characters and a vector is an array of arbitrary objects. Both are one-dimensional. (Most other programming languages support multidimensional arrays, but we don't think they are essential in Emacs Lisp.) Each type of array has its own read syntax; see Section 2.3.7 [String Type], page 26, and Section 2.3.8 [Vector Type], page 27.

An array may have any length up to the largest integer; but once created, it has a fixed size. 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 array type is contained in the sequence type and contains both strings and vectors.

2.3.7 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.

The read syntax for strings is a double-quote, an arbitrary number of characters, and another double-quote, `"like this"`. The Lisp reader accepts the same formats for reading the characters of a string as it does for reading single characters (without the question mark that begins a character literal). You can enter a nonprinting character such as tab, `C-a` or `M-C-A` using the convenient escape sequences, like this: `"\t, \C-a, \M-\C-a"`. You can include a double-quote in a string by preceding it with a backslash; thus, `"\""` is a string containing just a single double-quote character. (See Section 2.3.3 [Character Type], page 19, for a description of the read syntax for characters.)

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. This is not the same representation that the meta modifier has in a character regarded as a simple integer. See Section 2.3.3 [Character Type], page 19.

Strings cannot hold characters that have the hyper, super or alt modifiers; they can hold ASCII control characters, but no others. They do not distinguish case in ASCII control characters.

In contrast with the C programming language, Emacs Lisp allows newlines in string literals. 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 in a string literal.

```
"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."
```

The printed representation of a string consists of a double-quote, the characters it contains, and another double-quote. However, any backslash or double-quote characters in the string are preceded with a backslash like this: `"this \" is an embedded quote"`.

A string can hold properties of the text it contains, in addition to the characters themselves. This enables programs that copy text between strings and buffers to preserve the properties with no special effort. See Section 29.17 [Text Properties], page 494. Strings with text properties have a special read and print syntax:

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

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

beg end plist

The elements *beg* and *end* are integers, and together specify a portion of the string; *plist* is the property list for that portion.

See Chapter 4 [Strings and Characters], page 53, for functions that work on strings.

2.3.8 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 92, for functions that work with vectors.

2.3.9 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 may be used in programs as a variable, as a function name, or to hold a list of properties. 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.

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 quotes the single character that follows the backslash, without conversion. 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 type an tab (preceded with a backslash). But you would hardly ever do such a thing.

> **Common Lisp note:** in Common Lisp, lower case letters are always "folded" to upper case, unless they are explicitly escaped. This is in contrast to Emacs Lisp, in which 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 last 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.
```

2.3.10 Lisp 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 150).

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 159). A named function in Lisp is actually a symbol with a valid function in its function cell (see Section 11.4 [Defining Functions], page 155).

Most of the time, functions are called when their names are written in Lisp expressions in Lisp programs. However, a function object found or constructed at run time can be called and passed arguments with the primitive functions `funcall` and `apply`. See Section 11.5 [Calling Functions], page 156.

2.3.11 Lisp 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 parameter-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 165, for an explanation of how to write a macro.

2.3.12 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 8.2.7 [Special Forms], page 110).

It does not matter to the caller of a function whether the function is primitive. However, this does matter if you are trying to substitute a function written in Lisp for a primitive of the same name. The reason is that the primitive function may be called directly from C code. When the redefined function is called from Lisp, the new definition will be used; but calls from C code may still use the old definition.

The term *function* is used to refer to all Emacs functions, whether written in Lisp or C. See Section 2.3.10 [Lisp Function Type], page 28, 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.
```

2.3.13 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 14 [Byte Compilation], page 185, for information about the byte compiler.

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

2.3.14 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 to say that a file of Lisp code should be loaded when necessary to find the true definition of that symbol. 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 13.2 [Autoload], page 177, for more details.

2.4 Editing Types

The types in the previous section are common to many Lisp-like languages. But 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 24 [Buffers], page 383). Most buffers hold the contents of a disk file (see Chapter 22 [Files], page 345) 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 25 [Windows], page 397). But a buffer need not be displayed in a 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, text can be inserted into a buffer very quickly, while "inserting" text into a string is accomplished by concatenation and the result is an entirely new string object.

Each buffer has a designated position called *point* (see Chapter 27 [Positions], page 439). And one buffer is the *current buffer*. Most editing commands act on the contents of the current buffer in the neighborhood of point. Many other functions manipulate or test the characters in the current buffer and much of this manual is devoted to describing these functions (see Chapter 29 [Text], page 463).

Several other data structures are associated with each buffer:

- a local syntax table (see Chapter 31 [Syntax Tables], page 523);
- a local keymap (see Chapter 19 [Keymaps], page 291); and,
- a local variable binding list (see Section 10.9 [Buffer-Local Variables], page 143).

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

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

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

2.4.2 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, 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 2.4.3 [Frame Type], page 32.

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 25 [Windows], page 397, for a description of the functions that work on windows.

2.4.3 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 xemacs@mole.gnu.ai.mit.edu 0xdac80>
```

See Chapter 26 [Frames], page 423, for a description of the functions that work on frames.

2.4.4 Window Configuration Type

A *window configuration* stores information about the positions and sizes of windows at the time the window configuration is created, so that the screen layout may be recreated later.

Window configurations do not have a read syntax. They print as '#<window-configuration>'. See Section 25.16 [Window Configurations], page 420, for a description of several functions related to window configurations.

2.4.5 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. The position value is changed automatically as necessary as text is inserted into or deleted from the buffer. This is 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 28 [Markers], page 453, for information on how to test, create, copy, and move markers.

2.4.6 Process Type

The word *process* 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 Emacs process. External subprocesses, such as shells, GDB, ftp, and compilers, may be used to extend the processing capability of Emacs.

A process takes input from Emacs and returns output to Emacs for further manipulation. Both text and signals can be communicated between Emacs and a subprocess.

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

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

See Chapter 33 [Processes], page 541, for information about functions that create, delete, return information about, send input or signals to, and receive output from processes.

2.4.7 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 17 [Minibuffers], page 231) or output in the echo area (see Section 35.4 [The Echo Area], page 583).

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

See Chapter 16 [Streams], page 221, for a description of various functions related to streams, including various parsing and printing functions.

2.4.8 Keymap Type

A *keymap* maps keys typed by the user to functions. 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 19 [Keymaps], page 291, for information about creating keymaps, handling prefix keys, local as well as global keymaps, and changing key bindings.

2.4.9 Syntax Table Type

A *syntax table* is a vector of 256 integers. Each element of the vector defines how one character is interpreted when it appears in a buffer. For example, in C mode (see Section 20.1 [Major Modes], page 315), the '`+`' character is punctuation, but in Lisp mode it is a valid character in a symbol. These different interpretations are effected by changing the syntax table entry for '`+`', i.e., at index 43.

Syntax tables are only used for scanning text in buffers, not for reading Lisp expressions. The table the Lisp interpreter uses to read expressions is built into the Emacs source code and cannot be changed; thus, to change the list delimiters to be '`{`' and '`}`' instead of '`(`' and '`)`' would be impossible.

See Chapter 31 [Syntax Tables], page 523, for details about syntax classes and how to make and modify syntax tables.

2.4.10 Display Table Type

A *display table* specifies how to display each character code. Each buffer and each window can have its own display table. A display table is actually a vector of length 261. See Section 35.13 [Display Tables], page 596.

2.4.11 Overlay Type

An *overlay* specifies temporary alteration of the display appearance of a part of a buffer. It contains markers delimiting a range of the buffer, plus a property list (a list whose elements are alternating property names and values). Overlays are used to present parts of the buffer temporarily in a different display style.

See Section 35.8 [Overlays], page 588, 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 otherwise, since variables in Lisp are not declared to be of a certain type, as they are in other programming languages. It is therefore up to the individual function to test whether each actual argument belongs to a type that can be used by the function.

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 `+` which it cannot handle:

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

Many functions, called *type predicates*, are provided to test whether an object is a member of a given type. (Following a convention of long standing, the names of most Emacs Lisp predicates end in '`p`'.)

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 69

`arrayp` see Section 6.3 [Array Functions], page 90

`bufferp` see Section 24.1 [Buffer Basics], page 383

`byte-code-function-p`
see Section 2.3.13 [Byte-Code Type], page 30

`case-table-p`
see Section 4.8 [Case Table], page 63

`char-or-string-p`
see Section 4.2 [Predicates for Strings], page 54

`commandp` see Section 18.3 [Interactive Call], page 260

`consp` see Section 5.3 [List-related Predicates], page 69

`floatp` see Section 3.3 [Predicates on Numbers], page 40

`frame-live-p`
see Section 26.3 [Deleting Frames], page 427

`framep` see Chapter 26 [Frames], page 423

`integer-or-marker-p`
see Section 28.2 [Predicates on Markers], page 454

`integerp` see Section 3.3 [Predicates on Numbers], page 40

`keymapp` see Section 19.3 [Creating Keymaps], page 293

`listp` see Section 5.3 [List-related Predicates], page 69

`markerp` see Section 28.2 [Predicates on Markers], page 454 `natnump` see Section 3.3 [Predicates on Numbers], page 40

`nlistp` see Section 5.3 [List-related Predicates], page 69

`numberp` see Section 3.3 [Predicates on Numbers], page 40

`number-or-marker-p`
see Section 28.2 [Predicates on Markers], page 454

`overlayp` see Section 35.8 [Overlays], page 588

`processp` see Chapter 33 [Processes], page 541

`sequencep`
see Section 6.1 [Sequence Functions], page 87

`stringp` see Section 4.2 [Predicates for Strings], page 54

`subrp` see Section 11.8 [Function Cells], page 161

`symbolp` see Chapter 7 [Symbols], page 95

`syntax-table-p`
see Chapter 31 [Syntax Tables], page 523

`user-variable-p`
see Section 10.5 [Defining Variables], page 135

`vectorp` see Section 6.4 [Vectors], page 92

`window-configuration-p`
see Section 25.16 [Window Configurations], page 420

`window-live-p`
see Section 25.3 [Deleting Windows], page 400

`windowp` see Section 25.1 [Basic Windows], page 397

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. See the appropriate chapter describing the data type for these predicates.

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.

(The `make-symbol` function returns an uninterned symbol that is not interned in the standard `obarray`. When uninterned symbols are in use, symbol names are no longer unique. Distinct symbols with the same name are not `eq`. See Section 7.3 [Creating Symbols], page 97.)

```
(eq 'foo 'foo)
     ⇒ t

(eq 456 456)
     ⇒ t

(eq "asdf" "asdf")
     ⇒ nil

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

(eq [(1 2) 3] [(1 2) 3])
     ⇒ nil

(eq (point-marker) (point-marker))
     ⇒ 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.

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

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 an exponential notation as well: thus, 1.5e2 equals 150; in this example, '`e2`' stands for ten to the second power, and is multiplied by 1.5. Floating point values are not exact; they have a fixed, limited amount of precision.

Support for floating point numbers is a new feature in Emacs 19, and it is controlled by a separate compilation option, so you may encounter a site where Emacs does not support them.

3.1 Integer Basics

The range of values for an integer depends on the machine. The range is -8388608 to 8388607 (24 bits; i.e., -2^{23} to $2^{23} - 1$) on most machines, but on others it is -16777216 to 16777215 (25 bits), or -33554432 to 33554431 (26 bits). All of the examples shown below assume an integer has 24 bits.

The Lisp reader reads numbers as a sequence of digits with an optional sign.

```
 1               ; The integer 1.
+1               ; Also the integer 1.
-1               ; The integer -1.
 16777217        ; Also the integer 1, due to overflow.
 0               ; The number 0.
-0               ; The number 0.
 1.              ; The integer 1.
```

To understand how various functions work on integers, especially the bitwise operators (see Section 3.7 [Bitwise Operations], page 46), it is often helpful to view the numbers in their binary form.

In 24 bit binary, the decimal integer 5 looks like this:

```
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
```

-1 is represented as 24 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 1011
```

In this implementation, the largest 24 bit binary integer is the decimal integer 8,388,607. In binary, this number looks like this:

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

Since the arithmetic functions do not check whether integers go outside their range, when you add 1 to 8,388,607, the value is negative integer −8,388,608:

```
(+ 1 8388607)
     ⇒ -8388608
     ⇒ 1000 0000  0000 0000  0000 0000
```

Many of the following functions accept markers for arguments as well as integers. (See Chapter 28 [Markers], page 453.) More precisely, the actual parameters to such functions may be either integers or markers, which is why we often give these parameters the name *int-or-marker*. When the actual parameter is a marker, the position value of the marker is used and the buffer of the marker is ignored.

3.2 Floating Point Basics

Emacs version 19 supports floating point numbers, if compiled with the macro `LISP_FLOAT_TYPE` defined. 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 in question.

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. You can also use a minus sign to write negative floating point numbers, as in '`-1.0`'.

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.

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 28.2 [Predicates on Markers], page 454.

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.

natnump *object* Function

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

Markers are not converted to integers, hence `natnump` of a marker is always `nil`.

People have pointed out that this function is misnamed, because the term "natural number" is usually understood as excluding zero. We are open to suggestions for a better name to use in a future version.

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)` $\equiv$ `(= x 0)`.

3.4 Comparison of Numbers

Floating point numbers in Emacs Lisp actually take up storage, and 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*. If you want to compare just the numeric values, use `=`.

If you use `eq` to compare two integers, it always returns `t` if they have the same value. This is sometimes useful, because `eq` accepts arguments of any type and never causes an error, whereas `=` 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.

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)
  (< (/ (abs (- x y))
        (max (abs x) (abs y)))
     fuzz-factor))
```

> **Common Lisp note:** because of the way numbers are implemented in Common Lisp, you generally need to use '=' to test for equality between numbers of any kind.

`=` *number-or-marker1 number-or-marker2* — Function

This function tests whether its arguments are the same number, and returns `t` if so, `nil` otherwise.

`/=` *number-or-marker1 number-or-marker2* — Function

This function tests whether its arguments are not the same number, and returns `t` if so, `nil` otherwise.

`<` *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)
     ⇒ 2
```

```
(max 1 3 2)
     ⇒ 3
```

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

This function returns the smallest of its arguments.

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. You can call these functions with an integer argument also; if you do, they return it without change.

truncate *number* Function

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

floor *number* Function

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

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.

3.6 Arithmetic Operations

Emacs Lisp provides the traditional four arithmetic operations: addition, subtraction, multiplication, and division. A remainder function supplements the (integer) division function. 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 GNU Emacs Lisp, arithmetic functions do not check for overflow. Thus `(1+ 8388607)` may equal −8388608, 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,

```
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.

abs *number* Function

This returns the absolute value of *number*.

+ &rest *numbers-or-markers* Function

This function adds its arguments together. When given no arguments, `+` returns 0. It does not check for overflow.

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

- &optional *number-or-marker* &rest *other-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, each of the *other-numbers-or-markers* is subtracted from *number-or-marker*, cumulatively. If there are no arguments, the result is 0. This function does not check for overflow.

```
(- 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. It does not check for overflow.

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

/ *dividend divisor* &rest *divisors* Function

This function divides *dividend* by *divisors* and returns the quotient. If there are additional arguments *divisors*, then *dividend* is divided 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 has the same possibility for machine-dependent rounding. As a practical matter, all known machines round in the standard fashion.

If you divide by 0, an `arith-error` error is signaled. (See Section 9.5.3 [Errors], page 122.)

```
(/ 6 2)
     ⇒ 3
(/ 5 2)
     ⇒ 2
(/ 25 3 2)
     ⇒ 4
(/ -17 6)
     ⇒ -2
```

Since the division operator in Emacs Lisp is implemented using the division operator in C, the result of dividing negative numbers may in principle vary from machine to machine, depending on how they round the result. Thus, the result of `(/ -17 6)` could be -3 on some machines. In practice, nearly all machines round the quotient towards 0.

% *dividend divisor* Function

This function returns the value of *dividend* modulo *divisor*; in other words, the integer remainder after division of *dividend* by *divisor*. The sign of the result is the sign of *dividend*. The sign of *divisor* is ignored. The arguments must be integers.

For negative arguments, the value 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 numbers *dividend* and *divisor*,

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

always equals *dividend*.

3.7 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. If *count* is negative, `lsh` shifts zeros into the most-significant bit, producing a positive result even if *integer1* is negative. Contrast this with `ash`, below.

Thus, the decimal number 5 is the binary number 00000101. Shifted once to the left, with a zero put in the one's place, the number becomes 00001010, decimal 10.

Here are two examples of shifting the pattern of bits one place to the left. Since the contents of the rightmost place has been moved one place to the left, a value has to be inserted into the rightmost place. With `lsh`, a zero is placed into the rightmost place. (These examples 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.

Note, however that functions do not check for overflow, and a returned value may be negative (and in any case, no more than a 24 bit value) when an integer is sufficiently left shifted.

For example, left shifting 8,388,607 produces −2:

```
(lsh 8388607 1)          ; left shift
     ⇒ -2
```

In binary, in the 24 bit implementation, the numbers looks like this:

```
;; Decimal 8,388,607
0111 1111  1111 1111  1111 1111
```

which becomes the following when left shifted:

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

Shifting the 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 the pattern of bits 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 the pattern of bits one place to the right divides the value of the binary number by two, rounding downward.

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 a one in the leftmost position, while `lsh` puts a zero in the leftmost position.

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 1010
     ⇒
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)
     ⇒ 8388605

;; Decimal −6
;; becomes decimal 8,388,605.

1111 1111  1111 1111  1111 1010
     ⇒
0111 1111  1111 1111  1111 1101
```

In this case, the 1 in the leftmost position is shifted one place to the right, and a zero is shifted into the leftmost position.

Here are other examples:

```
                    ;         24-bit binary values

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

```
                       ;        24-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 1110
                       ; 13  =  0000 0000  0000 0000  0000 1101
                       ;  4  =  0000 0000  0000 0000  0000 0100
     ⇒ 4               ;  4  =  0000 0000  0000 0000  0000 0100
(logand)
     ⇒ -1              ; -1  =  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.

```
                       ;        24-bit binary values

(logior 12 5)          ; 12  =  0000 0000  0000 0000  0000 1100
                       ;  5  =  0000 0000  0000 0000  0000 0101
     ⇒ 13              ; 13  =  0000 0000  0000 0000  0000 1101
(logior 12 5 7)        ; 12  =  0000 0000  0000 0000  0000 1100
                       ;  5  =  0000 0000  0000 0000  0000 0101
                       ;  7  =  0000 0000  0000 0000  0000 0111
     ⇒ 15              ; 15  =  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. If `logxor` is passed just one argument, it returns that argument.

```
                       ;        24-bit binary values

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

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

```
;;  5  =  0000 0000  0000 0000  0000 0101
;; becomes
;; -6  =  1111 1111  1111 1111  1111 1010

(lognot 5)
     ⇒ -6
```

3.8 Transcendental Functions

These mathematical functions are available if floating point is supported. They 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*.

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.

expt *x y* Function

This function returns *x* raised to power *y*.

sqrt *arg* Function

This returns the square root of *arg*.

3.9 Random Numbers

In a computer, 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 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' process ID number.

random &optional *limit* Function

This function returns a pseudo-random integer. When called more than once, it returns a series of pseudo-random integers.

If *limit* is `nil`, then the value may in principle be any integer. If *limit* is a positive integer, the value is chosen to be nonnegative and less than *limit* (only in Emacs 19).

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, many functions are provided expressly for manipulating them. Emacs Lisp programs use strings more often than individual characters.

See Section 18.5.11 [Strings of Events], page 273, for special considerations when using strings of keyboard character events.

4.1 Introduction to Strings and Characters

Strings in Emacs Lisp are arrays that contain an ordered sequence of characters. Characters are represented in Emacs Lisp as integers; whether an integer was intended as 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 independent of the string contents, and cannot be altered. 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.) This means that any character, including the null character (ASCII code 0), is a valid element of a string.

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

Each character in a string is stored in a single byte. Therefore, numbers not in the range 0 to 255 are truncated when stored into a string. This means that a string takes up much less memory than a vector of the same length.

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 keyboard events 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 others. They do not distinguish case in ASCII control characters. See Section 2.3.3 [Character Type], page 19, for more information about representation of meta and other modifiers for keyboard input characters.

Like a buffer, a string can contain text properties for the characters in it, as well as the characters themselves. See Section 29.17 [Text Properties], page 494.

See Chapter 29 [Text], page 463, for information about functions that display strings or copy them into buffers. See Section 2.3.3 [Character Type], page 19, and Section 2.3.7 [String Type], page 26, for information about the syntax of characters and strings.

4.2 The Predicates for Strings

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

stringp *object* Function

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

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 4.5 [String Conversion], page 58), `make-vector` (see Section 6.4 [Vectors], page 92), and `make-list` (see Section 5.5 [Building Lists], page 71).

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 full string. 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 falls after the last character in the string. Thus:

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

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

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

But we recommend `copy-sequence` for this purpose (see Section 6.1 [Sequence Functions], page 87).

A `wrong-type-argument` error is signaled if either *start* or *end* are non-integers. 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 29.2 [Buffer Contents], page 464), 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. The arguments may be strings, lists of numbers, or vectors of numbers; they are not themselves changed. If no arguments are passed to `concat`, it returns an empty string.

```
(concat "abc" "-def")
     ⇒ "abc-def"
(concat "abc" (list 120 (+ 256 121)) [122])
     ⇒ "abcxyz"
(concat "The " "quick brown " "fox.")
     ⇒ "The quick brown fox."
```

```
(concat)
     ⇒ ""
```

The second example above shows how characters stored in strings are taken modulo 256. In other words, each character in the string is stored in one byte.

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. This special case exists for compatibility with Mocklisp, and we don't recommend you take advantage of it. If you want to convert an integer in this way, use `format` (see Section 4.6 [Formatting Strings], page 59) or `int-to-string` (see Section 4.5 [String Conversion], page 58).

```
(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 158, `vconcat` in Section 6.4 [Vectors], page 92, and `append` in Section 5.5 [Building Lists], page 71.

4.4 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
(char-to-string (+ 256 ?x))
     ⇒ "x"
(char-equal ?x  (+ 256 ?x))
     ⇒ t
```

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
```

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 by their ASCII codes. Keep in mind that lower case letters have higher numeric values in the ASCII character set than their upper case counterparts; numbers and many punctuation characters have a lower numeric value than upper case letters.

```
(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 without any characters in it is the smallest possible 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<`.

See `compare-buffer-substrings` in Section 29.3 [Comparing Text], page 465, for a way to compare text in buffers.

4.5 Conversion of Characters and Strings

Characters and strings may be converted into each other and into integers. `format` and `prin1-to-string` (see Section 16.5 [Output Functions], page 227) may also be used to convert Lisp objects into strings. `read-from-string` (see Section 16.3 [Input Functions], page 224) may be used to "convert" a string representation of a Lisp object into an object.

See Chapter 21 [Documentation], page 335, for a description of functions which return a string representing the Emacs standard notation of the argument character (`single-key-description` and `text-char-description`). These functions are used primarily for printing help messages.

char-to-string *character* Function

This function returns a new string with a length of one character. The value of *character*, modulo 256, is used to initialize the element of the string.

This function is similar to `make-string` with an integer argument of 1. (See Section 4.3 [Creating Strings], page 54.) This conversion can also be done with `format` using the '`%c`' format specification. (See Section 4.6 [Formatting Strings], page 59.)

```
(char-to-string ?x)
     ⇒ "x"
(char-to-string (+ 256 ?x))
     ⇒ "x"
(make-string 1 ?x)
     ⇒ "x"
```

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
int-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.

```
(int-to-string 256)
     ⇒ "256"
(int-to-string -23)
     ⇒ "-23"
(int-to-string -23.5)
     ⇒ "-23.5"
```

See also the function `format` in Section 4.6 [Formatting Strings], page 59.

string-to-number *string* Function
string-to-int *string* Function

This function returns the integer value of the characters in *string*, read as a number in base ten. It skips spaces 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.) If the first character after the ignored whitespace is not a digit or a 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
```

4.6 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 are matched with successive values from *objects*. Thus, the first format specification in *string* is matched with the first such value, the second format specification is matched with 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 will be ignored.

Certain format specifications require values of particular types. However, no error is signaled if the value actually supplied fails to have the expected type. Instead, the output is likely to be meaningless.

Here is a table of the characters that can follow '`%`' to make up a format specification:

'`s`' Replace the specification with the printed representation of the object, made without quoting. 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. 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 #<buffer strings.texi>."

(format "The octal value of 18 is %o,
         and the hex value is %x." 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.

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

(format "%-6d will be padded on the right" 123)
     ⇒ "123    will be 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."
```

4.7 Character Case

The character case functions change the case of single characters or of the contents of strings. The functions convert only alphabetic characters (the letters 'A' through 'Z' and 'a' through 'z'); other characters are not altered. The 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 category in the current syntax table (See Section 31.1.1 [Syntax Class Table], page 524).

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
```

4.8 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 string and character case conversion functions (see the previous section) and those that apply to text in the buffer (see Section 29.16 [Case Changes], page 492). Use case table if you are using a language which has letters that are not the standard ASCII letters.

A case table is a list of this form:

(*downcase upcase canonicalize equivalences*)

where each element is either `nil` or a string of length 256. The element *downcase* says how to map each character to its lower-case equivalent. The element *upcase* maps each character to its upper-case equivalent. If lower and upper case characters are in one-to-one correspondence, use `nil` for *upcase*; then Emacs deduces the upcase table from *downcase*.

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 directions.

The element *canonicalize* maps each character to a canonical equivalent; any two characters that are related by case-conversion have the same canonical equivalent character.

The element *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.)

You can provide `nil` for both *canonicalize* and *equivalences*, in which case both are deduced from *downcase* and *upcase*. Normally, that's what you should do, when you construct a case table. But when you look at the case table that's in use, you will find non-`nil` values for those components.

Each buffer has a case table. Emacs also has a *standard case table* which is copied into each buffer when you create the buffer. (Changing the standard case table doesn't affect any existing buffers.)

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 apply to 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 a string *downcase-table* provided as an argument; this should be a string to be used as the *downcase* part of a case table. They also modify two syntax tables, the standard syntax table and the Text mode syntax table. (See Chapter 31 [Syntax Tables], page 523.)

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

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

set-case-syntax *char syntax downcase-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.

You can load the library '`iso-syntax`' to set up the syntax and case table for the 256 bit ISO Latin 1 character set.

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 which represents an ordered pair. It records two Lisp objects, one labeled as the CAR, and the other labeled as the CDR. (These names are traditional.)

A list is made by chaining cons cells together, one cons cell per element. 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.

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:

```
 ---------------         ---------------
| car   | cdr   |       | car   | cdr   |
| tulip |   o---------->| lily  |  nil  |
|       |       |       |       |       |
 ---------------         ---------------
```

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 is the CAR of the first cons cell, contains the symbol `tulip`. The arrow from the CDR of the first cons cell to the second cons cell indicates that the CDR of the first cons cell points to the second cons cell.

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

```
 --- ---      --- ---
|___|___|--> |___|___|--> nil
  |            |
  |            |
   --> tulip    --> lily
```

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

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

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

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

See Section 2.3.5 [List Type], page 22, for the read and print syntax of 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 tests 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 which 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 116).

```
(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 less than zero, then the first element is returned.

```
(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))
```

nthcdr *n list* Function

This function returns the *n*th cdr of *list*. In other words, it removes the first *n* links of *list* and returns what follows.

If *n* is less than or equal to zero, then all of *list* is returned. If the length of *list* is *n* or less, the value is `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)
```

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 functions.

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 54).

```
(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, strings, or integers. All arguments except the last one are copied, so none of them are altered.

The final argument to `append` may be any object but it is typically a list. The final argument is not copied or converted; it becomes part of the structure of the new list.

Here is an example:

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

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

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

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

This once was the standard way to copy a list, before the function `copy-sequence` was invented. See Chapter 6 [Sequences Arrays Vectors], page 87.

With the help of `apply`, 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
```

In the special case where one of the *sequences* is an integer (not a sequence of integers), it is first converted to a string of digits making up the decimal print representation of the integer. This special case exists for compatibility with Mocklisp, and we don't recommend you take advantage of it. If you want to convert an integer in this way, use `format` (see Section 4.6 [Formatting Strings], page 59) or `number-to-string` (see Section 4.5 [String Conversion], page 58).

```
(setq trees '(pine oak))
     ⇒ (pine oak)
(char-to-string 54)
     ⇒ "6"
(setq longer-list (append trees 6 '(spruce)))
     ⇒ (pine oak 54 spruce)
(setq x-list (append trees 6 6))
     ⇒ (pine oak 54 . 6)
```

See `nconc` in Section 5.6.3 [Rearrangement], page 78, for another way to join lists without copying.

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`.

> **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` and 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. 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 *x1* and *x2*, showing why replacing `b` changes them both:

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

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

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

5.6.2 Altering the CDR of a List

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

setcdr *cons object* Function

> This function stores *object* into the cdr of *cons*. The value returned is *object*, not *cons*.

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 cell:

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

Here is the result in box notation:

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

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

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

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

Here is this result in box notation:

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

5.6.3 Functions that Rearrange Lists

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

nconc &rest *lists* Function

This function returns a list containing all the elements of *lists*. Unlike `append` (see Section 5.5 [Building Lists], page 71), 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 is done 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)
```

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)            ; This function should 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 destructively by reversing the CDRs in the cons cells forming the list. The cons cell which used to be the last one in *list* becomes the first 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 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:

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

sort *list predicate* Function

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

The argument *predicate* must be a function that accepts two arguments. It is called with two elements of *list*. To get an increasing order sort, the *predicate* should return `t` if the first element is "less than" the second, or `nil` if not.

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 sort a list without destroying the original, copy it first with `copy-sequence`.

The CARs of the cons cells are not changed; 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)
```

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 29.13 [Sorting], page 482, for more functions that perform sorting. See `documentation` in Section 21.2 [Accessing Documentation], page 336, for a useful example of `sort`.

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

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 2 '(1 2 3 2 1))
     ⇒ (2 3 2 1)
(memq '(2) '((1) (2)))    ; (2) and (2) are not eq.
     ⇒ nil
```

delq *object list* Function

This function 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 76).

```
(setq sample-list '(1 2 3 (4)))
     ⇒ (1 2 3 (4))
(delq 1 sample-list)
     ⇒ (2 3 (4))
sample-list
     ⇒ (1 2 3 (4))
(delq 2 sample-list)
     ⇒ (1 3 (4))
sample-list
     ⇒ (1 3 (4))
```

Note that `(delq 2 sample-list)` modifies `sample-list` to splice out the second element, but `(delq 1 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)
     ⇒ (1 3 (4))
```

The following two functions are like `memq` and `delq` but use `equal` rather than `eq` to compare elements. They are new in Emacs 19.

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, `memq` 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 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.

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 cell is the *key*, and the CDR is the *associated value*. (This usage of "key" is not related to the term "key sequence"; it means any object which can be looked up in a table.)

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)))
```

Here we regard `red` as the value associated with `rose`. One advantage of this method 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 100, 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 36). 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")
```

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 36.

```
(setq trees '((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 the CDR of the *alist* associations is tested 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 be used to 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 can be seen more clearly if the association is written in dotted pair notation:

```
(lily white) ≡ (lily . (white))
```

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
```

6 Sequences, Arrays, and Vectors

Recall that the *sequence* type is the union of three other Lisp types: lists, vectors, and strings. In other words, any list is a sequence, any vector is a sequence, and any string 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 directly containing all its elements. Therefore, all the elements are accessible in constant time. The length of an existing array cannot be changed. Both strings and vectors are 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. Therefore, 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 elements of vectors and lists may be any Lisp objects. The elements of strings are all characters.

The following diagram shows the relationship between these types:

```
 ___________________________________
|                                   |
|         Sequence                  |
|  ______   ______________________  |
| |      | |                      | |
| | List | |        Array         | |
| |      | |  ________   _______  | |
| |______| | |        | |       | | |
|          | | String | | Vector| | |
|          | |________| |_______| | |
|          |______________________| |
|___________________________________|
```

The Relationship between Sequences, Arrays, and Vectors

6.1 Sequences

In Emacs Lisp, a *sequence* is either a list, a vector or a string. The common property that all sequences have is that each is an ordered collection 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.

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 29.17 [Text Properties], page 494.

See also `append` in Section 5.5 [Building Lists], page 71, `concat` in Section 4.3 [Creating Strings], page 54, and `vconcat` in Section 6.4 [Vectors], page 92, 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)]
```

length *sequence* 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.

```
(length '(1 2 3))
    ⇒ 3
(length ())
    ⇒ 0
```

```
(length "foobar")
    ⇒ 6
(length [1 2 3])
    ⇒ 3
```

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 produce an `args-out-of-range` error.

```
(elt [1 2 3 4] 2)
     ⇒ 3
(elt '(1 2 3 4) 2)
     ⇒ 3
(char-to-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 duplicates `aref` (see Section 6.3 [Array Functions], page 90) and `nth` (see Section 5.4 [List Elements], page 69), except that it works for any kind of sequence.

6.2 Arrays

An *array* object refers directly to 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.

When you create an array, you must specify how many elements it has. The amount of space allocated depends on the number of elements. Therefore, it is impossible to change the size of an array once it is created. You cannot add or remove elements. However, you can replace an element with a different value.

Emacs defines two types of array, both of which are one-dimensional: *strings* and *vectors*. 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 2.3.7 [String Type], page 26, and Section 2.3.8 [Vector Type], page 27.

Both kinds of arrays 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 elements of an array may be referenced or changed with the functions `aref` and `aset`, respectively (see Section 6.3 [Array Functions], page 90).

In principle, if you wish to have an array of 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 characters.
- Strings can hold text properties. See Section 29.17 [Text Properties], page 494.
- 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 53.

6.3 Functions that Operate on Arrays

In this section, we describe the functions that accept both strings and vectors.

arrayp *object* Function

This function returns `t` if *object* is an array (i.e., either a vector or a string).

```
(arrayp [a])
⇒ t
(arrayp "asdf")
⇒ 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
(elt primes 4)
     ⇒ 11
```

```
(aref "abcdefg" 1)
     ⇒ 98               ; 'b' is ASCII code 98.
```

See also the function `elt`, in Section 6.1 [Sequence Functions], page 87.

aset *array index object* Function

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

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

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

If *array* is a string and *object* is not a character, a `wrong-type-argument` error results.

fillarray *array object* Function

This function fills the array *array* with pointers to *object*, replacing any previous values. It returns *array*.

```
(setq a [a b c d e f g])
     ⇒ [a b c d e f g]
(fillarray a 0)
     ⇒ [0 0 0 0 0 0 0]
a
     ⇒ [0 0 0 0 0 0 0]
(setq s "When in the course")
     ⇒ "When in the course"
(fillarray s ?-)
     ⇒ "------------------"
```

If *array* is a string and *object* is not a character, a `wrong-type-argument` error results.

The general sequence functions `copy-sequence` and `length` are often useful for objects known to be arrays. See Section 6.1 [Sequence Functions], page 87.

6.4 Vectors

Arrays in Lisp, like arrays in most languages, are blocks of memory whose elements can be accessed in constant time. A *vector* is a general-purpose array; its elements can be any Lisp objects. (The other kind of array provided in Emacs Lisp is the *string*, whose elements must be characters.) The main uses of vectors in Emacs are as syntax tables (vectors of integers) and 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.

The indices of the elements of a vector are numbered starting with zero in Emacs Lisp.

Vectors are printed with square brackets surrounding the elements in their order. Thus, a vector containing the symbols `a`, `b` and `c` is printed as `[a b c]`. 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. The elements of the vector are not evaluated. See Section 8.2.1 [Self-Evaluating Forms], page 106.

Here are examples of 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
```

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 *integer object* Function

This function returns a new vector consisting of *integer* 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 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)]
```

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. This special case exists for compatibility with Mocklisp, and we don't recommend you take advantage of it. If you want to convert an integer in this way, use `format` (see Section 4.6 [Formatting Strings], page 59) or `int-to-string` (see Section 4.5 [String Conversion], page 58).

For other concatenation functions, see `mapconcat` in Section 11.6 [Mapping Functions], page 158, `concat` in Section 4.3 [Creating Strings], page 54, and `append` in Section 5.5 [Building Lists], page 71.

The `append` function may be used to convert a vector into a list with the same elements (see Section 5.5 [Building Lists], page 71):

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

7 Symbols

A *symbol* is an object with a unique name. This chapter describes symbols, their components, and how they are created and interned. Property lists are also described. The uses of symbols as variables and as function names are described in separate chapters; see Chapter 10 [Variables], page 131, and Chapter 11 [Functions], page 149. For the precise syntax for symbols, see Section 2.3.9 [Symbol Type], page 27.

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 which names the symbol for reading and printing. See `symbol-name` in Section 7.3 [Creating Symbols], page 97.

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 10.6 [Accessing Variables], page 138.

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 161.

Property list
: The *property list cell* holds the property list of the symbol. See `symbol-plist` in Section 7.4 [Property Lists], page 100.

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 is done with a hashing algorithm that uses an obarray; see Section 7.3 [Creating Symbols], page 97.)

In normal usage, the function cell usually contains a function or macro, as that is what the Lisp interpreter expects to see there (see Chapter 8 [Evaluation], page 103). Keyboard macros (see Section 18.13 [Keyboard Macros], page 288), keymaps (see Chapter 19 [Keymaps], page 291) and autoload objects (see Section 8.2.8 [Autoloading], page 111) are also sometimes stored in the function cell 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.

Similarly, the property list cell normally holds a correctly formatted property list (see Section 7.4 [Property Lists], page 100), 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 the value of a cell which 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. 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 documentation about `buffer-file-name` in the '`DOC`' file. (29529 is the offset from the beginning of the '`DOC`' file where the documentation for the function begins.) 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 2.3.12 [Primitive Function Type], page 29). 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 135.

`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.) See Chapter 11 [Functions], page 149.

`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 165.

In GNU 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`' can recognize definitions, and add the appropriate information to tag tables and the '`emacs/etc/DOC-`*version*' file. See Section 21.2 [Accessing Documentation], page 336.

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, then it is used. If no such symbol is found, then a new symbol is created and added to the obarray bucket. Adding a symbol to an obarray is called *interning* it, and the symbol is then called an *interned symbol*. In Emacs Lisp, a symbol may be interned in only one obarray—if you try to intern the same symbol in more than one obarray, you will get unpredictable results.

It is possible for two different symbols to have the same name in different obarrays; these symbols are not `eq` or `equal`. However, this normally happens only as part of abbrev definition (see Chapter 32 [Abbrevs], page 535).

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

If a symbol is not in the obarray, then there is no way for Lisp to find it when its name is read. Such a symbol is called an *uninterned symbol* relative to the obarray. An uninterned symbol has all the other characteristics of symbols.

In Emacs Lisp, an obarray is represented as 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 scan 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 create an obarray that is not empty. This does not work—only `intern` can enter a symbol in an obarray properly. Also, don't try to put into an obarray of your own a symbol that is already interned in the main obarray, because in Emacs Lisp a symbol cannot be in two obarrays at once.

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"
```

Changing the string by substituting characters, etc, 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, a new one is created, added to the obarray, and returned. If *obarray* is supplied, it specifies the obarray to use; otherwise, 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
```

intern-soft *name* &optional *obarray* Function

This function returns the symbol whose name is *name*, or `nil` if a symbol with that name is not found in the obarray. Therefore, you can use `intern-soft` to test whether a symbol with a given name is interned. If *obarray* is supplied, it specifies the obarray to use; otherwise 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 applies *function* to every symbol in *obarray*. It returns `nil`. If *obarray* is not supplied, 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 21.2 [Accessing Documentation], page 336, for another example using `mapatoms`.

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 how to compile it, 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 29.17 [Text Properties], page 494.

The property names and values in a property list can be any Lisp objects, but the names are usually symbols. They are compared 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.

Association lists (see Section 5.8 [Association Lists], page 83) 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 when it is necessary to attach information to various Lisp function names or variables. If all the pairs are recorded in one association list, the program will need to search that entire list each time a function or variable is to be operated on. By contrast, if the information is recorded 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. For this reason, 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 use property names that are probably unique, such as by including the name of the library in the property name.) 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.

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 32 [Abbrevs], page 535).

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 value.

```
(put 'fly 'verb 'transitive)
     ⇒'transitive
(put 'fly 'noun '(a buzzing little bug))
     ⇒ (a buzzing little bug)
(get 'fly 'verb)
     ⇒ transitive
(symbol-plist 'fly)
     ⇒ (verb transitive noun (a buzzing little bug))
```

8 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*. The value is computed in a fashion that depends on the data type of the object, following 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 which 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 16.3 [Input Functions], page 224.

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 18 [Command Loop], page 255.

Evaluation is a recursive process. That is, evaluation of a form may cause `eval` to be called again in order 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`.

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 the form refers to a variable without creating a new binding for it, the value of the binding in the current environment is used. See Chapter 10 [Variables], page 131.

Evaluation of a form may create new environments for recursive evaluation by binding variables (see Section 10.3 [Local Variables], page 132). These environments are temporary and will be gone by the time evaluation

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

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)`.

Finally, evaluation of one particular function call, `byte-code`, invokes the *byte-code interpreter* on its arguments. Although the byte-code interpreter is not the same as the Lisp interpreter, it uses the same environment as the Lisp interpreter, and may on occasion invoke the Lisp interpreter. (See Chapter 14 [Byte Compilation], page 185.)

The details of what evaluation means for each kind of form are described below (see Section 8.2 [Forms], page 106).

8.1 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 13 [Loading], page 175).

eval *form* Function

This is the basic function for performing evaluation. It evaluates *form* in the current environment and returns the result. How the evaluation proceeds depends on the type of the object (see Section 8.2 [Forms], page 106).

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
;; 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-current-buffer &optional *stream* Command

This function evaluates the forms in the current buffer. It reads forms from the buffer and calls `eval` on them until the end of the buffer is reached, or until an error is signaled and not handled.

If *stream* is supplied, the variable `standard-output` is bound to *stream* during the evaluation (see Section 16.5 [Output Functions], page 227).

`eval-current-buffer` always returns `nil`.

eval-region *start end* &optional *stream* 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 supplied, `standard-output` is bound to it for the duration of the command.

`eval-region` always returns `nil`.

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"`). `eval` is called recursively to evaluate the arguments of Lisp function calls and to evaluate bodies of functions.

This limit, with the associated error when it is exceeded, is one way that Lisp avoids infinite recursion on an ill-defined function.

The default value of this variable is 200. If you set it to a value less than 100, Lisp will reset it to 100 if the given value is reached.

`max-specpdl-size` provides another limit on nesting. See Section 10.3 [Local Variables], page 132.

values Variable

The value of this variable is a list of values returned by all expressions which were read from buffers (including the minibuffer), evaluated, and printed. The elements are in order, 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-last
;;   before this example.
(nth 3 values)
     ⇒ 1
```

8.2 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". All three kinds are described in this section, starting with "all other types" which are self-evaluating forms.

8.2.1 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               ; An object, 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 it is inconvenient and not very useful; however, it is possible to put them inside Lisp programs when they are constructed from subexpressions rather than read. Here is an example:

```
;; Build such an expression.
(setq buffer (list 'print (current-buffer)))
     ⇒ (print #<buffer eval.texi>)
;; Evaluate it.
(eval buffer)
     ⊣ #<buffer eval.texi>
     ⇒ #<buffer eval.texi>
```

8.2.2 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 131.

In the following example, we set the value of a symbol with `setq`. When the symbol is later evaluated, that value is returned.

```
(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`. Thus, these two symbols act like self-evaluating forms, even though `eval` treats them like any other symbol.

8.2.3 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 rest of the list consists of *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 including Scheme.

8.2.4 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 a non-symbol is obtained. See Section 11.3 [Function Names], page 154, 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, causing 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 161). 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
```

After that function is called, its body is evaluated; 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))
```

8.2.5 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)
```

When a function call is evaluated, the first step is to evaluate the remaining elements of the list in the order they appear. The results are the actual argument values, one argument from each element. Then the function is called with this list of arguments, effectively using the function `apply` (see Section 11.5 [Calling Functions], page 156). 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 150); then the forms in the function body are evaluated in order, and the result of the last one is used as the value of the function call.

8.2.6 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, which is 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.

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 evaluated 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 165, for a complete description of Emacs Lisp macros.

8.2.7 Special Forms

A *special form* is a primitive function specially marked so that its arguments are not all evaluated. 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.

`let`
`let*` see Section 10.3 [Local Variables], page 132

`or` see Section 9.3 [Combining Conditions], page 116

`prog1`
`prog2`
`progn` see Section 9.1 [Sequencing], page 113

`quote` see Section 8.3 [Quoting], page 112

`save-excursion`
see Section 27.3 [Excursions], page 448

`save-restriction`
see Section 27.4 [Narrowing], page 449

`save-window-excursion`
see Section 25.16 [Window Configurations], page 420

`setq` see Section 10.7 [Setting Variables], page 139

`setq-default`
see Section 10.9.2 [Creating Buffer-Local], page 145

`track-mouse`
see Section 26.11 [Mouse Tracking], page 431

`unwind-protect`
see Section 9.5 [Nonlocal Exits], page 119

`while` see Section 9.4 [Iteration], page 118

`with-output-to-temp-buffer`
see Section 35.7 [Temporary Displays], page 586

> **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).

8.2.8 Autoloading

The *autoload* feature allows you to call a function or macro whose function definition has not yet been loaded into Emacs. When an autoload object appears as a symbol's function definition and that symbol is used as a function, Emacs will automatically install the real definition (plus other associated code) and then call that definition. (See Section 13.2 [Autoload], page 177.)

8.3 Quoting

The special form `quote` returns its single argument "unchanged".

quote *object* Special Form

This special form returns *object*, without evaluating it. This allows symbols and lists, which would normally be evaluated, to be included literally in a program. (It is not necessary to quote numbers, strings, and vectors since they are self-evaluating.)

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 159), which causes an anonymous lambda expression written in Lisp to be compiled, and ` (see Section 12.5 [Backquote], page 168), which is used to quote only part of a list, while computing and substituting other parts.

9 Control Structures

A Lisp program consists of expressions or *forms* (see Section 8.2 [Forms], page 106). 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 control structure 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 they are 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*.

Naturally, Emacs Lisp has many kinds of control structures, including other varieties of sequencing, function calls, conditionals, iteration, and (controlled) jumps. The built-in control structures are special forms since their subforms are not necessarily evaluated. You can use macros to define your own control structure constructs (see Chapter 12 [Macros], page 165).

9.1 Sequencing

Evaluating forms in the order they are written is the most common control structure. Sometimes this happens automatically, such as in a function body. 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`.

When Lisp was young, `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 of an `unwind-protect`, `and`, or `or`.

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 two conditional forms: `if`, which is much the same as in other languages, 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 113.)

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

`if` is a special form because the branch which 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
```

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 *body-forms* are evaluated, 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 the last clause to be executed 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 1) 'foo)
      (t "default"))
⇒ "default"
```

This expression is a `cond` which returns `foo` if the value of `a` is 1, and returns the string `"default"` otherwise.

Both `cond` and `if` can usually be written in terms of the other. Therefore, the choice between them is a matter of taste and 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 `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 the remaining *conditions* are ignored and the `and` returns right away.

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 the remaining *conditions* are ignored and the `or` returns right away. 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) (= 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 expressions 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 119).

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 an empty line is reached.

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 for 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:

```
(catch 'foo
  (progn
    ...
      (throw 'foo t)
    ...))
```

The `throw` 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 `throw` and the `catch` are matched through the first argument: `throw` searches for a `catch` whose first argument is `eq` to the one specified. Thus, in the above example, the `throw` specifies `foo`, and the `catch` specifies the same symbol, so that `catch` is applicable. If there is more than one applicable `catch`, the innermost one takes precedence.

All Lisp constructs between the `catch` and the `throw`, including function calls, are exited automatically along with the `catch`. 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 are exited normally (see Section 10.3 [Local Variables], page 132). Likewise, the buffer and position saved by `save-excursion` (see Section 27.3 [Excursions], page 448) are restored, and so is the narrowing status saved by `save-restriction` and the window selection saved by `save-window-excursion` (see Section 25.16 [Window Configurations], page 420). Any cleanups established with the `unwind-protect` special form are executed if the `unwind-protect` is exited with a `throw`.

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` which throw back to the editor command loop (see Section 18.10 [Recursive Editing], page 285).

> **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. The argument *tag* is evaluated normally before the return point is established.

With the return point in effect, the forms of the *body* are evaluated 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, the `throw` returns the value `yes` from the outer return point. 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 which 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. 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`. See Section 9.5.1 [Catch and Throw], page 119.

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 handled almost like an error. See Section 18.8 [Quitting], page 280.

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

This function signals an error with an error message constructed by applying `format` (see Section 4.5 [String Conversion], page 58) to *format-string* and *args*.

Typical uses of `error` is shown in the following examples:

```
(error "You have committed an error.
        Try something else.")
     error  You have committed 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`.

If you want to use a user-supplied 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 different sorts of errors are classified.

The number and significance of the objects in *data* depends on *error-symbol*. For example, with a `wrong-type-arg` error, there are two objects in the list: a predicate which describes the type that was expected, and the object which failed to fit that type. See Section 9.5.3.4 [Error Names], page 127, for a description of error symbols.

Both *error-symbol* and *data* are available to any error handlers which handle the error: a list `(error-symbol . data)` is constructed to become the value of the local variable bound in the `condition-case` form (see Section 9.5.3.3 [Handling Errors], page 124). If the error is not handled, both of them 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, Emacs searches for an active *handler* for the error. A handler is a specially marked place in the Lisp code of the current function or any of the functions by which it was called. If an applicable handler exists, its code is executed, and control resumes following the handler. The handler executes in the environment of the `condition-case` which established it; all functions called within that `condition-case` have already been exited, and the handler cannot return to them.

If no applicable handler is in effect in your program, 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.

When an error is not handled explicitly, it may cause the Lisp debugger to be called. The debugger is enabled if the variable `debug-on-error` (see Section 15.1.1 [Error Debugging], page 195) 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 subroutines called by this form, and 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 try to handle the same error, the inner of the two will actually handle it.

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 continues by returning 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.

`condition-case` 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 which 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*...). *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* which 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 Names], page 127). Emacs searches all the active `condition-case` forms for a handler which specifies one or more of these names; the innermost matching `condition-case` handles the error. The handlers in this `condition-case` are tested in the order in which they appear.

The body of the handler is then executed, and the `condition-case` returns normally, using the value of the last form in the 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, *var* is bound locally to a list of the form (*error-symbol* . *data*), giving the particulars of the error. 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 *var*.

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

Here is an example of using `condition-case` to handle the error that results from dividing by zero. The handler prints out a warning message and returns a very large number.

```
(defun safe-divide (dividend divisor)
  (condition-case err
      ;; Protected form.
      (/ dividend divisor)
    ;; The handler.
    (arith-error                        ; Condition.
     (princ (format "Arithmetic error: %s" 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: integer-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 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 usable as an error symbol, it must have an `error-conditions` property which gives a list of condition names. This list defines the conditions which 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.

Naturally, Emacs will never signal a `new-error` on its own; only an explicit call to `signal` (see Section 9.5.3 [Errors], page 122) 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. If `signal` were given a list of condition names rather than one error symbol, that would be cumbersome.

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 637, 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 ensure the data are consistent 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.

Only the *body* is actually protected by the `unwind-protect`. If any of the *cleanup-forms* themselves exit nonlocally (e.g., via a `throw` or an error), it is *not* guaranteed that the rest of them will be executed. If the failure of one of the *cleanup-forms* has the potential to cause trouble, then it should be protected by 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 132).

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 in time to kill it.)

Here is an actual example taken from the file '`ftp.el`'. It creates a process (see Chapter 33 [Processes], page 541) to try to establish a connection to a remote machine. As the function `ftp-login` is highly susceptible to numerous problems which 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 for 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 whether the same symbol has a function definition. See Section 7.1 [Symbol Components], page 95.

The textual form of a program is determined by its Lisp object representation; it is the read syntax for the Lisp object which constitutes the program. This is why a variable in a textual Lisp program is written as 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 the first argument of `setq`, the name of the variable, is not evaluated, but the second argument, the desired value, is evaluated normally.

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 another `setq`, the new value replaces the old one:

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

10.2 Variables that Never Change

Emacs Lisp has two special symbols, `nil` and `t`, that always evaluate to themselves. These symbols cannot be rebound, nor can their value cells be changed. An attempt to change the value of `nil` or `t` signals a `setting-constant` error.

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

10.3 Local Variables

Global variables are given values that last until explicitly superseded with new values. Sometimes it is useful to create variable values that exist temporarily—only while within a certain part of the program. 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 which last until the function exits. Similarly, the `let` special form explicitly establishes new local values for specified variables; these last until exit from the `let` form.

When a local value is established, the previous value (or lack of one) of the variable is saved away. 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 10.8.1 [Scope], page 141).

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 is called *dynamic scoping*; see Section 10.8 [Variable Scoping], page 141.) If there are no local bindings, the variable's global binding is its current binding. We also 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 function 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* any of the symbols are bound. Here is an example of this: `Z` is bound to the old value of `Y`, which is 2, not the new value, 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`, except that each symbol in *bindings* is bound as soon as its new value is computed, before the computation of the values of the following local bindings. Therefore, an expression in *bindings* may 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 which create local bindings:

- Function calls (see Chapter 11 [Functions], page 149).
- Macro calls (see Chapter 12 [Macros], page 165).
- `condition-case` (see Section 9.5.3 [Errors], page 122).

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 119) 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.

The default value is 600.

`max-lisp-eval-depth` provides another limit on depth of nesting. See Section 8.1 [Eval], page 104.

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 binding of *symbol* void. This causes any future attempt to use this symbol as a variable to signal the error `void-variable`, unless or 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 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 definition, using `defconst` or `defvar`.

In Emacs Lisp, definitions serve three purposes. First, they inform the user who reads the code that certain symbols are *intended* to be used 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 programs will change the variable. Emacs Lisp does not restrict the ways in which a variable can be used based on `defconst` or `defvar` declarations. However, it also makes a difference for initialization: `defconst` unconditionally initializes the variable, while `defvar` initializes it only if it is void.

One would expect user option variables to be defined with `defconst`, since programs do not change them. Unfortunately, this has bad results if the definition is in a library that is not preloaded: `defconst` would override any prior value when the library is loaded. Users would like to be able to set the option in their init files, and override the default value given in the definition. For this reason, user options must be defined with `defvar`.

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

This special form informs a person reading your code that *symbol* will be used as a variable that the programs are likely to set or change. It is also used for all user option variables except in the preloaded parts of Emacs. Note that *symbol* is not evaluated; the symbol to be defined must appear explicitly in the `defvar`.

If *symbol* already has a value (i.e., it is not void), *value* is not even evaluated, and *symbol*'s value remains unchanged. If *symbol* is void and *value* is specified, it is evaluated and *symbol* is set to the result. (If *value* is not specified, the value of *symbol* is not changed in any case.)

If *symbol* has a buffer-local binding in the current buffer, `defvar` sets the default value, not the local value.

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 21 [Documentation], page 335) look for this property.

If the first character of *doc-string* is '`*`', it means that this variable is considered to be a user option. This affects commands such as `set-variable` and `edit-options`.

For example, this form defines `foo` but does not set its value:

```
(defvar foo)
     ⇒ foo
```

The following example sets 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+ 23)` is not even performed.)

```
(defvar bar (1+ 23)
  "*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))
  (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 informs a person reading your code that *symbol* has a global value, established here, that will not normally be changed or locally bound by the execution of the program. The user, however, may be welcome to change it. Note that *symbol* is not evaluated; the symbol to be defined must appear explicitly in the `defconst`.

`defconst` always evaluates *value* and sets the global value of *symbol* to the result, provided *value* is given. If *symbol* has a buffer-local binding in the current buffer, `defconst` sets the default value, not the local value.

Please note: don't use `defconst` for user option variables in libraries that are not normally loaded. The user should be able to specify a value for such a variable in the '`.emacs`' file, so that it will be in effect if and when the library is loaded later.

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 "Pi to one place.")
     ⇒ pi
(setq pi 4)
     ⇒ pi
pi
     ⇒ 4
```

user-variable-p *variable* Function

This function returns `t` if *variable* is a user option, intended to be set by the user for customization, `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.

Note that if the `defconst` and `defvar` special forms are used while the variable has a local binding, the local binding's value is set, and the global binding is not changed. This would be confusing. But the normal way to use these special forms is at top level in a file, where no local binding should be in effect.

10.6 Accessing Variable Values

The usual way to reference a variable is to write the symbol which names it (see Section 8.2.2 [Symbol Forms], page 107). 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 *symbol* has neither a local binding nor a global value.

10.7 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.

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 be set.

The most-local existing binding of the variable is the binding that is set; shadowed bindings are not affected. If *symbol* is not actually a symbol, a `wrong-type-argument` error is signaled.

```
(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
```

Logically speaking, `set` is a more fundamental primitive that `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 needed only when the choice of variable to be set is made at run time. 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 value, ignoring any lexical bindings. In Emacs Lisp, all variables and all bindings are special, so `set` always affects the most local existing binding.

10.8 Scoping Rules for Variable Bindings

A given symbol `foo` may 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 textually within the function or block that binds the variable.

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

10.8.1 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 in user.
  (list x))
```

In a lexically scoped language, the binding of `x` from `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 call `binder`, then the binding made in `binder` will be seen in `user`:

  ```
  (defun foo (lose)
    (user))
  ```
- If we define `foo` as follows and 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`.

10.8.2 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 work as intended 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
```

10.8.3 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: variable-value pairs. At entry to a function or to a `let` form, we can push bindings on 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. When a new local binding is created, the local value is stored in the value cell, and the old value (belonging to a previous binding) is pushed on a stack. When a binding is eliminated, the old value is popped off the stack and stored in 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.

10.8.4 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.

10.9 Buffer-Local Variables

Global and local variable bindings are found in most programming languages in one form or another. Emacs also supports another, unusual kind of variable binding: *buffer-local* bindings, which apply only to one buffer. Emacs Lisp is meant for programming editing commands, and having different values for a variable in different buffers is an important customization method.

10.9.1 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 the global binding is unchanged; this means that the change is visible in that buffer alone.

A variable may have buffer-local bindings in some buffers but not in others. The global binding is shared by all the buffers that don't have their own bindings. Thus, if you set the variable in a buffer that does not have a buffer-local binding for it, the new value is visible in all buffers except those with buffer-local bindings. (Here we are assuming that there are no `let`-style local bindings to complicate the issue.)

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.

The usual way to make a buffer-local binding is with `make-local-variable`, which is what major mode commands use. This affects just the current buffer; all other buffers (including those yet to be created) continue to share the global value.

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 global value of the variable as usual, but any `setq` creates a buffer-local binding for the current buffer. The new value is stored in the buffer-local binding, leaving the (default) global binding untouched. The global value can no longer be changed with `setq`; you need to use `setq-default` to do that.

Warning: when a variable has 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`. To preserve your sanity, it is wise to avoid such situations. If you use `save-excursion` around each piece of code that changes to a different current buffer, you will not have this problem. Here is an example of incorrect code:

```
(setq foo 'b)
(set-buffer "a")
(make-local-variable 'foo)
```

```
(setq foo 'a)
(let ((foo 'temp))
  (set-buffer "b")
  ...)
foo ⇒ 'a       ; The old buffer-local value from buffer 'a'
               ;   is now the default value.
(set-buffer "a")
foo ⇒ 'temp    ; The local 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")
    ...))
```

Local variables in a file you edit are also represented by buffer-local bindings for the buffer that holds the file within Emacs. See Section 20.1.3 [Auto Major Mode], page 321.

10.9.2 Creating and Destroying 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
```

make-variable-buffer-local *variable* Command

This function marks *variable* (a symbol) automatically buffer-local, so that any attempt to set it will make it local to the current buffer at the time.

The value returned is *variable*.

buffer-local-variables &optional *buffer* Function

This function tells you what the buffer-local variables are in buffer *buffer*. It returns an association list (see Section 5.8 [Association Lists], page 83) in which each association contains one buffer-local variable and its value. When a buffer-local variable is void in *buffer*, then it appears directly in the resulting list. If *buffer* is omitted, the current buffer is used.

```
(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 local variables.
    ;; This one is local and void:
    foobar
    ;; This one is local and nonvoid:
    (bind-me . 69))
```

Note that storing new values into the CDRs of cons cells in this list does *not* change the 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 global (default) binding of *variable* becomes visible in this buffer. Usually this results in a change in the value of *variable*, since the global value is usually different from the buffer-local value just eliminated.

It is possible to kill the local binding of a variable that automatically becomes local when set. This causes the variable to show its global value in the current buffer. However, if you set the variable again, this will once again create a local value.

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

kill-all-local-variables Function

This function eliminates all the buffer-local variable bindings of the current buffer except for variables marker 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: its local keymap is set to `nil`, its syntax table is set to the value of `standard-syntax-table`, and its abbrev table is set to the value of `fundamental-mode-abbrev-table`.

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`.

A 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.

10.9.3 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 except when specifically overridden.

The functions `default-value` and `setq-default` allow you to access and change the 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 which 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 local value.

default-value *symbol* Function

This function returns *symbol*'s default value. This is the value that is seen in buffers that do not have their own values for this variable. If *symbol* is not buffer-local, this is equivalent to `symbol-value` (see Section 10.6 [Accessing Variables], page 138).

default-boundp *variable* Function

The function `default-boundp` tells you whether *variable*'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 value* Special Form

This sets the default value of *symbol* to *value*. *symbol* is not evaluated, but *value* is. The value of the `setq-default` form is *value*.

If a *symbol* is not buffer-local for the current buffer, and is not marked automatically buffer-local, this 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 'local)
     ⇒ local
(setq local 'value-in-foo)
     ⇒ value-in-foo
(setq-default local 'new-default)
     ⇒ new-default
local
     ⇒ value-in-foo
(default-value 'local)
     ⇒ new-default

;; In (the new) buffer ‘bar’:
local
     ⇒ new-default
(default-value 'local)
     ⇒ new-default
(setq local 'another-default)
     ⇒ another-default
(default-value 'local)
     ⇒ another-default

;; Back in buffer ‘foo’:
local
     ⇒ value-in-foo
(default-value 'local)
     ⇒ another-default
```

set-default *symbol value* Function

This function is like `setq-default`, except that *symbol* is evaluated.

```
(set-default (car '(a b c)) 23)
     ⇒ 23
(default-value 'a)
     ⇒ 23
```

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 that a function is a primitives is because it is fundamental, or 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.4 [Writing Emacs Primitives], page 628.

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 113.

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. See Chapter 12 [Macros], page 165, 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 18.2 [Defining Commands], page 256). 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.

Strings 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 18.1 [Command Overview], page 255.

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 programmers, 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 2.3.13 [Byte-Code Type], page 30.

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* for historical reasons, even though it is not really an expression at all—it is not a form that can be evaluated meaningfully.)

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 argument variable names (symbols). 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 132.

The documentation string is an actual string that serves to describe the function for the Emacs help facilities. See Section 11.2.4 [Function Documentation], page 153.

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 18.2 [Defining Commands], page 256, 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)
```

The body of this lambda expression is evaluated 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))
```

Here all the arguments `1`, `(* 2 3)`, and `(- 5 4)` are evaluated, left to right. Then the lambda expression is applied 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 132). 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 159).

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 Advanced 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 end of the string if you omit it. It is also convenient for certain functions to accept an indefinite number of arguments, as the functions `and` 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 more actual arguments than these unless `&rest` exists. 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`. 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 means to use the length of the string supplied. There is no way for the function to distinguish between an explicit argument of `nil` and an omitted argument.

> **Common Lisp note:** Common Lisp allows the function to specify what default value to use when an optional argument is omitted; GNU Emacs Lisp always uses `nil`.

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. If three actual arguments are given; then 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 21 [Documentation], page 335, for how the *documentation-string* is accessed.

It is a good idea to provide documentation strings for all commands, and for all other functions in your program that users of your program should know about; internal functions might as well have only comments, since comments don't take up any room when your program is loaded.

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, 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. *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`, 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 95). 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*. When the evaluator finds the function definition to use in place of the symbol, we call that *symbol function indirection*; see Section 8.2.4 [Function Indirection], page 108.

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 more convenient to refer to them by their names in other functions. 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).

Functions are often identified 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.

11.4 Defining Named 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*`)`

This lambda expression is stored in the function cell of *name*. The value returned by evaluating the `defun` form is *name*, but usually we ignore this value.

As described previously (see Section 11.2 [Lambda Expressions], page 150), *argument-list* is a list of argument names and may include the keywords `&optional` and `&rest`. Also, the first two forms in *body-forms* may be a documentation string and an interactive declaration.

Note that the same symbol *name* may also be used as a global variable, since the value cell is independent of the function cell.

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.

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. This process 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`. See Chapter 8 [Evaluation], page 103, for a description of evaluation.

When you write a list as an expression in your program, the function name is part of the program. This means that the choice of which function to call is made when you write the program. Usually that's just what you want. Occasionally you need to decide at run time which function to call. Then you can use the functions `funcall` and `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 this example with that 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 arguments to give to *function*, rather than a single argument. We also say that this list is *appended* to the other arguments.

`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)
```

An interesting example of using `apply` is found in the description of `mapcar`; see the following section.

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 such a function, 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 three such functions; `mapcar` and `mapconcat`, which scan a list, are described here. For the third mapping function, `mapatoms`, see Section 7.3 [Creating Symbols], page 97.

mapcar *function sequence* Function

`mapcar` applies *function* to each element of *sequence* in turn. The results are made into a `nil`-terminated list.

The argument *sequence* may be a list, a 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* (f &rest args)
  "Apply FUNCTION to successive cars of all ARGS, until one
ends.  Return the list of results."
  ;; If no list is exhausted,
  (if (not (memq 'nil args))
      ;; Apply function to CARs.
      (cons (apply f (mapcar 'car args))
            (apply 'mapcar* f
                   ;; 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 returns 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` (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 pass an anonymous function that multiplies a number by two:

```
(defun double-each (list)
  (mapcar '(lambda (x) (* 2 x)) list))
     ⇒ double-each
(double-each '(2 11))
     ⇒ (4 22)
```

In such cases, we usually use the special form `function` instead of simple quotation to quote the anonymous function.

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. See Section 8.3 [Quoting], page 112, for comparison.

Using `function` instead of `quote` makes a difference inside a function or macro that you are going to compile. For example:

```
(defun double-each (list)
  (mapcar (function (lambda (x) (* 2 x))) list))
     ⇒ double-each
(double-each '(2 11))
     ⇒ (4 22)
```

If this definition of `double-each` is compiled, the anonymous function is compiled as well. By contrast, in the previous definition where ordinary `quote` is used, the argument passed to `mapcar` is the precise list shown:

```
(lambda (arg) (+ arg 5))
```

The Lisp compiler cannot assume this list is a function, even though it looks like one, since it does not know what `mapcar` does with the list. Perhaps `mapcar` will check that the CAR of the third element is the symbol `+`! The advantage of `function` is that it tells the compiler 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*

See `documentation` in Section 21.2 [Accessing Documentation], page 336, 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.

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`); but `nil` or `void` is *an object*. 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

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 10.3 [Local Variables], page 132.)

```
(defun foo (x) x)
     ⇒ x
(fmakunbound 'foo)
     ⇒ x
(foo 1)
error  Symbol's function definition is void: foo
```

fset *symbol object* Function

This function stores *object* in the function cell of *symbol*. The result is *object*. Normally *object* should be a function or the name of a function, but this is not checked.

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.)
- Giving a symbol a function definition that is not a list and therefore cannot be made with `defun`. See Section 8.2.3 [Classifying Lists], page 107, for an example of this usage.
- 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 the first two uses:

```
;; Give first the same definition car has.
(fset 'first (symbol-function 'car))
     ⇒ #<subr car>
(first '(1 2 3))
     ⇒ 1

;; Make the symbol car the function definition of xfirst.
(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"
```

When writing a function that extends a previously defined function, the following idiom is often 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`.

See also the function `indirect-function` in Section 8.2.4 [Function Indirection], page 108.

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.

Another disadvantage is that making a large function inline can increase the size of compiled code both in files and in memory. Since the advantages of inline functions are 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. But the macro would have a limitation: you can use it only explicitly—a macro cannot be called with `apply`, `mapcar` and so on. Also, it takes some work to convert an ordinary function into a macro. (See Chapter 12 [Macros], page 165.) To convert it into an inline function is very easy; simply replace `defun` with `defsubst`.

Inline functions can be used and open coded later on in the same file, following the definition, just like macros.

Emacs versions prior to 19 did not have inline functions.

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 156.

`autoload` See Section 13.2 [Autoload], page 177.

`call-interactively`
See Section 18.3 [Interactive Call], page 260.

`commandp` See Section 18.3 [Interactive Call], page 260.

`documentation`
See Section 21.2 [Accessing Documentation], page 336.

`eval` See Section 8.1 [Eval], page 104.

`funcall` See Section 11.5 [Calling Functions], page 156.

`ignore` See Section 11.5 [Calling Functions], page 156.

`indirect-function`
See Section 8.2.4 [Function Indirection], page 108.

`interactive`
See Section 18.2.1 [Using Interactive], page 256.

`interactive-p`
See Section 18.3 [Interactive Call], page 260.

`mapatoms` See Section 7.3 [Creating Symbols], page 97.

`mapcar` See Section 11.6 [Mapping Functions], page 158.

`mapconcat`
See Section 11.6 [Mapping Functions], page 158.

`undefined`
See Section 19.8 [Key Lookup], page 302.

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 163.

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` has the value `x`—*not* the *value* of `x`. 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. This is used by byte compilation.

```
(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 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 use this feature, you must define the macro in the same file where it is used and before its first use.

While byte-compiling a file, any `require` calls at top-level are executed. One way to ensure that necessary macro definitions are available during compilation is to require the file that defines them. See Section 13.4 [Features], page 180.

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 mapping functions 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*`)`

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 152). Macros may have a documentation string, but any `interactive` declaration is ignored since macros cannot be called interactively.

12.5 Backquote

It could prove rather awkward to write macros of significant size, simply due to the number of times the function `list` needs to be called. To make writing these forms easier, a macro '`' (often called *backquote*) exists.

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 8.3 [Quoting], page 112). For example, these two forms yield identical results:

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

By inserting a special marker, '`,`', inside of the argument to backquote, it is possible to evaluate desired portions of the argument:

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

It is also possible to have an evaluated list *spliced* into the resulting list by 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 (,@ nil)))
     ⇒ (use the words foo bar as elements)
```

The reason for `(,@ nil)` is to avoid a bug in Emacs version 18. The bug occurs when a call to `,@` is followed only by constant elements. Thus,

```
(` (use the words (,@ (cdr list)) as elements))
```

would not work, though it really ought to. `(,@ nil)` avoids the problem by being a nonconstant element that does not affect the result.

` *list* — Macro

This macro returns *list* as `quote` would, except that the list is copied each time this expression is evaluated, and any sublist of the form `(, `*subexp*`)` is replaced by the value of *subexp*. Any sublist of the form `(,@ `*listexp*`)` is replaced by evaluating *listexp* and splicing its elements into the containing list in place of this sublist. (A single sublist can in this way be replaced by any number of new elements in the containing list.)

There are certain contexts in which '`,`' would not be recognized and should not be used:

```
;; Use of a `,' expression as the cdr of a list.
(` (a . (, 1)))                         ; Not (a . 1)
     ⇒ (a \, 1)

;; Use of `,' in a vector.
(` [a (, 1) c])                         ; Not [a 1 c]
     error  Wrong type argument

;; Use of a `,' as the entire argument of ``'.
(` (, 2))                               ; Not 2
     ⇒ (\, 2)
```

Common Lisp note: in Common Lisp, '`,`' and '`,@`' are implemented as reader macros, so they do not require parentheses. Emacs Lisp implements them as functions because reader macros are not supported (to save space).

12.6 Common Problems Using Macros

The basic facts of macro expansion have all been described above, but there consequences are often counterintuitive. This section describes some important consequences that can lead to trouble, and rules to follow to avoid trouble.

12.6.1 Evaluating Macro Arguments Too Many Times

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, e.g.,
    (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.)

This macro suffers 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 introduces another problem.

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 97). 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 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` which 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 you evaluate any of the macro argument expressions during the computation of the expansion, such as by calling `eval` (see Section 8.1 [Eval], page 104). 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 'b)
(foo a) ↦ (setq 'b t)      ; invalid!
error  Symbol's value is void: b
```

It makes a difference whether the user types `a` or `x`, because `a` conflicts with the macro argument variable `a`.

In general it is best to avoid calling `eval` in a macro definition at all.

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.

In particular, constructing objects is a kind of side effect. If the macro is called once, then the objects are constructed only once. In other words, the same structure of objects is used each time the macro call is executed. In interpreted operation, the macro is reexpanded each time, producing a fresh collection of objects each time. Usually this does not matter—the objects have the same contents whether they are shared or not. But if the surrounding program does side effects on the objects, it makes a difference whether they are shared. Here is an example:

```
(defmacro new-object ()
  (list 'quote (cons nil nil)))

(defun initialize (condition)
  (let ((object (new-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 `new-object` is expanded during compilation, producing a single "constant" `(nil)` that is reused and altered each time `initialize` is called.

13 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, optionally, 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.

13.1 How Programs Do Loading

There are several interface functions for loading. For example, the `autoload` function creates a Lisp object that loads a file when it is evaluated (see Section 13.2 [Autoload], page 177). `require` also causes files to be loaded (see Section 13.4 [Features], page 180). Ultimately, all these facilities call the `load` function to do the work.

load *filename* &optional *missing-ok nomessage nosuffix* — 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 has '`.elc`' appended. If such a file exists, it is loaded. But if there is no file by that name, then `load` looks for a file whose name has '`.el`' appended. If that file exists, it is loaded. Finally, if there is no file by either name, `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.

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 it is represented as `nil`. All three possible suffixes are tried in the first directory in `load-path`, then all three in the second directory in `load-path`, etc.

If you get a warning that '`foo.elc`' is older than '`foo.el`', it means you should consider recompiling '`foo.el`'. See Chapter 14 [Byte Compilation], page 185.

Messages like '`Loading foo...`' and '`Loading foo...done`' appear in the echo area during loading unless *nomessage* is non-`nil`.

Any errors that are encountered while loading a file cause `load` to abort. If the load was done for the sake of `autoload`, certain kinds of top-level forms, those which define functions, are undone.

The error `file-error` is signaled (with '`Cannot open load file` *filename*') if no file is found. No error is signaled if *missing-ok* is non-`nil`—then `load` just returns `nil`.

`load` returns `t` if the file loads successfully.

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 it exists; otherwise it is set to the default specified in '`emacs/src/paths.h`' when Emacs is built.

The syntax of `EMACSLOADPATH` is the same as that of `PATH`; fields are separated by '`:`', 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/lib/emacs/lisp
```

Here is how to set it using `sh`:

```
export EMACSLOADPATH
EMACSLOADPATH=.:/user/bil/emacs:/usr/local/lib/emacs/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")
       load-path))
```

In this example, the path searches the current working directory first, followed then by the '/user/bil/emacs' directory and then by the '/usr/local/lisplib' directory, which are then followed by the standard directories for Lisp code.

When Emacs version 18 processes command options '-l' or '-load' which specify Lisp libraries to be loaded, it temporarily adds the current directory to the front of `load-path` so that files in the current directory can be specified easily. Newer Emacs versions also find such files in the current directory, but without altering `load-path`.

load-in-progress Variable

This variable is non-`nil` if Emacs is in the process of loading a file, and it is `nil` otherwise. This is how `defun` and `provide` determine whether a load is in progress, so that their effect can be undone if the load fails.

To learn how `load` is used to build Emacs, see Section B.1 [Building Emacs], page 623.

13.2 Autoload

The *autoload* facility allows you to make a function or macro available but put off loading its actual definition. An attempt to call a symbol whose definition is an autoload object automatically reads the file to install the real definition and its other associated code, and then calls the real definition.

To prepare a function or macro for autoloading, you must call `autoload`, specifying the function name and the name of the file to be loaded. A file such as 'emacs/lisp/loaddefs.el' usually does this when Emacs is first built.

The following example shows how `doctor` is prepared for autoloading 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 cause the documentation string to be put in the '`etc/DOC`' file. (See Section B.1 [Building Emacs], page 623.) In any other source file, you would write just this:

```
(autoload 'doctor "doctor"
  "Switch to *doctor* buffer and start giving psychotherapy."
  t)
```

Calling `autoload` creates an autoload object containing the name of the file and some other information, and makes this the function definition of the specified symbol. When you later try to call that symbol as a function or macro, the file is loaded; the loading should redefine that symbol with its proper definition. After the file completes loading, the function or macro is called as if it had been there originally.

If, at the end of loading the file, the desired Lisp function or macro has not been defined, then the error `error` is signaled (with data `"Autoloading failed to define function` *function-name*`"`).

The autoloaded file may, of course, contain 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 the 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 appear defined, but they may fail to work properly for the lack of certain subroutines defined later in the file and not loaded successfully.

Emacs as distributed comes with many autoloaded functions. The calls to `autoload` are in the file '`loaddefs.el`'. There is a convenient way of updating them automatically.

Write '`;;;###autoload`' on a line by itself before the real definition of the function, in its autoloadable source file; then the command `M-x update-file-autoloads` automatically puts the `autoload` call into '`loaddefs.el`'. `M-x update-directory-autoloads` is more powerful; it updates autoloads for all files in the current directory.

You can also put other kinds of forms into '`loaddefs.el`', by writing '`;;;###autoload`' followed on the same line by the form. `M-x update-file-autoloads` copies the form from that line.

The commands for updating autoloads work by visiting and editing the file '`loaddefs.el`'. To make the result take effect, you must save that file's buffer.

autoload *symbol filename* &optional *docstring interactive type* — Function

This function defines the function (or macro) named *symbol* so as to load automatically from *filename*. The string *filename* is a file name which will be passed to `load` when the function is called.

The argument *docstring* is the documentation string for the function. Normally, this is the same string that is in the function definition itself. This makes it possible to look at the documentation without loading the real definition.

If *interactive* is non-`nil`, then the function can be called interactively. This lets completion in `M-x` work without loading the function's real definition. The complete interactive specification need not be given here. If *type* is `macro`, then the function is really a macro. If *type* is `keymap`, then the function is really a keymap.

If *symbol* already has a non-`nil` function definition that is not an autoload object, `autoload` does nothing and returns `nil`. If the function cell of *symbol* is void, or is already an autoload object, then it is set to an autoload object that looks 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`' file (see Section 21.1 [Documentation Basics], page 335), `t` means the function is interactive, and `nil` that it is not a macro.

13.3 Repeated Loading

You may load a 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 in which it is located.

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, remember to byte-compile it if necessary; otherwise you may find yourself inadvertently reloading the older, byte-compiled file instead of your newer, non-compiled file!

When writing the forms in a library, keep in mind that the library might be loaded more than once. For example, the choice of `defvar` vs. `defconst` for defining a variable depends on whether it is desirable to reinitialize the variable if the library is reloaded: `defconst` does so, and `defvar` does not. (See Section 10.5 [Defining Variables], page 135.)

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)))
```

Occasionally you will want to test explicitly whether a library has already been loaded; you can do so as follows:

```
(if (not (boundp 'foo-was-loaded))
    execute-first-time-only)

(setq foo-was-loaded t)
```

13.4 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.

The use of named features simplifies the task of determining whether required definitions have been defined. A feature name is a symbol that stands for a collection of functions, variables, etc. A program that needs the collection may ensure that they are defined by *requiring* the feature. If the file that contains the feature has not yet been loaded, then it will be loaded (or an error will be signaled if it cannot be loaded). The file thus loaded must *provide* the required feature or an error will be signaled.

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`.

Features are normally named after the files they are provided in so that `require` need not be given the file name.

For example, in '`emacs/lisp/prolog.el`', the definition for `run-prolog` includes the following code:

```
(defun run-prolog ()
  "Run an inferior Prolog process,\
 input and output via buffer *prolog*."
  (interactive)
  (require 'comint)
  (switch-to-buffer (make-comint "prolog" prolog-program-name))
  (inferior-prolog-mode))
```

The expression `(require 'shell)` loads the file '`shell.el`' if it has not yet been loaded. This ensures that `make-shell` is defined.

The '`shell.el`' file contains the following top-level expression:

```
(provide 'shell)
```

This adds `shell` to the global `features` list when the '`shell`' file is loaded, so that `(require 'shell)` will henceforth know that nothing needs to be done.

When `require` is used at top-level in a file, it takes effect if you byte-compile that file (see Chapter 14 [Byte Compilation], page 185). 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.
```

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)
```

During autoloading, if the file is not completely loaded (due to an error in the evaluation of the contents) any function definitions or `provide` calls that occurred during the load are undone. See Section 13.2 [Autoload], page 177.

require *feature* &optional *filename* Function

This function checks whether *feature* is present in the current Emacs session (using `(featurep` *feature*`)`; see below). If it is not, then `require` loads *filename* with `load`. If *filename* is not supplied, then the name of the symbol *feature* is used as the file name to load.

If *feature* is not provided after the file has been loaded, Emacs will signal the error `error` (with data '`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., *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.

13.5 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* Command

This command unloads the library that provided feature *feature*. It undefines all functions and variables defined with `defvar`, `defmacro`, `defconst`, `defsubst` and `defalias` by the library which provided feature *feature*. It then restores any autoloads associated with those symbols.

The `unload-feature` function is written in Lisp; its actions are based on the variable `load-history`.

load-history feature association list 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, which were defined as functions or variables.
- Lists of the form `(require .` *feature*`)` indicating the features that are required.

- Lists of the form `(provide . feature)` indicating the features that are 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.

13.6 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.

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*.

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`.

14 Byte Compilation

GNU Emacs Lisp has a *compiler* that translates functions written in Lisp into a special representation called *byte-code* that can be executed more efficiently. The compiler replaces Lisp function definitions with byte-code. When a byte-code function is called, its definition is evaluated by the *byte-code interpreter*.

Because the byte-compiled code is evaluated by the byte-code interpreter, instead of being executed directly by the machine's hardware (as true compiled code is), byte-code is completely transportable from machine to machine without recompilation. It is not, however, as fast as true compiled code.

In general, any version of Emacs can run byte-compiled code produced by recent earlier versions of Emacs, but the reverse is not true. In particular, if you compile a program with Emacs 18, you can run the compiled code in Emacs 19, but not vice versa.

See Section 15.3 [Compilation Errors], page 205, for how to investigate errors occurring in byte compilation.

14.1 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`.

When you run the byte compiler, you may get warnings in a buffer called '`*Compile-Log*`'. These report usage in your program that suggest a problem, but are not necessarily erroneous.

Be careful when byte-compiling code that uses macros. 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 167.

While byte-compiling a file, any `require` calls at top-level are executed. One way to ensure that necessary macro definitions are available during compilation is to require the file that defines them. See Section 13.4 [Features], page 180.

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. For a rough comparison, consider the example below:

```
(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)
⇒ ("Thu Jan 12 20:18:38 1989"
    "Thu Jan 12 20:19:29 1989")  ; 51 seconds

(byte-compile 'silly-loop)
⇒ [Compiled code not shown]

(silly-loop 100000)
⇒ ("Thu Jan 12 20:21:04 1989"
    "Thu Jan 12 20:21:17 1989")  ; 13 seconds
```

In this example, the interpreted code required 51 seconds to run, whereas the byte-compiled code required 13 seconds. These results are representative, but actual results will vary greatly.

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` does not compile macros. `byte-compile` returns the new, compiled definition of *symbol*.

```
(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 compiled function object. The string it contains is the actual byte-code; each character in it is 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 appending '`c`' 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.

If a '`.el`' file exists, but there is no corresponding '`.elc`' file, then *flag* is examined. If it is `nil`, the file is ignored. If it is non-`nil`, the user is asked whether the file should be compiled.

The returned value of this command is unpredictable.

batch-byte-compile Function

This function runs `byte-compile-file` on the files remaining 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. (The file which gets the error will not, of course, produce any compiled 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 newer Emacs versions (19 and up), byte-code is usually executed as part of a compiled function object, and only rarely as part of a call to `byte-code`.

14.2 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`. Using `require` is preferable if there is a substantial amount of code to be executed in this way.

eval-when-compile *body* Special Form

This form marks *body* to be evaluated at compile time *only*. The result of evaluation by the compiler becomes a constant which appears in the compiled program. When the program is interpreted, not compiled at all, *body* is evaluated normally.

At top-level, this is analogous to the Common Lisp idiom `(eval-when (compile) ...)`. Elsewhere, the Common Lisp '`#.`' reader macro (but not when interpreting) is closer to what `eval-when-compile` does.

14.3 Byte-Code 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 '`[`'.

In Emacs version 18, there was no byte-code function object data type; compiled functions used the function `byte-code` to run the byte code.

A byte-code function object must have at least four elements; there is no maximum number, but only the first six elements are actually used. They are:

arglist The list of argument symbols.

byte-code The string containing the byte-code instructions.

constants The vector of constants referenced by the byte code.

stacksize The maximum stack size this function needs.

docstring The documentation string (if any); otherwise, `nil`. For functions preloaded before Emacs is dumped, this is usually an integer which is an index into the 'DOC' file; use `documentation` to convert this into a string (see Section 21.2 [Accessing Documentation], page 336).

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, we hope, always makes the elements consistent.

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.

14.4 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. Values get stored by being pushed onto the stack, and are popped off and manipulated, the results being pushed back onto the stack. When a function returns, the top of the stack is popped and returned as the value of the function.

In addition to the stack, values used during byte-code execution can be stored in ordinary Lisp variables. Variable values can be pushed onto the stack, and variables can be set by popping 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 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
                            ;   else pop top of stack
                            ;   and continue
                            ;   (this exits the while loop).

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.

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
```

15 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 the Lisp debugger to investigate what is happening during execution.
- 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 a 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 34.7 [Terminal Input], page 572.

For debugging problems in terminal descriptions, the `open-termscript` function can be useful. See Section 34.8 [Terminal Output], page 576.

15.1 The Lisp Debugger

The *Lisp debugger* provides you with 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 18.10 [Recursive Editing], page 285.

15.1.1 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 get Lisp errors when invoked in inappropriate contexts (such as `C-f` at the end of the buffer) and during ordinary editing it would be very unpleasant to enter the debugger each time this happens. If you want errors to enter the debugger, set the variable `debug-on-error` to non-`nil`.

debug-on-error User Option

This variable determines whether the debugger is called when a error is signaled and not handled. If `debug-on-error` is `t`, all errors call the debugger. 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.

15.1.2 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`. Contrariwise, `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 18.8 [Quitting], page 280.

15.1.3 Entering the Debugger on a Function Call

To investigate a problem that happens in the middle of a program, one useful technique is to cause the debugger to be entered when 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. Even functions that are commands may be debugged—they will enter the debugger when called inside a function, or when called interactively (after the reading of the arguments). Primitive functions (i.e., those written in C) may not be debugged.

When `debug-on-entry` is called interactively, it prompts for *function-name* in the minibuffer.

Caveat: if `debug-on-entry` is called more than once on the same function, the second call does nothing. If you redefine a function after using `debug-on-entry` on it, the code to enter the debugger is lost.

`debug-on-entry` returns *function-name*.

```
(defun fact (n)
  (if (zerop n) 1
      (* n (fact (1- n)))))
     ⇒ fact
(debug-on-entry 'fact)
     ⇒ fact
(fact 3)
     ⇒ 6

------ 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 `cancel-debug-on-entry` is called more than once on the same function, the second call does nothing. `cancel-debug-on-entry` returns *function-name*.

15.1.4 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`. 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 isn't ignored, it will alter the execution of the program!) Usually this means inside a `progn` or an implicit `progn` (see Section 9.1 [Sequencing], page 113).

15.1.5 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 18.10 [Recursive Editing], page 285) 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 contents of the backtrace buffer show you the functions that are executing and the arguments that were given to them. 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 should always 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.

15.1.6 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 file for the function and type `C-M-x` on its definition.)

`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 the effect of any variables or data structures you may have changed while inside the debugger).

  Continuing is possible after entry to the debugger due to function entry or exit, explicit invocation, quitting or certain errors. Most errors cannot be continued; trying to continue an unsuitable error causes the same error to occur again.

`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.

`e`
: Read a Lisp expression in the minibuffer, evaluate it, and print the value in the echo area. This is the same as the command `M-ESC`, except that `e` is not normally disabled like `M-ESC`.

`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 makes a difference when the debugger was invoked due to exit from a Lisp call frame (as requested with `b`);

then the value specified in the `r` command is used as the value of that frame.

You can't use `r` when the debugger was entered due to an error.

15.1.7 Invoking the Debugger

Here we describe fully the function 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, leaving that buffer in Debugger mode and displayed in the selected window.

Debugger mode provides a `c` command which operates by exiting the recursive edit, switching back to the previous buffer, and returning to whatever called `debug`. The `r` command also returns from `debug`. These are the only ways the function `debug` can return to its caller.

If the first of the *debugger-args* passed to `debug` is `nil` (or if it is not one of the following special values), then the rest of the arguments to `debug` are printed at the top of the '`*Backtrace*`' buffer. This mechanism is used to display a message to the user.

However, if the first argument passed to `debug` is one of the following special values, then it has special significance. Normally, these values are passed to `debug` only by the internals of Emacs and the debugger, and not by programmers calling `debug`.

The special values are:

`lambda` When the first argument is `lambda`, the debugger displays '`Entering:`' as a line of text at the top of the buffer. This means that a function is being entered when `debug-on-next-call` is non-`nil`.

`debug` When the first argument is `debug`, the debugger displays '`Entering:`' just as in the `lambda` case. However, `debug` as the argument indicates that the reason for entering the debugger is that a function set to debug on entry is being entered.

In addition, `debug` as the first argument directs the debugger to mark the function that called `debug` so that it will invoke the debugger when exited. (When `lambda` is the first argument, the debugger does not do this, because it has already been done by the interpreter.)

`t` When the first argument is `t`, the debugger displays the following as the top line in the buffer:

```
Beginning evaluation of function call form:
```

This indicates that it was entered due to the evaluation of a list form at a time when `debug-on-next-call` is non-`nil`.

`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:`' on 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.

15.1.8 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, `backtrace` is called explicitly in a Lisp expression. When the expression is evaluated, the backtrace is printed to the stream `standard-output`: in this case, to the buffer '`backtrace-output`'. Each line of the backtrace represents one function call. If the arguments of the function call are all known, they are displayed; if they are being computed, that fact is stated. 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

This variable determines whether the debugger is called before the next `eval`, `apply` or `funcall`. It is automatically reset to `nil` when the debugger is entered.

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 eval frame *level* levels down to *flag*. If *flag* is non-`nil`, this will cause the debugger to be entered when that frame exits. Even a nonlocal exit through that frame will enter the debugger.

The debug-on-exit flag is an entry in the stack frame of a function call. This flag is examined on every exit from a function.

Normally, this function is only called by the debugger.

command-debug-status Variable

This variable records the debugging status of 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.

The advantage of using this variable rather that defining another global variable is that the data will never carry over to a later other 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 eval frame *level* 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 CAR of 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 the argument is out of range, `backtrace-frame` returns `nil`.

15.2 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 not 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.

15.2.1 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 to study 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.

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.

Then 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`, since the old indentation is probably appropriate to the intended parentheses.

After you think you have fixed the problem, use `C-M-q` again. It should not change anything, if the problem is really fixed.

15.2.2 Excess Close Parentheses

To deal with an excess close parenthesis, first insert an open parenthesis at the beginning of the file 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 the 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`, since the old indentation is probably appropriate to the intended parentheses.

15.3 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. Regardless, 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, it can't localize the error precisely, but can still show you which function to check.

15.4 Edebug

Edebug is a source-level debugger for Emacs Lisp programs that provides the following features:

- Step through evaluation, stopping before and after each expression.
- Set conditional or unconditional breakpoints.
- Trace slow or fast stopping briefly at each stop point, or each breakpoint.
- Evaluate expressions as if outside of Edebug.
- Automatically reevaluate a list of expressions and display their results each time Edebug updates the display.
- Output trace info on function enter and exit.

The first three sections of this chapter should tell you enough about Edebug to enable you to use it.

15.4.1 Using Edebug

To debug a Lisp program with Edebug, you must first *prepare* the Lisp functions that you want to debug. See Section 15.4.2 [Edebug Prepare], page 208.

Once a function is prepared, any call to the function activates Edebug. This involves entering a recursive edit which is a level of Edebug activation.

Activating Edebug may stop execution and let you step through the function, or it may continue execution while checking for debugging commands, depending on the selected Edebug execution mode. See Section 15.4.3 [Edebug Modes], page 208.

Within Edebug, you normally view an Emacs buffer showing the source of the Lisp function you are debugging. We call this the *Edebug buffer*—but note that it is not always the same buffer, and it is not reserved for Edebug use.

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 prepare the definition of `fac` (shown below) for Edebug 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. Stop points before variables are optional, under the control of the value of `edebug-stop-before-symbols`. Here we show with periods the stop points normally found in the function `fac`:

```
(defun fac (n)
  .(if .(< 0 n.).
      .(* n. .(fac (1- n.).).).
    1).)
```

While a buffer is the Edebug buffer, the special commands of Edebug are available in it, instead of many usual editing commands. Type `?` to display a list of Edebug commands. In particular, you can exit the innermost Edebug activation level with `C-]`, and you can return all the way to top level with `q`.

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 state that you get:

```
(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. Use the `r` command to display the value again later.

While Edebug is active, it catches all errors (if `debug-on-error` is non-`nil`) and quits (if `debug-on-quit` is non-`nil`) instead of the standard debugger. When this happens, Edebug displays the last stop point that it knows about. This may be the location of a call to a function which was not prepared for Edebug debugging, within which the error actually occurred.

15.4.2 Preparing Functions for Edebug

In order to use Edebug to debug a function, you must first *prepare* the function. Preparing a function inserts additional code into it which invokes Edebug at the proper places.

Any call to an Edebug-prepared function activates Edebug. This may or may not stop execution, depending on the Edebug execution mode in use. Some Edebug modes only update the display to indicate the progress of the evaluation without stopping execution. The default initial Edebug mode is `step` which does stop execution. See Section 15.4.3 [Edebug Modes], page 208.

Once you have loaded Edebug, the command `C-M-x` is redefined so that when used on a function or macro definition, it prepares the function or macro if given a prefix argument. If the variable `edebug-all-defuns` is non-`nil`, that inverts the meaning of the prefix argument: then `C-M-x` prepares the function or macro *unless* it has a prefix argument. The default value of `edebug-all-defuns` is `nil`. The command `M-x edebug-all-defuns` toggles the value of the variable `edebug-all-defuns`.

If `edebug-all-defuns` is non-`nil`, then the commands `eval-region` and `eval-current-buffer` also prepare any functions and macros whose definitions they evaluate.

Loading a file does not prepare functions and macros for Edebug.

See Chapter 8 [Evaluation], page 103 for discussion of other evaluation functions available inside of Edebug.

15.4.3 Edebug Modes

Edebug supports several execution modes for running the program you are debugging. We call these alternatives *Edebug modes*; do not confuse them with major modes or minor modes. The current Edebug mode determines how Edebug displays the progress of the evaluation, whether it stops at each stop point, or continues to the next breakpoint, for example.

Normally, you specify the Edebug mode for execution 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.

`SPC` Step: stop at the next stop point encountered.

`t` Trace: pause one second at each Edebug stop point.

`T` Rapid trace: mention each stop point, but don't actually pause.

`g` Go: run until the next breakpoint. See Section 15.4.6 [Breakpoints], page 211.

`c` Continue: pause for one second at each breakpoint, but don't stop.

`C` Continue: mention each breakpoint, but don't actually pause.

`G` Non-stop: ignore breakpoints. You can still stop the program by typing `S`.

In general, the execution modes earlier in the above list run the program more slowly or stop sooner.

When you enter a new Edebug level, the mode comes from the value of the variable `edebug-initial-mode`. By default, this specifies *step* mode. If the mode thus specified is not stop mode, then the Edebug level executes the program (or part of it).

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 that you typed. For example, typing `t` during execution switches to trace mode at the next stop point.

You can use the `S` command to stop execution without doing anything else.

If your function happens to read input, a character you hit 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.

15.4.4 Stepping

`f` Run the program forward over one expression. More precisely, set a temporary breakpoint at the position that `C-M-f` would reach, then execute in go mode so that the program will stop at breakpoints. See Section 15.4.6 [Breakpoints], page 211 for the details on 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 `condition-case`, for example.

This command does `forward-sexp` starting at point rather than the stop point, thus providing more flexibility. If you want to execute one expression from the current stop point, type `w` first, to move point there.

`o` Run the program until the end of the containing sexp. If the containing sexp is the top level defun, run until just before the function returns. If that is where you are now, return from the function and then stop.

This command does not exit the currently executing function unless you are positioned after the last sexp of the function.

If the program does a non-local exit, it may fail to reach the temporary breakpoint that this command sets.

`i` Step into the function about to be called. Use this command before any of the arguments of the function call are evaluated, since otherwise it is too late.

One undesirable side effect of using `edebug-step-in` is that the next time the stepped-into function is called, Edebug will be called there as well.

`h` Proceed to the stop point near where point is. This uses a temporary breakpoint.

The `f` command runs the program forward over one expression. More precisely, set a temporary breakpoint at the position that `C-M-f` would reach, then execute in go mode so that the program will stop at breakpoints. See Section 15.4.6 [Breakpoints], page 211 for the details on 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 `condition-case`, for example.

The `f` command uses the existing value of point as the basis for setting the breakpoint, because that is more flexible. To execute one expression *from the current stop point*, type `w` and then `f`.

The `o` command continues "out of" an expression. It places a temporary breakpoints at the end of the containing sexp. If the containing sexp is the top level defun, it continues until just before the function returns. If that is where you are now, it returns from the function and then stops.

This command does not exit the currently executing function unless you are positioned after the last sexp of the function.

The `i` command steps into the function about to be called. Use this command before any of the arguments of the function call are evaluated, since otherwise it is too late.

One undesirable side effect of using `i` is that the next time the stepped-into function is called, Edebug will be called there as well.

The `h` command proceeds to the stop point near where point is, using a temporary breakpoint.

All the commands in this section 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.

15.4.5 Miscellaneous

Some miscellaneous commands are described here.

`C-]` Abort one level of Edebug activity.

`q` Return to the top level editor command loop. This exits all recursive editing levels, including all levels of Edebug activity.

`r` Redisplay the result of the previous expression in the echo area.

`d` Display a backtrace, excluding Edebug's own functions for clarity.

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.

15.4.6 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 15.4.1 [Using Edebug], page 206—even before a symbol. For setting and unsetting breakpoints, the stop point that is affected is the first one at or after point in the Edebug buffer. Here are the Edebug commands for breakpoints:

`b` Set a breakpoint at the stop point at or after point. 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 the current point.

`x` *cond* `RET`

Set a conditional breakpoint which stops the program only if *cond* evaluates to a non-`nil` value. If you use a prefix argument, the breakpoint is temporary (it turns off the first time it stops the program).

`B` Move point to the next breakpoint in the current function definition.

While in Edebug, you can set a breakpoint with `b` (`edebug-set-breakpoint`) and unset one with `u` (`edebug-unset-breakpoint`). First you must move point to a position at or before the desired Edebug stop point, then hit the key to change the breakpoint. Unsetting a breakpoint that has not been set does nothing.

Reevaluating the function with `edebug-defun` clears all breakpoints in the function.

A *conditional breakpoint* tests a condition each time the program gets there, to decide whether to stop. To set a conditional breakpoint, use `x`, and specify the condition expression in the minibuffer.

You can make both conditional and unconditional breakpoints *temporary* by using a prefix arg to the command to set the breakpoint. After breaking at a temporary breakpoint, it is automatically cleared.

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` (`edebug-next-breakpoint`) command, which moves point to the next breakpoint in the function following point, or to the first breakpoint if there are no following breakpoints. This command does not continue execution—it just moves point in the buffer.

15.4.7 Views

These Edebug commands let you view aspects of the buffer and window status that obtained before entry to Edebug.

`v` View the outside window configuration.

`p` Temporarily display the outside current buffer with point at its outside position.

`w` Switch back to the buffer showing the currently executing function, and move point back to the current stop point.

`W` Forget the saved outside window configuration—so that the current window configuration will remain unchanged when you next exit Edebug (by continuing the program). Also toggle the `edebug-save-windows` variable.

15.4.8 Evaluation

While within Edebug, you can evaluate expressions "as if" Edebug were not running. Edebug tries to be invisible to the expression's evaluation.

`e` *exp* `RET` Evaluate expression *exp* in the context outside of Edebug. That is, Edebug tries to avoid altering the effect of *exp*.

`M-ESC` *exp* `RET`
Evaluate expression *exp* in the context of Edebug itself.

`C-x C-e` Evaluate the expression in the buffer before point, in the context outside of Edebug.

15.4.9 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 is reentered.

`E` Switch to the evaluation list buffer '`*edebug*`'.

In the '`*edebug*`' buffer you can use the commands of Lisp Interaction as well as these special commands:

`LFD` Evaluate the expression before point, in the context outside of Edebug, and insert the value in the buffer.

`C-x C-e` Evaluate the expression before point, in the context outside of Edebug.

`C-c C-u` Build a new evaluation list from the first expression of each group, reevaluate and redisplay. Groups are separated by a line starting with a comment.

`C-c C-d` Delete the evaluation list group that point is in.

`C-c C-w` Switch back to the Edebug buffer at the current stop point.

You can evaluate expressions in the evaluation list window with `LFD` 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 of your function unless you add them to the evaluation list with `C-c C-u` (`edebug-update-eval-list`). This command builds a new list from the first expression of each *evaluation list group*. Groups are separated by a line starting with a comment.

When the evaluation list is redisplayed, each expression is displayed followed by the result of evaluating it, and a comment line. If an error occurs during an evaluation, 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*>
;-------------------------------------------------------------------
(point-min)
1
;-------------------------------------------------------------------
(point-max)
2
;-------------------------------------------------------------------
edebug-outside-point-max
"Symbol's value as variable is void: edebug-outside-point-max"
;-------------------------------------------------------------------
(recursion-depth)
0
;-------------------------------------------------------------------
this-command
eval-last-sexp
;-------------------------------------------------------------------
```

To delete a group, move point into it and type `C-c C-d` (`edebug-delete-eval-item`), or simply delete the text for it and update the evaluation list with `C-c C-u`. When you add a new group, be sure to add a comment at the beginning.

After selecting '`*edebug*`', you can return to the source code buffer (the Edebug buffer) with `C-c C-w`. The `*edebug*` buffer is killed when you continue execution of your function, and recreated next time it is needed.

15.4.10 Printing

If the results of your expressions contain circular references to other parts of the same structure, you can print them more usefully with the '`custom-print`'.

To load the package and activate custom printing only for Edebug, simply use the command `edebug-install-custom-print-funcs`. Then set the variable `print-circle` to enable special handling of circular structure. To restore the standard print functions, use `edebug-reset-print-funcs`.

15.4.11 The Outside Context

Edebug tries to be transparent to the program you are debugging, but it does not succeed completely. In addition, most evaluations you do within Edebug (see Chapter 8 [Evaluation], page 103) occur in the same outside context which is temporarily restored for the evaluation. This section explains precisely how use Edebug fails to be completely transparent.

15.4.11.1 Just Checking

Whenever Edebug is entered just to think about whether to take some action, it needs to save and restore certain data.

- `max-lisp-eval-depth` and `max-specpdl-size` are both incremented for each `edebug-enter` call so that your code should not be impacted by Edebug frames on the stack.
- The state of keyboard macro execution is saved and cleared out.

15.4.11.2 Outside Window Configuration

When Edebug needs to display something (e.g., in trace mode), it saves the current window configuration from "outside" Edebug (see Section 25.16 [Window Configurations], page 420). When you exit Edebug (by continuing the program), it restores the previous window configuration.

Emacs redisplays only when it pauses. Usually, when you continue Edebug, 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 window configuration for within Edebug, with no interruption.

The window configuration proper does not include which buffer is current or where point and mark are in the current buffer, but Edebug saves and restores these also.

Entry to Edebug for displaying something also saves and restores the following data. (Some of these variables are deliberately not restored if an error or quit signal occurs.)

- The position of point in the Edebug buffer is saved and restored if the outside current buffer is the same as the Edebug buffer.
- The outside window configuration, as described above, is saved and restored if `edebug-save-windows` is non-`nil`.
- The current buffer, and point and mark in the current buffer are normally saved and restored even if the current buffer is the same as the Edebug buffer.
- The value of point in each displayed buffers 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. This permits recursive use of Edebug, and use of Edebug while using GUD.
- `cursor-in-echo-area` is locally bound to `nil` so that the cursor shows up in the window.

15.4.11.3 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, for whichever buffer was current.
- `last-command`, `this-command`, `last-command-char`, and `last-input-char`. Commands used within Edebug do not affect these variables outside of Edebug.

  But note that it is not possible to preserve the status reported by `(this-command-keys)` and the variable `unread-command-char`.
- `standard-output` and `standard-input`.

15.4.11.4 Side Effects

Edebug operation unavoidably alters some data in Emacs, and this can interfere with debugging certain programs.

- Lisp stack usage is increased, but the limits, `max-lisp-eval-depth` and `max-specpdl-size`, are also increased proportionally.
- The key sequence returned by `this-command-keys` is changed by executing commands within Edebug and there appears to be no way to reset the key sequence from Lisp.
- Edebug cannot save and restore the value of `unread-command-char` or `unread-command-events`. Entering Edebug while these variables have nontrivial values 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.
- Horizontal scrolling of the Edebug buffer is not recovered.

15.4.12 Macro Calls

When Edebug prepares for stepping through an expression that uses a Lisp macro, it needs additional advice to do the job properly. This is because there is no way to tell which parts of the macro call are forms to be evaluated. You must explain the format of calls to each macro to enable Edebug to handle it. To do this, use `def-edebug-form-spec` to define the format of calls to a given macro.

def-edebug-form-spec *macro argpattern* Macro

Specify which parts of a call to macro *macro* are subexpressions to be evaluated. The second argument, *argpattern*, details what the argument list looks like.

Here is a table of the possibilities for *argpattern* and its subexpressions:

`t` A list of any number of evaluated arguments.

`0` A list of unevaluated arguments.

`sexp` A single unevaluated object.

`form` A single evaluated expression.

`symbolp` An unevaluated symbol.

`integerp` An unevaluated number.

`stringp` An unevaluated string.

`vectorp` An unevaluated vector.

`atom` An unevaluated object that is not a cons cell.

`function` A function argument: a quoted symbol, a quoted lambda expression, or a form (that should evaluate to a function or lambda expression). Edebug treats the body of a lambda expression treated as evaluated.

function A function serves as a predicate—it designates the set of possible arguments for which it would return non-`nil`.

`'`*object* The precise object *object*, treated as unevaluated.

`(`*patterns*`)`

A list whose elements are described by *patterns*. A sublist of the same format as the top level, processed recursively.

`[`*patterns*`]`

A sequence of arguments that are described by *patterns*.

`&optional`

This symbol serves as a flag saying that all following elements in the specification list at this level are optional. They may or may not match arguments; as soon as one does not match, processing of the specification list at this level terminates. To make just one item optional, use `[&optional` *pattern*`]`.

`&rest` This symbol serves as a flag saying that the following elements in the specification list at this level may be repeated, in order, zero or more times. Only one `&rest` may appear at the same level of a specification list, and `&rest` must not be followed by `&optional`.

To specify repetition of certain types of arguments, followed by dissimilar arguments, use `[&rest` *patterns*...`]`.

`&or` This symbol serves as an operator saying that the following elements in the specification list at this level are alternatives. To group two or more list elements as one alternative, bracket them in `[...]`. Only one `&or` may appear in a list, and it may not be followed by `&optional` or `&rest`. One of the alternatives must match, unless the `&or` is preceded by `&optional` or `&rest`.

If the actual arguments of a macro call fail to match the specification, taking account of alternatives, optional arguments and repeated arguments, Edebug reports a syntax error in use of the macro.

The combination of backtracking, `&optional`, `&rest`, `&or`, and `[...]` for grouping provides the equivalent of regular expressions. The `(...)` lists require balanced parentheses, which is the only context free (finite state with stack) construct supported.

Here are some examples of using `def-edebug-form-spec`. First, for the `let` special form:

```
(def-edebug-form-spec let
  '((&rest
    &or symbolp (symbolp &optional form))
   &rest form))
```

Here's the spec for the `for` loop macro (see Section 12.6 [Problems with Macros], page 169) and for the `case` and `do` macros in 'cl.el':

```
(def-edebug-form-spec for
  '(symbolp 'from form 'to form 'do &rest form))

(def-edebug-form-spec case
  '(form &rest (sexp form)))

(def-edebug-form-spec do
  '((&rest &or symbolp (symbolp &optional form form))
    (form &rest form)
    &rest body))
```

Finally, the functions `mapcar`, `mapconcat`, `mapatoms`, `apply`, and `funcall` all take function arguments, and Edebug defines specifications for them. Here's one example:

```
(def-edebug-form-spec apply '(function &rest form))
```

The backquote (`) macro results in an expression that is not necessarily evaluated. Edebug cannot step through code generated by use of backquote.

15.4.13 Edebug Options

These options affect the behavior of Edebug:

edebug-all-defuns User Option

If non-`nil`, normal evaluation of `defun` and `defmacro` forms prepares the functions and macros for stepping with Edebug. This applies to `eval-defun`, `eval-region` and `eval-current-buffer`.

The default value is `nil`.

edebug-stop-before-symbols User Option

If non-`nil`, Edebug places stop points before symbols as well as after.

This option takes effect for a function when you prepare it for stepping with Edebug. Changing the option's value during execution of Edebug has no effect on the functions already set up for Edebug execution.

edebug-save-windows User Option

If non-`nil`, save and restore window configuration on Edebug calls. It takes some time to save and restore, so if your program does not care what happens to the window configurations, it is better to set this variable to `nil`.

The default value is `t`.

edebug-save-point User Option

If non-`nil`, Edebug saves and restores point and the mark in source code buffers. The default value is `t`.

edebug-save-displayed-buffer-points User Option

If non-`nil`, save and restore point in all buffers when entering Edebug mode.

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, the buffer's point will be changed to the window's point.

Saving and restoring is an expensive operation since it visits each window and each displayed buffer twice for each Edebug call, so it is best to avoid it if you can.

The default value is `nil`.

edebug-initial-mode User Option

If this variable is non-`nil`, it specifies an Edebug mode to start in each time the program enters a new Edebug recursive-edit level.

Possible values are `step`, `go`, `Go-nonstop`, `trace`, `Trace-fast`, `continue`, and `Continue-fast`.

The default value is `step`.

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. You can customize this display by replacing the functions `edebug-print-trace-entry` and `edebug-print-trace-exit`.

The default value is `nil`.

16 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 [Types of Lisp Object], 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).

16.1 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, reading the text '`(a . 5)`' returns 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. 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 are `a` and `b`.

However, these two operations are not precisely inverses. There are two kinds of exceptions:

- Printing can produce text that cannot be read. For example, buffers, windows, subprocesses and markers print into 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.

16.2 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*, one character per call. Normally *function* is called with no arguments, and should return a character.

  Occasionally *function* is called with one argument (always a character). When that happens, *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 want to "put it back where it came from".

`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` used as a stream means that the value of `standard-input` should be used instead; that value is the *default input stream*, and must be a non-`nil` input stream.

symbol
: A symbol as output stream is equivalent to the symbol's function definition (if any).

Here is an example of reading from a stream which 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 at the beginning of the buffer. Reading skips any amount of whitespace preceding the significant text.

In Emacs 18, reading a symbol discarded the delimiter terminating the symbol. Thus, point would end up at the beginning of 'contents' rather than after 'the'. The Emacs 19 behavior is superior because it correctly handles input such as 'bar(foo)' where the delimiter that ends one object is needed as the beginning of another object.

Here is an example of reading from a stream that is a marker, initialized to point 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 6 in foo>    ;; After 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 characters 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
     ⇒ (41)
```

Note that the close parenthesis remains in the list. The reader has read it, discovered that it ended the input, and unread it. Another attempt to read from the stream at this point would get an error due to the unmatched close parenthesis.

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.

16.3 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 results if an unterminated list or vector is found.

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 *end* is also supplied, then reading stops at 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 . 6)
;; 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`.

16.4 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* is in at the marker position. The position advances as characters are inserted. The value of point in the buffer has no effect when the stream is a marker.

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 free to do anything at all with the characters it receives.

`t` The output characters are displayed in the echo area.

`nil` `nil` specified as an output stream means that the value of `standard-output` should be used as the output stream; that value is the *default output stream*, and must be a non-`nil` output stream.

symbol A symbol as output stream is equivalent to the symbol's function definition (if any).

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 points in buffer `foo`, between the 't' and the 'h' in the word 'the'. At the end, the marker has been advanced over the inserted text so that it still points 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 ----------
m
     ⇒ #<marker at 11 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 35 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 an output stream which is a function. 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 71). 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\"
"
```

16.5 Output Functions

This section describes the Lisp functions for printing Lisp objects.

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 are used to distinguish strings from symbols, and to prevent punctuation characters in strings and symbols from being taken as delimiters. 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 it is best to 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 better to print without quoting.

Printing a self-referent Lisp object requires an infinite amount of text. In certain cases, trying to produce this text leads to a stack overflow. 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` 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 any spaces or 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 quoting characters are not used and double-quotes are not printed 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 4.5 [String Conversion], page 58, for other ways to obtain the printed representation of a Lisp object as a string.

16.6 Variables Affecting Output

standard-output Variable

The value of this variable is the default output stream, used when the *stream* argument is omitted or `nil`.

print-escape-newlines Variable

If this variable is non-`nil`, then newline characters in strings are printed as '`\n`'. Normally they are printed as actual newlines.

This variable affects the print functions `prin1` and `print`, as well as everything that uses them. 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-length Variable

The value of this variable is the maximum number of elements of a list that will be printed. If the list being printed has more than this many elements, then 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 that will be 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.

This variable exists in version 19 and later versions.

17 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 screen, in the same place as the echo area, but only while it is in use for reading an argument.

17.1 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. (Sometime frames have no minibuffer window, and sometimes a special kind of frame contains nothing but a minibuffer window; see Section 26.6 [Minibuffers and Frames], page 429.)

The minibuffers window is normally a single line; you can resize it temporarily with the window sizing commands, but reverts to its normal size when the minibuffer is exited.

A *recursive minibuffer* may be created when there is an active minibuffer and a command is invoked that requires input from a 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 17.8 [Minibuffer Misc], page 251).

Like other buffers, a minibuffer may use any of several local keymaps (see Chapter 19 [Keymaps], page 291); these contain various exit commands and in some cases completion commands. See Section 17.5 [Completion], page 237.

- `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.

17.2 Reading Text Strings with the Minibuffer

The minibuffer is usually used to read text which is returned as a string, but can also be used to read a Lisp object in textual form. The most basic primitive for minibuffer input is `read-from-minibuffer`.

read-from-minibuffer *prompt-string* &optional *initial keymap read hist* 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 16.3 [Input Functions], page 224).

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, if *initial* is a string; its contents are inserted into the minibuffer as initial contents. The text thus inserted is treated as if the user had inserted it; the user can alter it with Emacs editing commands.

The value of *initial* may also be a cons cell of the form `(`*string* `.` *position*`)`. This means to insert *string* in the minibuffer but put the cursor *position* characters from the beginning, rather than at the end.

If *keymap* is non-`nil`, that keymap is the local keymap to use while reading. 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 minibuffer input for various applications including 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 17.4 [Minibuffer History], page 236.

When the user types a command to exit the minibuffer, the current minibuffer contents are usually made into a string which becomes the value of `read-from-minibuffer`. However, if *read* is non-`nil`, `read-from-minibuffer` converts the result to a Lisp object, and returns that object, unevaluated.

Suppose, for example, you are writing a search command and want to record the last search string and provide it as a default for the next search. Suppose that the previous search string is stored in the variable `last-search-string`. Here is how you can read a search string while providing the previous string as initial input to be edited:

```
(read-from-minibuffer "Find string: " last-search-string)
```

Assuming the value of `last-search-string` is '`No`', and the user wants to search for '`Nope`', the interaction looks like this:

```
(setq last-search-string "No")

(read-from-minibuffer "Find string: " last-search-string)
---------- Buffer: Minibuffer ----------
Find string: No⋆
---------- Buffer: Minibuffer ----------
;; The user now types pe RET:
     ⇒ "Nope"
```

This technique is no longer preferred for most applications; it is usually better to use a history list.

read-string *prompt* &optional *initial* Function

This function reads a string from the minibuffer and returns 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-string prompt initial)
≡
(read-from-minibuffer prompt initial nil nil)
```

minibuffer-local-map Variable

This is the default local keymap for reading from the minibuffer. It is the keymap used by the minibuffer for local bindings in the function `read-string`. By default, it makes the following bindings:

`LFD` `exit-minibuffer`

`RET` `exit-minibuffer`

`C-g` `abort-recursive-edit`

`M-n` and `M-p`
`next-history-element` and `previous-history-element`

`M-r` `next-matching-history-element`

`M-s` `previous-matching-history-element`

read-no-blanks-input *prompt* &optional *initial* 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* and *initial* 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:

`LFD` `exit-minibuffer`

`SPC` `exit-minibuffer`

`TAB` `exit-minibuffer`

`RET` `exit-minibuffer`

`C-g` `abort-recursive-edit`

`?` `self-insert-and-exit`

`M-n` and `M-p`
`next-history-element` and `previous-history-element`

`M-r` `next-matching-history-element`

`M-s` `previous-matching-history-element`

17.3 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 in the minibuffer and returns it, without evaluating it. The arguments *prompt* and *initial* are used as in `read-from-minibuffer`; in particular, *initial* must be a string or `nil`.

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 in 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 16.5 [Output Functions], page 227.

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. 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 evaluating 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.

17.4 Minibuffer History

A minibuffer history list records previous minibuffer inputs so the user can reuse them conveniently. There are many separate history lists which contain different kinds of inputs. The Lisp programmer's job is 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 If you specify a variable (a symbol), that variable is the history list.

(*variable* . *startpos*)
: If you specify a cons cell of this form, then *variable* is the history list variable, and *startpos* specifies the initial history position (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 *initial*, 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. The value of the history list variable is a list of strings, most recent first.

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.

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.

17.5 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.

17.5.1 Basic Completion Functions

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 which implements a virtual set of strings.

If *collection* is an alist (see Section 5.8 [Association Lists], page 83), completion compares the CAR of each cons cell in it against *string*; if the beginning of the CAR equals *string*, the cons cell matches. If no cons cells match, `try-completion` returns `nil`. If only one cons cell matches, and the match is exact, then `try-completion` returns `t`. Otherwise, the value is the longest initial sequence common to all the matching strings in the alist.

If *collection* is an obarray (see Section 7.3 [Creating Symbols], page 97), the names of all symbols in the obarray form the space of possible completions. They are tested and used just like the CARs of the elements of an association list. (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.

It is also possible to use a function symbol 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 17.5.2 [Programmed Completion], page 240.

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* — Function

This function returns a list of all possible completions, instead of the longest substring they share. The parameters to this function are the same as to `try-completion`.

If *collection* is a function, it is called with three arguments: *string*, *predicate* and `t`, and `all-completions` returns whatever the function returns. See Section 17.5.2 [Programmed Completion], page 240.

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))
 (function test))
     ⇒ ("foobar1" "foobar2")
```

completion-ignore-case — Variable

If the value of this variable is non-`nil`, Emacs does not consider case significant in completion.

The two functions `try-completion` and `all-completions` have nothing in themselves to do with minibuffers. However, completion is most often used there, which is why it is described in this chapter.

17.5.2 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`. This command arranges to pass the 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 an exact match already, or `nil` if the string matches no possibility.
- `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 which 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 22.10.6 [File Name Completion], page 370.

17.5.3 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* 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. 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.

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 `t`, the user is not allowed to exit unless the input completes to an element of *collection*. If *require-match* is neither `nil` nor `t`, then `completing-read` does not exit unless the input typed is itself an element of *collection*. To accomplish this, `completing-read` calls `read-minibuffer`. It uses the value of `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`.

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 17.4 [Minibuffer History], page 236.

Case is ignored when comparing the input against the possible matches if the built-in variable `completion-ignore-case` is non-`nil`. See Section 17.5.1 [Basic Completion], page 237.

For example:

```
(completing-read
 "Complete a foo: "
 '(("foobar1" 1) ("barfoo" 2) ("foobaz" 3) ("foobar2" 4))
 nil t "fo")
;; After evaluating 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 which actually do completion. Here they are:

`minibuffer-completion-table`
: This variable is bound to the *collection* argument. It is passed to the `try-completion` function.

`minibuffer-completion-predicate`
: This variable is bound to the *predicate* argument. It is passed to the `try-completion` function.

`minibuffer-completion-confirm`
: This variable is bound to the *require-match* argument. It is used in the `minibuffer-complete-and-exit` function.

17.5.4 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`.

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 which exits the minibuffer unconditionally. By default, this keymap makes the following bindings:

`?` `minibuffer-completion-help`

`SPC` `minibuffer-complete-word`

`TAB` `minibuffer-complete`

`LFD` `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 all the minibuffer completion functions, 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 31 [Syntax Tables], page 523.

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 non-`nil`. If confirmation *is* required, it is given by repeating this command immediately.

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 16 [Streams], page 221, 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.

17.5.5 High-Level Completion Functions

This section describes the higher-level convenient functions for reading certain sorts of names with completion.

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. 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 this; 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 evaluating 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-command *prompt* 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 18.3 [Interactive Call], page 260.

```
(read-command "Command name? ")

;; After evaluating the preceding expression,
;;   the following appears in the 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 the `completing-read` function. It uses the `commandp` predicate to allow only commands to be entered, and it uses the variable `obarray` so as to be able to complete all extant Lisp symbols:

```
(read-command prompt)
≡
(intern (completing-read prompt obarray 'commandp t nil))
```

read-variable *prompt* Function

This function reads the name of a user variable and returns it as a symbol.

```
(read-variable "Variable name? ")

;; After evaluating 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` will return `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))
```

17.5.6 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`.

If *existing* is non-`nil`, then the name must refer to an existing file; then `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.

The argument *directory* specifies the directory to use for completion of relative file names. Usually it is inserted in the minibuffer as initial input as well. It defaults to the current buffer's default directory.

If you specify *initial*, that is an initial file name to insert in the buffer along with *directory*. In this case, point goes after *directory*, before *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`.

Here is an example:

```
(read-file-name "The file is ")

;; After evaluating 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 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. 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 containing 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 ----------
```

17.5.7 Lisp Symbol Completion

If you type a part of a symbol, and then type `M-TAB` (`lisp-complete-symbol`), this command attempts to fill in as much more of the symbol name as it can. Not only does this save typing, but it can help you with the name of a symbol that you have partially forgotten.

lisp-complete-symbol Command

This function performs completion on the symbol name preceding point. The name is completed against the symbols in the global variable `obarray`, and characters from the completion are inserted into the buffer, making the name longer. If there is more than one completion, a list of all possible completions is placed in the '*Help*' buffer. The bell rings if there is no possible completion in obarray.

If an open parenthesis immediately precedes the name, only symbols with function definitions are considered. (By reducing the number of alternatives, this may succeed in completing more characters.) Otherwise, symbols with either a function definition, a value, or at least one property are considered.

lisp-complete-symbol returns t if the symbol had an exact, and unique, match; otherwise, it returns nil.

In the following example, the user has already inserted '(forwa' into the buffer 'foo.el'. The command lisp-complete-symbol then completes the name to '(forward-'.

```
---------- Buffer: foo.el ----------
(forwa⋆
---------- Buffer: foo.el ----------
(lisp-complete-symbol)
     ⇒ nil
---------- Buffer: foo.el ----------
(forward-⋆
---------- Buffer: foo.el ----------
```

17.6 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.

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 35.4 [The Echo Area], page 583), which uses the same screen space as the minibuffer. The cursor moves to the echo area while the question is being asked.

The meanings of answers, even '`y`' and '`n`', are not hardwired. They are controlled by the keymap `query-replace-map`. See Section 30.4 [Replacement], page 515.

In the following example, the user first types `q`, which is invalid. At the next prompt the user types `n`.

```
(y-or-n-p "Do you need a lift? ")

;; After evaluating 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 ----------
```

Note that we show successive lines of echo area messages here. Only one actually appears on the screen at a time.

yes-or-no-p *prompt* Function

This function asks the user a question, expecting input in 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 evaluating 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 ----------
```

17.7 Asking Multiple Y-or-N Queries

map-y-or-n-p *prompter actor list* &optional *help action-alist* — Function

This function, new in Emacs 19, asks the user a series of questions, reading a single-character answer in the echo area for each one.

The value of *list* specifies what varies from question to question within the series. 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 or `nil` meaning there are no more 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.

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 30.4 [Replacement], page 515.

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.

The return value of `map-y-or-n-p` is the number of objects acted on.

17.8 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 18.4 [Command Loop Info], page 263).

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*. At the time of printing, we have not made a final decision about how to get the pattern interactively or how to match it against history elements.

next-matching-history-element *pattern* Command

This command replaces the minibuffer contents with the value of the next (newer) history element that matches *pattern*.

minibuffer-help-form Variable

The current value of this variable is used to rebind `help-form` locally inside the minibuffer (see Section 21.5 [Help Functions], page 341).

minibuffer-window &optional *frame* Function

This function returns the window that is used for the minibuffer. In Emacs 18, there is one and only one minibuffer window; this window always exists and cannot be deleted. In Emacs 19, each frame can have its own minibuffer, and this function returns the minibuffer window used for frame *frame* (which defaults to the currently selected frame).

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 there is more than one frame.

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 18.10 [Recursive Editing], page 285):

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`) which use minibuffers even while in the minibuffer window. Such invocation produces a recursive editing level for a new minibuffer. The outer-level minibuffer is invisible while you are editing the inner one.

This variable only affects invoking the minibuffer while the minibuffer window is selected. If you switch windows while in the minibuffer, you can always invoke minibuffer commands while some other window is selected.

If a command name has a property `enable-recursive-minibuffers` which 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 bound to `M-s` in the minibuffer) uses this feature.

18 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.

18.1 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 18.6.1 [Key Sequence Input], page 275). Lisp programs can also do input at a lower level with `read-event` (see Section 18.6.2 [Reading One Event], page 276) or discard pending input with `discard-input` (see Section 18.6.4 [Peeking and Discarding], page 278).

The key sequence is translated into a command through the currently active keymaps. See Section 19.8 [Key Lookup], page 302, 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 is used instead. This is done by the command `execute-extended-command` (see Section 18.3 [Interactive Call], page 260).

Once the command is chosen, it must be executed, which includes reading arguments to be given to it. This is done by calling `command-execute` (see Section 18.3 [Interactive Call], page 260). For commands written in Lisp, the `interactive` specification says how to read the arguments. This may use the prefix argument (see Section 18.9 [Prefix Command Arguments], page 282) or may read with prompting in the minibuffer (see Chapter 17 [Minibuffers], page 231). 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 18.13 [Keyboard Macros], page 288).

If a command runs away, typing `C-g` terminates its execution immediately. This is called *quitting* (see Section 18.8 [Quitting], page 280).

pre-command-hook Variable

> The editor command loop runs this normal hook before each command.

post-command-hook Variable

The editor command loop runs this normal hook after each command, and also when the command loop is entered, or reentered after an error or quit.

18.2 Defining Commands

A Lisp function becomes a command when its body contains, at top level, a form which 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.

18.2.1 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 the way the arguments to the command are to be computed when the command is called interactively.

A command may be called from Lisp programs like any other function, but then the arguments are supplied by the caller 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.
- 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 in the prompt. This is done using `format` (see Section 4.6 [Formatting Strings], page 59). 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 '@').

18.2.2 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 17.5 [Completion], page 237). `?` 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.

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 which is `fboundp`). Existing, Completion, Prompt.

'`b`' The name of an existing buffer. By default, uses the name of the current buffer (see Chapter 24 [Buffers], page 383). 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, 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 a number (see Section 27.1 [Point], page 439). No I/O.

'`D`' A directory name. The default is the current default directory of the current buffer, `default-directory` (see Section 34.3 [System Environment], page 567). Existing, Completion, Default, Prompt.

'`e`' The first or next mouse event in the key sequence that invoked the command. More precisely, '`e`' gets events which are lists, so you can look at the data in the lists. See Section 18.5 [Input Events], page 264. No I/O.

You can use '`e`' more than once in a single command's interactive specification. If the key sequence which invoked the command has *n* events with parameters, the *n*th '`e`' provides the *n*th list event. Events which are not lists, such as function keys and ASCII characters, do not count where '`e`' is concerned.

Even though '`e`' does not use a prompt string, you must follow it with a newline if it is not the last code character.

'f' A file name of an existing file (see Section 22.10 [File Names], page 364). The default directory is `default-directory`. Existing, Completion, Default, Prompt.

'F' A file name. The file need not exist. Completion, Default, Prompt.

'k' A key sequence (see Section 19.1 [Keymap Terminology], page 291). 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`.

'm' The position of the mark as a number. No I/O.

'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 raw prefix argument. If the prefix argument is `nil`, then a number is read as with `n`. Requires a number. 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.) See Section 18.9 [Prefix Command Arguments], page 282. 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 17.2 [Text from Minibuffer], page 232). Terminate the input with either `LFD` 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 17.5.5 [High-Level Completion], page 244. Existing, Completion, Prompt.

'x' A Lisp object specified in printed representation, terminated with a `LFD` or `RET`. The object is not evaluated. See Section 17.3 [Object from Minibuffer], page 234. Prompt.

'X' A Lisp form is read as with `x`, but then evaluated so that its value becomes the argument for the command. Prompt.

18.2.3 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
```

18.3 Interactive Call

After the command loop has translated a key sequence into a definition, it invokes that definition using the function `command-execute`. If the definition is a function that is a command, `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, autoload objects that are declared as interactive (non-`nil` fourth argument to `autoload`), and some of the primitive functions.

A symbol is `commandp` if its function definition is `commandp`.

Keys and keymaps are not commands. Rather, they are used to look up commands (see Chapter 19 [Keymaps], page 291).

See `documentation` in Section 21.2 [Accessing Documentation], page 336, for a realistic example of using `commandp`.

call-interactively *command* &optional *record-flag* Function

This function calls the interactively callable function *command*, reading arguments according to its interactive calling specifications. An error is signaled if *command* cannot be called interactively (i.e., it is not a command). Note that keyboard macros (strings and vectors) are not accepted, even though they are considered commands.

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 18.12 [Command History], page 287.

command-execute *command* &optional *record-flag* Function

This function executes *command* as an editing command. The argument *command* must satisfy the `commandp` predicate; i.e., it must be an interactively callable function or a string.

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.

execute-extended-command *prefix-argument* Command

This function reads a command name from the minibuffer using `completing-read` (see Section 17.5 [Completion], page 237). 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, the value *prefix-argument* is supplied. 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 ----------
M-x forward-word RET
---------- Buffer: Minibuffer ----------
     ⇒ t
```

interactive-p Function

This function returns `t` if the containing function (the one that called `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.) Note that if the containing function was called by Lisp evaluation (or with `apply` or `funcall`), then it was not called interactively.

The usual application 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)
  (and (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)
```

18.4 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 set by copying the value of `this-command` when a command returns to the command loop, except when the command specifies a prefix argument for the following command.

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.

This variable is set by the command loop just before the command is run, and its value is copied into `last-command` when the command finishes (unless the command specifies a prefix argument for the following command).

Some commands change the value of this variable during their execution, simply as a flag for whatever command runs next. In particular, the functions that kill 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.

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 18.5 [Input Events], page 264.

```
(this-command-keys)
;; Now type C-u C-x C-e.
     ⇒ "^U^X^E"
```

last-nonmenu-event Variable

This variable holds the last input event read as part of a key sequence, aside from events resulting from mouse menus.

One use of this variable is to figure out a good default location to pop up another menu.

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-char
;; Now type C-u C-x C-e.
     ⇒ 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 26.7 [Input Focus], page 429.

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 (say `C-x`) and then delays this many seconds before continuing, the key `C-x` is echoed in the echo area. Any subsequent characters in the same command will be echoed as well.

If the value is zero, then command input is not echoed.

18.5 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.

A command invoked using events that are lists can get the full values of these events using the '`e`' interactive code. See Section 18.2.2 [Interactive Codes], page 257.

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.

eventp *object* Function

This function returns non-`nil` if *event* is an input event.

18.5.1 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. In Emacs versions 18 and earlier, characters were the only events.

An input character event consists of a *basic code* between 0 and 255, plus any or all of these *modifier bits*:

meta
: The 2**23 bit in the character code indicates a character typed with the meta key held down.

control
: The 2**22 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**22 (assuming the terminal supports non-ASCII control characters).

shift
: The 2**21 bit in the character code indicates an ASCII control character typed with the shift key held down.

: For letters, the basic code 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**21 bit for those characters.

: However, ASCII provides no way to distinguish `C-A` from `C-A`, so Emacs uses the 2**21 bit in `C-A` and not in `C-a`.

hyper
: The 2**20 bit in the character code indicates a character typed with the hyper key held down.

super The 2**19 bit in the character code indicates a character typed with the super key held down.

alt The 2**18 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.)

In the future, Emacs may support a larger range of basic codes. We may also move the modifier bits to larger bit numbers. Therefore, you should avoid mentioning specific bit numbers in your program. Instead, the way to test the modifier bits of a character is with the function `event-modifiers` (see Section 18.5.9 [Classifying Events], page 271).

18.5.2 Function Keys

Most keyboards also have *function keys*—keys which have names or symbols that are not characters. Function keys are represented in Lisp as symbols; the symbol's name is the function key's label. For example, pressing a key labeled `F1` places the symbol `f1` in the input stream.

For all keyboard events, the event type (which classifies the event for key lookup purposes) is identical to the event—it is the character or the symbol. See Section 18.5.9 [Classifying Events], page 271.

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. Emacs lets you distinguish them if you wish, by returning 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` is set up to map `tab` into 9. Thus, a key binding for character code 9 also applies to `tab`. Likewise for the other symbols in this group. The function `read-char` also 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.

`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.

`left`, `up`, `right`, `down`
: Cursor arrow keys

You can use the modifier keys `CTRL`, `META`, `HYPER`, `SUPER`, `ALT` and `SHIFT` 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 (though the order does not matter in arguments to the key-binding lookup and modification functions).

18.5.3 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
  (column . row) 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 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.

column
row These are the column and row 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 ⟨undefined⟩ [Repeat Events], page ⟨undefined⟩.

The meanings of *buffer-pos*, *row* and *column* 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 `(`*column*` . `*row*`)` 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, *row* does not have meaningful data. For the vertical line, *column* does not have meaningful data.

buffer-pos may be a list containing a symbol (one of the symbols listed above) instead of just the symbol. This is what happens after the imaginary prefix keys for these events are inserted into the input stream. See Section 18.6.1 [Key Sequence Input], page 275.

18.5.4 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
  (column1 . row1) timestamp1)
 (window2 buffer-pos2
  (column2 . row2) timestamp2)
 click-count)
```

For a drag event, the name of the symbol *event-type* contains the prefix '`drag-`'. 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 18.5.3 [Click Events], page 267). 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 which 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.

18.5.5 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 which look exactly like click events (see Section 18.5.3 [Click Events], page 267), except that the name of *event-type* contains the prefix '`down-`'. The '`down-`' prefix follows the modifier key prefixes such as '`C-`' and '`M-`'.

The function `read-key-sequence`, and the Emacs command loop, ignore any button-down events that don't have command bindings. 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.

18.5.6 Repeat Events

If you press the same mouse button more than once in quick succession without moving the mouse, Emacs uses special *repeat* mouse events for the second and subsequent presses.

[1] Button-down is the conservative antithesis of drag.

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, the command binding of the double click event must be written to assume that the single-click command has already run. It must produce the desired results of a double click, starting from the results of a single click.

This means that it is most convenient to give double clicks a meaning that somehow "builds on" the meaning of a single click. This is what user interface experts recommend that double clicks should do.

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 click.

Before the double-click or double-drag event, Emacs generates a *double-down* event when the button is pressed 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 double-down event, followed by either a double-click or a double-drag.

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 quadruple, quintuple, etc. events as separate event types. 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 count as double- and triple-clicks, mouse clicks must be at the same location as the first click, and the number of milliseconds between the first release and the second 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.

18.5.7 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
  (column . row) timestamp))
```

The second element of the list describes the current position of the mouse, just as in a click event (see Section 18.5.3 [Click Events], page 267).

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.

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.

When the user releases the button, that generates a click event. Normally *body* should return when it sees the click event, and discard the event.

18.5.8 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 26.7 [Input Focus], page 429.

Focus events are represented in Lisp as lists that look like this:

```
(switch-frame new-frame)
```

where *new-frame* is the frame switched to.

In X windows, most 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 the 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.

18.5.9 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))
```

Or, 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))
```

Or, 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 the following pair of events:

```
(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))
```

18.5.10 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 which 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 what text to act on.

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 property of a mouse event symbol always has one of `click`, `drag`, and `down` among the modifiers. For example:

```
(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.

18.5.11 Accessing Events

This section describes convenient functions for accessing the data in an event which is a list.

The following functions return the starting or ending position of a mouse-button event. The position is a list of this form:

(*window buffer-position* (*col* . *row*) *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 four 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 location in *position*.

posn-col-row *position* Function
Return the row and column in *position*, as a list `(`*col* `.` *row*`)`.

posn-timestamp *position* Function
Return the timestamp of *position*.

scroll-bar-scale *ratio total* Function
This function multiples (in effect) *ratio* by *total*, rounding the result to an integer. *ratio* is not a number, but rather a pair `(`*num* `.` *denom*`)`.

This 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-col-row (event-start event))
      (- (point-max) (point-min))))
```

18.5.12 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 which conceptually contain keyboard characters; for example, they may be key sequences or keyboard macro definitions. There are special rules for how to put keyboard characters into a string, because they are not limited to the range of 0 to 255 as text characters are.

A keyboard character typed using the `META` key is called a *meta character*. The numeric code for such an event includes the 2**23 bit; it does not even come close to fitting in a string. However, earlier Emacs versions used a different representation for these characters, which gave them codes in the range of 128 to 255. That did fit in a string, and many Lisp programs contain string constants that use '`\M-`' to express meta characters, especially as the argument to `define-key` and similar functions.

We provide backward compatibility to run those programs with special rules for how to put a keyboard character event in a string. Here are the rules:

- If the keyboard event value is in the range of 0 to 127, it can go in the string unchanged.
- The meta variants of those events, with codes in the range of 2**23 to 2**23+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**23 bit, resulting in a value between 128 and 255.
- 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 can construct strings containing events follow these rules.

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.

New programs can avoid dealing with these rules by using vectors instead of strings for key sequences when there is any possibility that these issues might arise.

The reason we changed the representation of meta characters as keyboard events is to make room for basic character codes beyond 127, and support meta variants of such larger character codes.

18.6 Reading Input

The editor command loop reads keyboard input using the function `read-key-sequence`, which uses `read-event`. These and other functions for keyboard input are also available for use in Lisp programs. See also `momentary-string-display` in Section 35.7 [Temporary Displays], page 586, and `sit-for` in Section 18.7 [Waiting], page 279. See Section 34.7 [Terminal Input], page 572, for functions and variables for controlling terminal input modes and debugging terminal input.

For higher-level input facilities, see Chapter 17 [Minibuffers], page 231.

18.6.1 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 full 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 18.5.11 [Strings of Events], page 273). 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.

Quitting is suppressed inside `read-key-sequence`. In other words, a `C-g` typed while reading with this function is treated like any other character, and does not set `quit-flag`. See Section 18.8 [Quitting], page 280.

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"
```

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.

If an input character is an upper case letter and has no key binding, but the 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 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 itself shows nothing special—only the symbol that would normally represent that mouse button and modifier keys. The information about the screen region 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`.

For example, if you call `read-key-sequence` and then click the mouse on the window's mode line, this is what happens:

```
(read-key-sequence "Click on the mode line: ")
     ⇒ [mode-line
         (mouse-1
          (#<window 6 on NEWS> mode-line
           (40 . 63) 5959987))]
```

You can define meanings for mouse clicks in special window regions by defining key sequences using these imaginary prefix keys.

18.6.2 Reading One Event

The lowest level functions for command input are those which read a single event.

read-event 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.

The function `read-event` does not display any message to indicate it is waiting for input; use `message` first, if you wish to display one. If you have not displayed a message, `read-event` does *prompting*: it displays descriptions of the events that led to or were read by the current command. See Section 35.4 [The Echo Area], page 583.

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.

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 `1` (which is ASCII code 49). The second example shows a keyboard macro definition that calls `read-char` from the minibuffer. `read-char` reads the keyboard macro's very next character, which is `1`. The value of this function is displayed in the echo area by the command `eval-expression`.

```
(read-char)
     ⇒ 49

(symbol-function 'foo)
     ⇒ "^[^[(read-char)^M1"
(execute-kbd-macro foo)
     ⊣ 49
     ⇒ nil
```

18.6.3 Quoted Character Input

You can use the function `read-quoted-char` when you want the user to specify a character, and allow the user to specify a control or meta character conveniently with quoting or as an octal character code. The command `quoted-insert` calls 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 up to two more octal digits (but stopping if a non-octal digit is found) and returns the character represented by those digits as an octal number.

Quitting is suppressed when the first character is read, so that the user can enter a `C-g`. See Section 18.8 [Quitting], page 280.

If *prompt* is supplied, it specifies a string for prompting the user. The prompt string is always printed in the echo area and 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
```

18.6.4 Peeking and Discarding

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.

The variable is used because in some cases a function reads a 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 when the functions to read command input are called.

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 it does not recognize.

unread-command-char Variable

This variable holds a character to be read as command input. A value of -1 means "empty".

This variable is pretty much obsolete now that you can use `unread-command-events` instead; it exists only to support programs written for Emacs versions 18 and earlier.

listify-key-sequence *key* Function

This function converts the string or vector *key* to a list of events which you can put in `unread-command-events`. Converting a vector is simple, but converting a string is tricky because of the special representation used for meta characters in a string (see Section 18.5.11 [Strings of Events], page 273).

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 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, a character is read (the character `1`, ASCII code 49). It becomes the value of `last-input-char`, while `C-e` (from the `C-x C-e` command used to evaluate this expression) remains the value of `last-command-char`.

```
(progn (print (read-char))
       (print last-command-char)
       last-input-char)
     ⊣ 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, any characters that have been typed are discarded.

```
(progn (sleep-for 2)
  (discard-input))
     ⇒ nil
```

18.7 Waiting for Elapsed Time or Input

The waiting commands are designed to make Emacs 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 result is `t` if `sit-for` waited the full time with no input arriving (see `input-pending-p` in Section 18.6.4 [Peeking and Discarding], page 278). Otherwise, the value is `nil`.

The optional argument *millisec* specifies an additional waiting period measured in milliseconds. This adds to the period specified by *seconds*. 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*.

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).

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 optional argument *millisec* specifies an additional waiting period measured in milliseconds. This adds to the period specified by *seconds*. 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*.

Use `sleep-for` when you wish to guarantee a delay.

See Section 34.5 [Time of Day], page 570, for functions to get the current time.

18.8 Quitting

Typing `C-g` while the command loop has run a Lisp function 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, the result is 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` quit directly.

C-g causes a quit by setting the variable quit-flag to a non-nil value. 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, quits cannot happen just anywhere; only at the special places which 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 for a "critical section", where you wish to make sure that quitting does not happen within that part of the program.

In some functions (such as read-quoted-char), C-g is handled in a special way which 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 ((count 0) (code 0) char)
    (while (< count 3)
      (let ((inhibit-quit (zerop count))
            (help-form nil))
        (and prompt (message "%s-" prompt))
        (setq char (read-char))
        (if inhibit-quit (setq quit-flag nil)))
      ...)
    (logand 255 code)))
```

quit-flag Variable

If this variable is non-nil, then Emacs quits immediately, unless inhibit-quit is non-nil. Typing C-g 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 122.)

You can specify a character other than `C-g` to use for quitting. See the function `set-input-mode` in Section 34.7 [Terminal Input], page 572.

18.9 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 represented by a value that is always available (though it may be `nil`, meaning there is 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 `-`.

The various possibilities may be illustrated 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 `print-prefix` with various raw prefix arguments:

```
        M-x print-prefix  ⊣ nil

C-u     M-x print-prefix  ⊣ (4)

C-u C-u M-x print-prefix  ⊣ (16)

C-u 3   M-x print-prefix  ⊣ 3

M-3     M-x print-prefix  ⊣ 3       ; (Same as C-u 3.)

C-u -   M-x print-prefix  ⊣ -

M- -    M-x print-prefix  ⊣ -       ; (Same as C-u -.)

C-u -7  M-x print-prefix  ⊣ -7

M- -7   M-x print-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 18.3 [Interactive Call], page 260.) Alternatively, functions may look at the value of the prefix argument directly in the variable `current-prefix-arg`, but this is less clean.

Do not call the functions `universal-argument`, `digit-argument`, or `negative-argument` unless you intend to let the user enter the prefix argument for the *next* command.

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.

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 any other symbol, 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 is the value of the raw prefix argument for the *current* command. Commands may examine it directly, but the usual way to access it is with `(interactive "P")`.

prefix-arg — Variable

The value of this variable is the raw prefix argument for the *next* editing command. Commands that specify prefix arguments for the following command work by setting this variable.

18.10 Recursive Editing

The Emacs command loop is entered automatically when Emacs starts up. This top-level invocation of the command loop is never exited until the Emacs is killed. 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 finished. (The special commands for exiting are always available, but 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 119). 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 has a command to go back to the previous mode. (This technique is used by the `w` command in Rmail.) 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 18.8 [Quitting], page 280.

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.

18.11 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, that string is included in the message printed when the command is used:

```
(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. The user's '`.emacs`' file is optionally altered so that this will apply to future sessions.

disable-command *command* Command

Require special confirmation to execute *command* from now on. The user's '`.emacs`' file is optionally altered so that this will apply to future sessions.

disabled-command-hook Variable

This variable is a normal hook that is run instead of a disabled command, when the user runs the disabled command interactively. The hook functions can use `this-command-keys` to determine what the user typed to run the command, and thus find the command itself.

By default, `disabled-command-hook` contains a function that asks the user whether to proceed.

18.12 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-ESC` 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 all but the first (most recent) thirty elements are deleted when a garbage collection takes place (see Section B.3 [Garbage Collection], page 625).

```
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 17.4 [Minibuffer History], page 236), 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 history commands used are the same ones available in any minibuffer.

18.13 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. Don't confuse keyboard macros with Lisp macros (see Chapter 12 [Macros], page 165).

execute-kbd-macro *macro* &optional *count* Function

This function executes *macro* as a sequence of events. If *macro* 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 *macro* is a symbol, then its function definition is used in place of *macro*. 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; *macro* is executed that many times. If *count* is omitted or `nil`, *macro* is executed once. If it is 0, *macro* is executed over and over until it encounters an error or a failing search.

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`.

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.

defining-kbd-macro Variable

This variable indicates whether a keyboard macro is being defined. It is set to `t` by `start-kbd-macro`, and `nil` by `end-kbd-macro`. You can use this variable to make a command behave differently when run from a keyboard macro (perhaps indirectly by calling `interactive-p`). However, do not set this variable yourself.

The commands are described in the user's manual (see section "Keyboard Macros" in *The GNU Emacs Manual*).

19 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 with another keymap or with a command. When an event is bound to a keymap, that keymap is used to look up the next character typed; this continues until a command is found. The whole process is called *key lookup*.

19.1 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 ordinary ASCII characters, function keys, and mouse actions (see Section 18.5 [Input Events], page 264).

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 characters 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 19.5 [Prefix Keys], page 295, 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 a key sequence. In other words, removing one or more events from the end of any valid key must always yield a prefix key. For example, `C-f C-f` is not a key; `C-f` is not a prefix key, so a longer sequence starting with `C-f` cannot be a key.

Note that 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 19.7 [Active Keymaps], page 300, for details.

19.2 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.

An ordinary element is a cons cell of the form `(`*type*` . `*binding*`)`. This specifies one binding which applies to 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 18.5.9 [Classifying Events], page 271.

A cons cell whose CAR is `t` is 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.

If an element of a keymap is a vector, the vector counts as bindings for all the ASCII characters; vector element *n* is the binding for the character with code *n*. This is a more 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 every ASCII character even if the vector element is `nil`. Such a binding of `nil` overrides any default binding in the keymap. However, default bindings are still meaningful for events that are not 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.

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 19.6 [Menu Keymaps], page 296.

Keymaps do not directly record bindings for the meta characters, whose codes are from 128 to 255. 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`.

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. Practically speaking, this function tests for a list whose CAR is `keymap`.

```
(keymapp '(keymap))
    ⇒ t
(keymapp (current-global-map))
    ⇒ t
```

19.3 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 which contains 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 19.6 [Menu Keymaps], page 296).

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 which 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
```

19.4 Inheritance and Keymaps

A keymap can inherit the bindings of another keymap. Do do this, make a keymap whose "tail" is another existing keymap to inherit from. Such a keymap looks like this:

`(keymap` *bindings*... . *other-keymap*`)`

The effect is that this keymap inherits all the bindings of *other-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 *other-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 *other-keymap*.

Here is an example showing how to make a keymap that inherits from `text-mode-map`:

```
(setq my-mode-map (cons 'keymap text-mode-map))
```

19.5 Prefix Keys

A *prefix key* has an associated keymap which defines what to do with key sequences that start with the prefix key. For example, `C-x` is a prefix key, and it uses a keymap which is also stored in the variable `ctl-x-map`. Here is a list of the standard prefix keys of Emacs and their keymaps:

- `esc-map` is used for events that follow `ESC`. 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 used for events that follow `C-h`.
- `mode-specific-map` is for events that follow `C-c`. 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.
- `ctl-x-map` is the variable name for the map used for events that follow `C-x`. This map is also the function definition of `Control-X-prefix`.
- `ctl-x-4-map` is used for events that follow `C-x 4`.
- `ctl-x-5-map` used is for events that follow `C-x 5`.
- A nameless keymap is used for events that follow `C-x n`. Others are used for events following `C-x r` and `C-x a`.

The binding of a prefix key is the keymap to use for looking up the events that follow 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 definition is the keymap for `C-x` commands. (The same keymap is also the value of `ctl-x-map`.)

Prefix key definitions of this sort 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 19.7 [Active Keymaps], page 300.

If a key is defined as a prefix in more than one active map, then the 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 defines *symbol* as a prefix command: it creates a full keymap and stores it as *symbol*'s function definition. Storing the symbol as the binding of a key makes the key a prefix key which has a name. It also sets *symbol* as a variable, to have the keymap as its value. The function returns *symbol*.

In Emacs version 18, only the function definition of *symbol* was set, not the value as a variable.

19.6 Menu Keymaps

A keymap can define a menu as well as ordinary keys and mouse button meanings. Menus are normally actuated with the mouse, but they can work with the keyboard also.

19.6.1 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 19.2 [Format of Keymaps], page 292.) 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 19.3 [Creating Keymaps], page 293).

The individual bindings in the menu keymap should also have prompt strings; these strings become the items displayed in the menu. A binding with a prompt string looks like this:

(*string* . *real-binding*)

As far as `define-key` and `lookup-key` are concerned, the string is part of the event's binding. However, only *real-binding* is used for executing the key.

You can also supply a second string, called the help string, as follows:

(*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 hope to make *help-string* serve as extended documentation for the menu item, available on request.

The prompt string for a binding should be short—one or two words. It should describe the action of the command it corresponds to.

If *real-binding* is `nil`, then *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 which 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 with the mouse.

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.

19.6.2 Menus and the Mouse

The way to make a menu keymap produce a menu is to make it the definition of a prefix key.

When 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 which brought about that menu item.

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 a prompt string that starts with '`@`'. The rest of the prompt string becomes the name of the pane. See the file '`lisp/mouse.el`' for an example of this. Any ordinary bindings with '`@`'-less prompt strings are grouped into one pane, which appears along with the other panes explicitly created for the submaps.

You can also get multiple panes 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. See Section 19.7 [Active Keymaps], page 300.

A Lisp program can explicitly pop up a menu and receive the user's choice. You can use keymaps for this also. See Section 26.13 [Pop-Up Menus], page 432.

19.6.3 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 prompt 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.

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.

In a menu intended for keyboard use, each menu item must clearly indicate what character to type. The best convention to use is to make the character the first letter of the menu item prompt string. That is something users will understand without being told.

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`.

19.6.4 Menu Example

Here is a simple example of how to set up a menu for mouse use.

```
(defvar my-menu-map
  (make-sparse-keymap "Key Commands <==> Functions"))
(fset 'help-for-keys my-menu-map)

(define-key my-menu-map [bindings]
  '("List all keystroke commands" . describe-bindings))
(define-key my-menu-map [key]
  '("Describe key briefly" . describe-key-briefly))
(define-key my-menu-map [key-verbose]
  '("Describe key verbose" . describe-key))
```

```
(define-key my-menu-map [function]
  '("Describe Lisp function" . describe-function))
(define-key my-menu-map [where-is]
  '("Where is this command" . where-is))

(define-key global-map [C-S-down-mouse-1] 'help-for-keys)
```

The symbols used in the key sequences bound in the menu are fictitious "function keys"; they don't appear on the keyboard, but that doesn't stop you from using them in the menu. Their names were chosen to be mnemonic, because they show up in the output of `where-is` and `apropos` to identify the corresponding menu items.

However, if you want the menu to be usable from the keyboard as well, you must use real ASCII characters instead of fictitious function keys.

19.6.5 The Menu Bar

Under X Windows, each frame can 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 submenu containing all the subcommands of that item—the global subcommands, the local subcommands, and the minor mode subcommands, all together.

In order for a frame to display a menu bar, its `menu-bar-lines` property 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 you try one or two as the value of `menu-bar-lines`. See Section 26.2.3 [X Frame Parameters], page 424.

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)
;; to be the menu bar item's definition.
(define-key global-map [menu-bar words]
  (cons "Words" (make-sparse-keymap "Words")))

;; Make specific subcommands in the item's submenu.
```

```
(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)`; thus, the '`Help`' menu item normally appears at the end of the menu bar, following local menu items.

19.6.6 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, 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 key *after*. For example,

```
(define-key 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`.

19.7 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. At all times, the current buffer's local keymap is active. Text properties can specify an alternative local map for certain parts of the buffer; see Section 29.17.4 [Special Properties], page 497.

Each minor mode may have a keymap; if it does, the keymap is active whenever the minor mode is enabled.

All the active keymaps are used together to determine what command to execute when a key is entered. The key lookup proceeds as described earlier (see Section 19.8 [Key Lookup], page 302), but Emacs *first* searches for the key in the minor mode maps (one map at a time); 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.

Since every buffer that uses the same major mode normally uses the very same local keymap, it may appear as if the keymap is local to the mode. A change to the local keymap of a buffer (using `local-set-key`, for example) will be seen also in the other buffers that share that keymap.

The local keymaps that are used for Lisp mode, C mode, and several other major modes exist even if they have not yet been used. These local maps are the values of the variables `lisp-mode-map`, `c-mode-map`, and so on. For most other 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 Chapter 17 [Minibuffers], page 231.

See Appendix E [Standard Keymaps], page 643, for a list of standard keymaps.

global-map Variable

This variable contains the default global keymap that maps Emacs keyboard input to commands. Normally this keymap is the global keymap. The default global keymap is a full keymap that binds `self-insert-command` to all of the printing characters.

current-global-map Function

This function returns the current global keymap. This is always 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 current local keymap of the current buffer. If *keymap* is `nil`, then there will be no local keymap. It returns `nil`. Most major modes 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 which enables or disables a minor mode. See Section 20.2.2 [Keymaps and Minor Modes], page 326.

When more than one minor mode keymap is active, their order of priority is the order of `minor-mode-map-alist`.

See also `minor-mode-key-binding` in Section 19.9 [Functions for Key Lookup], page 304.

19.8 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 types 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 18.5 [Input Events], page 264. Such a pseudo-key-sequence is insufficient for `command-execute`, 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 directly associated with an event type in a keymap. While any Lisp object may be stored as a keymap entry, not all make sense for key lookup. Here is a list 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, that is equivalent to an entry of `nil` for that type.

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*.

command
: The events used so far in the lookup form a complete key, and *command* is its binding.

string
vector
: The events used so far in the lookup form a complete key, whose binding is a keyboard macro. See Section 18.13 [Keyboard Macros], page 288, for more information.

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 space) means, "Use the global binding of `Meta-SPC`, whatever that may be."
- If the CAR of *list* is a string, it serves as a menu item name if the keymap is used as a menu. For executing the key, the string is discarded and the CDR of *list* is used instead. (Any number of strings can be discarded from the front of the list in this way.) See Section 19.6 [Menu Keymaps], page 296.

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 which 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 also 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 18.3 [Interactive Call], page 260).

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 which 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)))
```

19.9 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*. If the string or vector *key* is not a valid key sequence according to the prefix keys specified in *keymap* (which means it is "too long" and has extra events at the end), 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 an element of *key* is `t`.

All the other functions described in this chapter that look up keys use `lookup-key`.

```
(lookup-key (current-global-map) "\C-x\C-f")
    ⇒ find-file
(lookup-key (current-global-map) "\C-x\C-f12345")
    ⇒ 2
```

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
```

This function does not modify the specified events in ways that discard information as `read-key-sequence` does (see Section 18.6.1 [Key Sequence Input], page 275). 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`.

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 the variable which 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 is a non-prefix, 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 19.5 [Prefix Keys], page 295). 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.

```
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!
```

19.10 Changing Key Bindings

The way to rebind a key is to change its entry in a keymap. You can change the global keymap, so that the change is effective in all buffers (except those that override the global binding with a local one). Or you can change the current buffer's local map, which usually affects all buffers using the same major mode. The `global-set-key` and `local-set-key` functions are convenient interfaces for these operations. Or you can use `define-key` and specify explicitly which map to change.

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)
```

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.

In writing the key sequence to rebind, it is useful to use the special escape sequences for control and meta characters (see Section 2.3.7 [String Type], page 26). 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`.

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. However, 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 19.8 [Key Lookup], page 302.) 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* may be defined as specified.

The following example creates a sparse keymap and makes a number of bindings:

```
(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* &optional *oldmap* — Function

This function replaces *olddef* with *newdef* for any keys in *keymap* that were bound to *olddef*. In other words, *olddef* is replaced with *newdef* wherever it appears. The function returns `nil`.

For example, this redefines `C-x C-f`, if you do it in an Emacs with standard bindings:

```
(substitute-key-definition
 'find-file 'find-file-read-only (current-global-map))
```

If *oldmap* is non-`nil`, then its bindings determine which keys to rebind. The rebindings still happen in *newmap*, 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 replacing the self-insertion commands for numbers with the `digit-argument` function, unless *nodigits* is non-`nil`, and by replacing the functions for the rest of the printing characters with `undefined`. This means that ordinary insertion of text is impossible in a buffer with a local keymap on which `suppress-keymap` has been called.

`suppress-keymap` returns `nil`.

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 24.6 [Read Only Buffers], page 389).

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)
...
```

19.11 Commands for Binding Keys

This section describes some convenient interactive interfaces for changing key bindings. They work by calling `define-key`.

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 which uses it implicitly 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)
```

19.12 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 accessed (via prefix keys) from *keymap*. The value is an association list with elements of the form `(key . map)`, where *key* is a prefix key whose definition in *keymap* is *map*.

The elements of the alist are ordered so that the *key* increases in length. The first element is always `("" . keymap)`, because the specified keymap is accessible from itself with a prefix of no events.

If *prefix* is given, it should be a prefix key sequence; then `accessible-keymaps` includes only the submaps whose prefixes start with *prefix*. These elements look just as they do in the value of `(accessible-keymaps)`; the only difference is that some elements are omitted.

In the example below, the returned alist indicates that the key `ESC`, which is displayed as '`^[`', is a prefix key whose definition is the sparse keymap `(keymap (83 . center-paragraph) (115 . foo))`.

```
(accessible-keymaps (current-local-map))
⇒(("" keymap
      (27 keymap    ; Note this keymap for ESC is repeated below.
          (83 . center-paragraph)
          (115 . center-line))
      (9 . tab-to-tab-stop))

   ("^[" keymap
    (83 . center-paragraph)
    (115 . foo)))
```

In the following example, `C-h` is a prefix key that uses a sparse keymap starting with `(keymap (118 . describe-variable)...)`. Another prefix, `C-x 4`, uses a keymap which happens to be `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 an actual case.

where-is-internal *command* &optional *keymap* *firstonly* Function

This function returns a list of key sequences (of any length) that are bound to *command* in *keymap* and the global keymap. The argument *command* can be any object; it is compared with all keymap entries using `eq`. If *keymap* is not supplied, then the global map alone is used.

If *firstonly* is non-`nil`, then the value is a single string representing the first key sequence found, rather than a list of all possible key sequences.

This function is used by `where-is` (see section "Help" in *The GNU Emacs Manual*).

```
(where-is-internal 'describe-function)
    ⇒ ("\^hf" "\^hd")
```

describe-bindings *prefix* Command

This function creates a listing of all defined keys, and their definitions. The listing is put in a buffer named '`*Help*`', which is then displayed in a window.

A meta character is shown as `ESC` followed by the corresponding non-meta character. Control characters are indicated with `C-`.

When several characters with consecutive ASCII codes have the same definition, they are shown together, as '*firstchar*..*lastchar*'. In this instance, you need to know the ASCII codes to understand which characters this means. For example, in the default global map, the characters '`SPC .. ~`' are described by a single line. `SPC` is ASCII 32, `~` is ASCII 126, and the characters between them include all the normal printing characters, (e.g., letters, digits, punctuation, etc.); all these characters are bound to `self-insert-command`.

If *prefix* is non-`nil`, it should be a prefix key; then only keys that start with *prefix* are described.

20 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 may be enabled individually.

This chapter covers both major and minor modes, the way they are indicated in the mode line, and how they run hooks supplied by the user. Related topics such as keymaps and syntax tables are covered in separate chapters. (See Chapter 19 [Keymaps], page 291, and Chapter 31 [Syntax Tables], page 523.)

20.1 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 `LFD` (`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 it for its new function. 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 is an example of a case where one piece of text is put temporarily into a different major mode so it can be edited in a different way (with ordinary Emacs commands rather than Rmail). 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 alternative major modes avoids this limitation. See Section 18.10 [Recursive Editing], page 285.

The standard GNU Emacs Lisp library directory contains the code for several major modes, in files including '`text-mode.el`', '`texinfo.el`', '`lisp-mode.el`', '`c-mode.el`', and '`rmail.el`'. You can look at these libraries to see how modes are written. Text mode is perhaps the simplest major mode aside from Fundamental mode. Rmail mode is a rather complicated, full-featured mode.

20.1.1 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 keep these conventions in mind when you create 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 local variables in an existing buffer without changing the buffer's text.
- Write a documentation string for this command which describes the special commands available in this mode. `C-h m` (`describe-mode`) will print this.

  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 21.3 [Keys in Documentation], page 339. The `describe-mode` function replaces these special documentation substrings with their current meanings. See Section 21.2 [Accessing Documentation], page 336.
- 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 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 [Style Tips], page 615.
- 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 function should call `use-local-map` to install this local map. See Section 19.7 [Active Keymaps], page 300, for more information.

  This keymap should be kept in a global variable named *modename*`-mode-map`. Normally the library that defines the mode sets this variable. Use `defvar` to set the variable, so that it is not reinitialized if it already has a value. (Such reinitialization could discard customizations made by the user.)

- 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`. The reasons for this are the same as for using a keymap variable. See Chapter 31 [Syntax Tables], page 523.
- 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 32.2 [Abbrev Tables], page 535.
- To give a variable a buffer-local binding, 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 10.9 [Buffer-Local Variables], page 143.
- If hooks are appropriate for the mode, the major mode command should run the hooks after completing all other initialization so the user may further customize any of the settings. See Section 20.4 [Hooks], page 332.
- 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 it is desirable that Emacs use the new mode by default after visiting 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 20.1.3 [Auto Major Mode], page 321.
- 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.

20.1.2 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" 'tab-to-tab-stop)
  (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)     ; This provides the local keymap.
  (setq mode-name "Text")           ; This name goes into the mode line.
  (setq major-mode 'text-mode)      ; This is how describe-mode
                                    ;   finds the doc string to print.
  (setq local-abbrev-table text-mode-abbrev-table)
  (set-syntax-table text-mode-syntax-table)
  (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)
  ;; The lisp-syntax argument is nil in Emacs Lisp mode,
  ;;   and t in the other two Lisp modes.
  (cond (lisp-syntax
         (if (not lisp-mode-syntax-table)
             ;; The Emacs Lisp mode syntax table always exists, but
             ;;   the Lisp Mode syntax table is created the first time a
             ;;   mode that needs it is called.  This is to save space.
```

```
              (progn (setq lisp-mode-syntax-table
                           (copy-syntax-table emacs-lisp-mode-syntax-table))
                     ;; Change some entries for Lisp mode.
                     (modify-syntax-entry ?\| "\"   "
                                          lisp-mode-syntax-table)
                     (modify-syntax-entry ?\[ "_   "
                                          lisp-mode-syntax-table)
                     (modify-syntax-entry ?\] "_   "
                                          lisp-mode-syntax-table)))
          (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 '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-l` to `run-lisp`, but the other Lisp modes do not. However, all Lisp modes have some commands in common. The following function adds these common commands to a given keymap.

```
(defun lisp-mode-commands (map)
  (define-key map "\e\C-q" 'indent-sexp)
  (define-key map "\177" 'backward-delete-char-untabify)
  (define-key map "\t" 'lisp-indent-line))
```

Here is an example of using `lisp-mode-commands` to initialize a keymap, as part of the code for Emacs Lisp mode. First we declare a variable with `defvar` to hold the mode-specific keymap. When this `defvar` executes, it sets the variable to `nil` if it was void. Then we set up the keymap if the variable is `nil`.

This code avoids changing the keymap or the variable if it is already set up. This lets the user customize the keymap if he or she so wishes.

```
(defvar emacs-lisp-mode-map () "")

(if emacs-lisp-mode-map
    ()
  (setq emacs-lisp-mode-map (make-sparse-keymap))
  (define-key emacs-lisp-mode-map "\e\C-x" 'eval-defun)
  (lisp-mode-commands emacs-lisp-mode-map))
```

Finally, here is the complete major mode function definition for Emacs Lisp mode.

```
(defun emacs-lisp-mode ()
  "Major mode for editing Lisp code to run in Emacs.
Commands:
Delete converts tabs to spaces as it moves back.
Blank lines separate paragraphs.  Semicolons start comments.
\\{emacs-lisp-mode-map}
Entry to this mode runs the hook `emacs-lisp-mode-hook'."
  (interactive)
  (kill-all-local-variables)
  (use-local-map emacs-lisp-mode-map)    ; This provides the local keymap.
  (set-syntax-table emacs-lisp-mode-syntax-table)
  (setq major-mode 'emacs-lisp-mode)     ; This is how describe-mode
                                         ;   finds out what to describe.
  (setq mode-name "Emacs-Lisp")          ; This goes into the mode line.
  (lisp-mode-variables nil)              ; This define various variables.
  (run-hooks 'emacs-lisp-mode-hook))     ; This permits the user to use a
                                         ;   hook to customize the mode.
```

20.1.3 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.

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, so it is not readily customizable.

normal-mode &optional *find-file* Command

This function establishes the proper major mode and 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, any 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. The variable `enable-local-variables` controls whether to do so.

If you run `normal-mode` yourself, the argument *find-file* is normally `nil`. In this case, `normal-mode` unconditionally processes any local variables list. See section "Local Variables in Files" in *The GNU Emacs Manual*, for the syntax of the local variables section of a file.

`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`.

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`), or on the value of a local variable). 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 the 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.

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 name. 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 30.2 [Regular Expressions], page 507) and corresponding major mode functions. Usually, the file name patterns test for suffixes, such as '`.el`' and '`.c`', but this need not be the case. Each 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 22.10.4 [File Name Expansion], page 367) matches a *regexp*, `set-auto-mode` calls the corresponding *mode-function*. This feature enables Emacs to select the proper major mode for most files.

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
   ;; Filename starts with a dot.
   '(("/\\.[^/]*$" . fundamental-mode)
     ;; Filename has no dot.
     ("[^\\./]*$" . fundamental-mode)
     ("\\.C$" . c++-mode))
   auto-mode-alist))
```

hack-local-variables &optional *force* Function

This function parses, and binds or evaluates as appropriate, any local variables for the current buffer.

The handling of `enable-local-variables` documented for `normal-mode` actually takes place here. The argument *force* reflects the argument *find-file* given to `normal-mode`.

20.1.4 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 21.2 [Accessing Documentation], page 336.)

major-mode Variable

This variable holds the symbol for the current buffer's major mode. This symbol should be the name of the function that is called to initialize the mode. The `describe-mode` function uses the documentation string of this symbol as the documentation of the major mode.

20.2 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 is unwieldy.

A minor mode is not usually a modification of single major mode. For example, Auto Fill mode may be used in 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 deactivate a minor mode and restore the environment of the major mode to the state it was in before the minor mode was activated.

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.

20.2.1 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 represent the minor mode. Its value should enable or disable the mode (`nil` to disable; anything else to enable.) We call this the *mode 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 `overwrite-mode`. It shows the use of `overwrite-mode` as a variable which enables or disables the mode's behavior.

  ```
  (setq overwrite-mode
        (if (null arg) (not overwrite-mode)
          (> (prefix-numeric-value arg) 0)))
  ```

- Add an element to `minor-mode-alist` for each minor mode (see Section 20.3.2 [Mode Line Variables], page 329). This element should be a list of the following form:

  (*mode-variable string*)

  Here *mode-variable* is the variable that controls enablement 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)))
  ```

20.2.2 Keymaps and Minor Modes

As of Emacs version 19, each minor mode can have its own keymap which is active when the mode is enabled. See Section 19.7 [Active Keymaps], page 300. To set up a keymap for a minor mode, add an element to the alist `minor-mode-map-alist`.

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. This is the only way to accomplish this in general, since there is no way to customize what `self-insert-command` does except in certain special cases (designed for abbrevs and Auto Fill mode). (Do not try substituting your own definition of `self-insert-command` for the standard one. The editor command loop handles this function specially.)

minor-mode-map-alist Variable

This variable is an alist of elements element that look like this:

`(`*variable* `.` *keymap*`)`

where *variable* is the variable which indicates whether the minor mode is enabled, and *keymap* is the keymap. The keymap *keymap* is active whenever *variable* has a non-`nil` value.

Note that elements of `minor-mode-map-alist` do not have the same structure as elements of `minor-mode-alist`. The map must be the CDR of the element; a list with the map as the second element will not do.

What's more, the keymap itself must appear in the CDR. It does not work to store a variable in the CDR and make the map the value of that variable.

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.

20.3 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 of the buffer.

This section describes how the contents of the mode line are controlled. It is in the chapter on modes 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 the mode lines will appear the same (except perhaps for the percentage of the file scrolled off the top).

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`, you may want to force an update of the mode line so as to display the new information.

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 35.11 [Inverse Video], page 594.

20.3.1 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.

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`. For most purposes, it is sufficient to alter the variables referenced by `mode-line-format`.

A mode line construct may be a list, cons cell, symbol, or string. If the value is a list, each element may be a list, a cons cell, 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 20.3.3 [%-Constructs], page 331.

symbol A symbol as a mode line construct stands for its value. The value of *symbol* is used in place of *symbol* unless *symbol* is `t` or `nil`, or is void, in which case *symbol* is ignored.

There is one exception: if the value of *symbol* is a string, it is processed verbatim in that the `%`-constructs are not recognized.

`(`*string rest*...`)` or `(`*list rest*...`)`
A list whose first element is a string or list, means to concatenate all the elements. 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 of the list (*then*) is processed recursively as a mode line element. But if the value of *symbol* is `nil`, the third element of the list (if there is one) is processed recursively.

`(`*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 all the same variables that are used by the default value, rather than duplicating their contents or displaying the information in another fashion. This permits customizations made by the user, by libraries (such as `display-time`) or by 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 hostname and default directory.

```
(setq mode-line-format
  (list ""
   'mode-line-modified
   "%b--"
```

```
(getenv "HOST")        ; One element is not constant.
":"
'default-directory
"   "
'global-mode-string
"   %[(" 'mode-name
'minor-mode-alist
"%n"
'mode-line-process
")%]----"
'(-3 . "%p")
"-%-"))
```

20.3.2 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-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, and '`--%%-`' if the buffer is read only.

Changing this variable does not force an update of the mode line.

mode-line-buffer-identification Variable

This variable identifies the buffer being displayed in the window. Its default value is '`Emacs: %17b`', which means that it displays '`Emacs:`' followed by the buffer name. You may want to change this in modes such as Rmail that do not behave like a "normal" Emacs.

global-mode-string Variable

This variable holds a string that is displayed in the mode line. The command `display-time` puts the time and load in this variable. The '`%M`' construct substitutes the value of `global-mode-string`, but this is obsolete, since the variable is included directly in the mode line.

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*)

The string *mode-line-string* is included 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
⇒ ((abbrev-mode " Abbrev")
    (overwrite-mode " Ovwrt")
    (auto-fill-function " Fill")
    (defining-kbd-macro " Def"))
```

(In earlier Emacs versions, `auto-fill-function` was called `auto-fill-hook`.)

`minor-mode-alist` is not buffer-local. The variables mentioned in the alist should be buffer-local if the 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:

```
("" 
 mode-line-modified
 mode-line-buffer-identification
 "   "
 global-mode-string
 "   %[("
 mode-name
 minor-mode-alist
 "%n"
 mode-line-process
 ")%]----"
 (-3 . "%p")
 "-%-")
```

20.3.3 %-Constructs in the Mode Line

The following table lists the recognized %-constructs and what they mean.

`%b` the current buffer name, using the `buffer-name` function.

`%f` the visited file name, using the `buffer-file-name` function.

`%*` '%' if the buffer is read only (see `buffer-read-only`);
'*' if the buffer is modified (see `buffer-modified-p`);
'-' otherwise.

`%s` the status of the subprocess belonging to the current buffer, using `process-status`.

`%p` the percent of the buffer above the top of window, or 'Top', 'Bottom' or 'All'.

`%n` 'Narrow' when narrowing is in effect; nothing otherwise (see `narrow-to-region` in Section 27.4 [Narrowing], page 449).

`%[` an indication of the depth of recursive editing levels (not counting minibuffer levels): one '[' for each editing level.

`%]` 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 are obsolete since use of the `mode-name` and `global-mode-string` variables will produce the same results.

`%m` the value of `mode-name`.

`%M` the value of `global-mode-string`. Currently, only `display-time` modifies the value of `global-mode-string`.

20.4 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 lots of 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 645, 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. The reason most hooks are normal hooks is so that you can use them in a uniform way. You can always tell when a hook is a normal hook, because its name ends in '`-hook`'.

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 149). Most normal hook variables are initially void; `add-hook` knows how to deal with this.

As for abnormal hooks, those whose names end in '`-function`' have a value which is a single function. Those whose names end in '`-hooks`' have a value which is a list of functions. Any hook which is abnormal is abnormal because a normal hook won't do the job; either the functions are called with arguments, or their values are meaningful. The name shows you that the hook is abnormal and you need to look up how to use it properly.

Most major modes run hooks as the last step of initialization. This makes it easy for a user to customize the behavior of the mode, by overriding the local variable assignments already made by the mode. But hooks may also be used in other contexts. For example, the hook `suspend-hook` runs just before Emacs suspends itself (see Section 34.2.2 [Suspending Emacs], page 566).

For example, you can put the following expression in your '`.emacs`' file if you want to turn on Auto Fill mode when in Lisp Interaction mode:

```
(add-hook 'lisp-interaction-mode-hook 'turn-on-auto-fill)
```

The next example shows how to use a hook to customize the way Emacs formats C code. (People often have strong personal preferences for one format compared to another.) Here the hook function is an anonymous lambda expression.

```
(add-hook 'c-mode-hook
  (function (lambda ()
              (setq c-indent-level 4
                    c-argdecl-indent 0
                    c-label-offset -4
```

```
                        c-continued-statement-indent 0
                        c-brace-offset 0
                        comment-column 40))))

(setq c++-mode-hook c-mode-hook)
```

Finally, here is an example of how to use the Text mode hook to provide a customized mode line for buffers in Text mode, displaying the default directory in addition to the standard components of the mode line. (This may cause the mode line to run out of space if you have very long file names or display the time and load.)

```
(add-hook 'text-mode-hook
  (function (lambda ()
              (setq mode-line-format
                    '(mode-line-modified
                      "Emacs: %14b"
                      "  "
                      default-directory
                      " "
                      global-mode-string
                      "%[("
                      mode-name
                      minor-mode-alist
                      "%n"
                      mode-line-process
                      ") %]---"
                      (-3 . "%p")
                      "-%-")))))
```

At the appropriate time, Emacs uses the `run-hooks` function to run particular hooks. This function calls the hook functions you have added with `add-hooks`.

run-hooks &rest *hookvar* Function

This function takes one or more hook names as arguments and runs each one 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.

For example:

```
(run-hooks 'emacs-lisp-mode-hook)
```

Major mode functions use this function to call any hooks defined by the user.

add-hook *hook function* &optional *append* — Function

This function is the handy way to add function *function* to hook variable *hook*. For example,

```
(add-hook 'text-mode-hook 'my-text-hook-function)
```

adds `my-text-hook-function` to the hook called `text-mode-hook`.

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.

21 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.

21.1 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`, print only the first line of a multi-line documentation string. Also, you should not indent the second line of a documentation string, if you have one, because that looks odd when you use `C-h f` (`describe-function`) or `C-h v` (`describe-variable`).

Documentation strings may 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 21.2 [Accessing Documentation], page 336.)

Within the Lisp world, a documentation string is kept with 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 150). The function `documentation` knows how to extract it.
- The documentation for a variable is stored on the variable's property list under the property name `variable-documentation`. The function `documentation-property` knows how to extract it.

However, to save space, the documentation for preloaded functions and variables (including primitive functions and autoloaded functions) are stored in the '`emacs/etc/DOC-`*version*' file. The '`emacs/etc/DOC-`*version*' file can be accessed by both the `documentation` and the `documentation-property` functions, and the process is transparent to the user. In this case, the documentation string is replaced with an integer offset into the '`emacs/etc/DOC-`*version*' file. Keeping the documentation strings out of the Emacs core image saves a significant amount of space. See Section B.1 [Building Emacs], page 623.

For information on the uses of documentation strings, see section "Help" in *The GNU Emacs Manual*.

The '`emacs/etc`' directory contains two utilities that you can use to print nice-looking hardcopy for the file '`emacs/etc/DOC-`*version*'. These are '`sorted-doc.c`' and '`digest-doc.c`'.

21.2 Access to Documentation Strings

documentation-property *symbol property* &optional *verbatim* Function

This function returns the documentation string that is recorded *symbol*'s property list under property *property*. This uses the function `get`, but does more than that: it also retrieves the string from the file '`emacs/etc/DOC-`*version*' if necessary, and runs `substitute-command-keys` to substitute the actual (current) key bindings.

If *verbatim* is non-`nil`, that inhibits running `substitute-command-keys`. (The *verbatim* argument exists only as of Emacs 19.)

```
(documentation-property 'command-line-processed
   'variable-documentation)
     ⇒ "t 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*. This function will access the documentation string if it is stored in the '`emacs/etc/DOC-`*version*' file.

In addition, `documentation` runs `substitute-command-keys` on the resulting string, so the value contains the actual (current) key bindings. (This is not done if *verbatim* is non-`nil`; the *verbatim* argument exists only as of Emacs 19.)

The function `documentation` signals a `void-function` error unless *function* has a function definition. However, *function* does not need to have a documentation string. If there is no documentation string, `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 C-x C-n, or nil.
set-goal-column Command: 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 623.

Emacs finds the file *filename* in the '`emacs/etc`' directory. When the dumped Emacs is later executed, the same file is found in the directory `data-directory`. Usually *filename* is `"DOC-`*version*`"`.

data-directory Variable

This variable holds the name of the directory in which Emacs finds certain data files that come with Emacs or are built as part of building Emacs. (In older Emacs versions, this directory was the same as `exec-directory`.)

21.3 Substituting Key Bindings in Documentation

This function makes it possible for you to write a documentation string that enables a user to display information about the current, actual key bindings. if you call `documentation` with non-`nil` *verbatim*, you might later call this function to do the substitution that you prevented `documentation` from doing.

substitute-command-keys *string* Function

This function returns *string* with certain special substrings replaced by the actual (current) key bindings. This permits the documentation to be displayed with accurate information about key bindings. (The key bindings may be changed by the user between the time Emacs is built and the time that the documentation is asked for.)

This table lists the forms of the special substrings and what they are replaced with:

`\[`*command*`]`
: is replaced either by a keystroke sequence that will invoke *command*, or by '`M-x` *command*' if *command* is not bound to any key sequence.

`\{`*mapvar*`}`
: is replaced by a summary of the value of *mapvar*, taken as a keymap. (The summary is made by `describe-bindings`.)

`\<`*mapvar*`>`
: makes this call to `substitute-command-keys` use the value of *mapvar* as the keymap for future '`\[`*command*`]`' substrings. This special string does not produce any replacement text itself; it only affects the replacements done later.

Please note: each '`\`' must be doubled when written in a string in Emacs Lisp.

Here are examples of the special substrings:

```
(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
LFD             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."
```

21.4 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 characters to sequences of printing characters. The description of a 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 18.5 [Input Events], page 264, 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 is represented by itself, but a control character turns into a string starting with '`C-`', a meta character turns into a string starting with '`M-`', and space, linefeed, etc. are transformed to '`SPC`', '`LFD`', etc. A function key is represented by its name. An event which is a list is represented by 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 LFD 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 leading 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"
```

21.5 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 predicate* Command

This function finds all symbols whose names contain a match for the regular expression *regexp*, and returns a list of them. It also displays the symbols in a buffer named '`*Help*`', each with a one-line description.

If *do-all* is non-`nil`, then `apropos` also shows key bindings for the functions that are found.

If *predicate* is non-`nil`, it should be a function to be called on each symbol that has matched *regexp*. Only symbols for which *predicate* returns a non-`nil` value are listed or displayed.

In the first of the following examples, `apropos` finds all the symbols with names containing '`exec`'. In the second example, it finds and returns only those symbols that are also commands. (We don't show 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)

(apropos "exec" nil 'commandp)
     ⇒ (Buffer-menu-execute execute-extended-command)
```

The command `C-h a` (`command-apropos`) calls `apropos`, but specifies a *predicate* to restrict the output to symbols that are commands. The call to `apropos` looks like this:

```
(apropos string t 'commandp)
```

super-apropos *regexp* &optional *do-all* — Command

This function differs from `apropos` in that it searches documentation strings as well as symbol names for matches for *regexp*. By default, it searches only the documentation strings, and only those of functions and variables that are included in Emacs when it is dumped. If *do-all* is non-`nil`, it scans the names and documentation strings of all functions and variables.

help-command — Command

This command is not a function, but rather a symbol which is equivalent to the keymap called `help-map`. It is defined in '`help.el`' as follows:

```
(define-key global-map "\C-h" 'help-command)
(fset 'help-command help-map)
```

help-map — Variable

The value of this variable is a local keymap for characters following the Help key, `C-h`.

print-help-return-message &optional *function* — Function

This function builds a string which is a message explaining 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 example in the section describing the `documentation` function (see Section 21.2 [Accessing Documentation], page 336).

The constructed message will have one of the forms shown below.

```
---------- Echo Area ----------
Type C-x 1 to remove help window.
---------- Echo Area ----------

---------- Echo Area ----------
Type C-x 4 b RET to restore old contents of help window.
---------- Echo Area ----------
```

help-char Variable

The value of this variable is the character that Emacs recognizes as meaning Help. When Emacs reads this character (which is usually 8, the value of `C-h`), Emacs evaluates `(eval help-form)`, and displays the result if it is a string. If `help-form`'s value is `nil`, this character is read normally.

help-form Variable

The value of this variable is a form to execute when the character `help-char` is read. If the form returns a string, that string is displayed. If `help-form` is `nil`, then the help character is not recognized.

Entry to the minibuffer binds this variable to the value of `minibuffer-help-form`.

prefix-help-command Variable

This variable holds a command that prints help for a prefix character. The command is run when the user types the help character after a prefix character. The default value of `prefix-help-command` is `describe-prefix-bindings`; that command uses `this-command-keys` to find what prefix character was used, then uses `describe-bindings` to describe it.

The following two functions are found in the library '`helper`'. They are for modes that want to provide help without relinquishing control, such as the "electric" modes. You must load that library with `(require 'helper)` in order to use them. 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`.

22 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 24 [Buffers], page 383, and those related to backups and auto-saving are described in Chapter 23 [Backups and Auto-Saving], page 373.

22.1 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 which 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.

22.1.1 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 24.3 [Buffer File Name], page 385, for functions and variables that access the visited file name of a buffer or that find an existing buffer by its visited file name.

find-file *filename* Command

This function reads the file *filename* into a buffer and displays that buffer in the selected window so that the user can edit it.

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 25.7 [Displaying Buffers], page 405.)

When `find-file` is called interactively, it prompts for *filename* in the minibuffer.

find-file-noselect *filename* Function

This function is the guts of all the file-visiting functions. It reads a file into a buffer and returns the buffer. You may then make the buffer current or display it in a window if you wish, but this function does not do so.

If no buffer is currently visiting *filename*, then one is created and the file is visited. If *filename* does not exist, the buffer is left empty, and `find-file-noselect` displays the message '`New file`' in the echo area.

If a buffer is already visiting *filename*, then the `find-file-noselect` function uses that buffer rather than creating a new one. However, it does verify 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.

The `find-file-noselect` function calls `after-find-file` after the file is read in (see Section 22.1.2 [Subroutines of Visiting], page 348). The `after-find-file` 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`.

The `find-file-noselect` function returns the buffer that is visiting the file *filename*.

```
(find-file-noselect "/etc/fstab")
     ⇒ #<buffer fstab>
```

find-alternate-file *filename* Command

This function reads the file *filename* into a buffer and selects it, killing the buffer current at the time the command is run. It is useful if you have visited the wrong file by mistake, so that you can get rid of the buffer that you did not want to create, at the same time as you visit the file you intended.

When this function is called interactively, it prompts for *filename*.

find-file-other-window *filename* Command

This function visits the file *filename* and displays its buffer in a window other than the selected window. It may use another existing window or split a window; see Section 25.7 [Displaying Buffers], page 405.

When this function is called interactively, it prompts for *filename*.

find-file-read-only *filename* Command

This function visits the file named *filename* and selects its buffer, just like `find-file`, but it marks the buffer as read-only. See Section 24.6 [Read Only Buffers], page 389, for related functions and variables.

When this function is called interactively, it prompts for *filename*.

view-file *filename* Command

This function views *filename* in View mode, returning to the previous buffer when done. View mode is a mode that allows you to skim rapidly through the file but does not let you modify it.

After loading the file, `view-file` runs the normal hook `view-hook` using `run-hooks`. See Section 20.4 [Hooks], page 332.

When this function 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 could be a normal hook, but we think that renaming it would not be advisable.

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 *filename*. These functions are called as soon as the error is detected. `buffer-file-name` is already set up. The functions are called in the order given, until one of them returns non-`nil`.

This is not a normal hook because the values of the functions are used and they may not all be run.

22.1.2 Subroutines of Visiting

The `find-file-noselect` function uses the `create-file-buffer` and `after-find-file` functions as subroutines. Sometimes it is useful to call them directly.

create-file-buffer *filename* — Function

This function creates a suitably named buffer for visiting *filename*, and returns it. The string *filename* (sans directory) is used unchanged if that name is free; otherwise, a string such as ‘<2>’ is appended to get an unused name. See also Section 24.8 [Creating Buffers], page 391.

Please note: `create-file-buffer` does *not* associate the new buffer with a file and does not make it the current buffer.

```
(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 24.8 [Creating Buffers], page 391).

after-find-file &optional *error warn* — Function

This function is called by `find-file-noselect` and by the default revert function (see Section 23.3 [Reverting], page 381). It sets the buffer major mode, and parses local variables (see Section 20.1.3 [Auto Major Mode], page 321).

If there was an error in opening the file, the calling function should pass *error* a non-`nil` value. In that case, `after-find-file` issues a warning: ‘`(New File)`’. Note that, for serious errors, you would not even call `after-find-file`. Only “file not found” errors get here with a non-`nil` *error*.

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 `find-file-hooks`.

22.2 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 or if the buffer was previously modified. 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` 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.

buffer-offer-save Variable

When this variable is non-`nil` in a buffer, Emacs offers to save the buffer on exit even if the buffer is not visiting a file. The variable is automatically local in all buffers. Normally, Mail mode (used for editing outgoing mail) sets this to `t`.

write-file *filename* Command

This function writes the current buffer into file *filename*, makes the buffer visit that file, and marks it not modified. The buffer is renamed to correspond to *filename* unless that name is already in use.

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 `basic-save-buffer` does when it writes a file in the usual way.

Here is an example showing how to add an element to `write-file-hooks` but avoid adding it twice:

```
(or (memq 'my-write-file-hook write-file-hooks)
    (setq write-file-hooks
          (cons
          'my-write-file-hook write-file-hooks)))
```

local-write-file-hooks Variable

This works just like `write-file-hooks`, but it is intended to be made local to particular buffers. It's not a good idea to make `write-file-hooks` local to a buffer—use this variable instead.

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 to be used for hooks that pertain to the contents of the file, as opposed to hooks that pertain to where the file came from.

after-save-hook Variable

This normal hook runs after a buffer has been saved in its visited 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.

(This feature worked differently in older Emacs versions.)

Some modes set this non-`nil` locally 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 Emacs silently puts 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 Emacs asks the user whether to add a newline each time the case arises.

If the value of the variable is `nil`, then Emacs doesn't add newlines at all. `nil` is the default value, but a few major modes set it to `t` in particular buffers.

22.3 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* 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.

If *visit* is non-`nil`, it also 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 should probably 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 you want to pass a file name to another process so that another program can read the file, see the function `file-local-copy` in Section 22.11 [Magic File Names], page 371.

22.4 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* Command

This function writes the region (of the current buffer) delimited by *start* and *end* 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 region is appended to the existing file contents (if any).

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 `write-file` and you should probably 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.

Normally, `write-region` displays a message '`Wrote file` *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 which the user does not need to know about.

22.5 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 do not work properly when multiple machines can share file systems, such as with NFS. Perhaps a better file locking system will be implemented in the future. 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 24.5 [Modification Time], page 388.

file-locked-p *filename* Function

This function returns `nil` if the file *filename* is not locked by this Emacs process. It returns `t` if it is locked by this Emacs, and it returns the name of the user who has locked it if it is locked by someone else.

```
(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 name *other-user*. The value it returns tells Emacs what to do next:

- A value of `t` tells Emacs to grab the lock on the file. Then this user may edit the file and *other-user* loses the lock.
- A value of `nil` tells Emacs 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 to the buffer which 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.

The default definition of this function asks the user to choose what to do. If you wish, you can replace the `ask-user-about-lock` function with your own version that decides in another way. The code for its usual definition is in '`userlock.el`'.

22.6 Information about Files

The functions described in this section are similar in as much as they all operate on strings which are interpreted as file names. All 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.

Most of the file-oriented functions take a single argument, *filename*, which must be a string. The file name is expanded using `expand-file-name`, so '`~`' is handled correctly, as are relative file names (including '`../`'). Environment variable substitutions, such as '`$HOME`', are not recognized by these functions. See Section 22.10.4 [File Name Expansion], page 367.

22.6.1 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 access files inside the directory.

file-writable-p *filename* Function

This function returns `t` if *filename* can be written or created by you. It is writable if the file exists and you can write it. It is creatable if the file does not exist, but the specified directory does exist and you can write in that directory. `file-writable-p` returns `nil` otherwise.

In the third example below, '`foo`' is not writable because the parent directory does not exist, even though the user could create it.

```
(file-writable-p "~rms/foo")
     ⇒ t
(file-writable-p "/foo")
     ⇒ nil
(file-writable-p "~rms/no-such-dir/foo")
     ⇒ nil
```

file-accessible-directory-p *dirname* Function

This function returns `t` if you have permission to open existing files in directory *dirname*; otherwise (and 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 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.

file-newer-than-file-p *filename1 filename2* Function

This functions 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`.

You can use `file-attributes` to get a file's last modification time as a list of two numbers. See Section 22.6.4 [File Attributes], page 357.

In the following example, assume that the file '`aug-19`' was written on the 19th, and '`aug-20`' was written on the 20th. 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
```

22.6.2 Distinguishing Kinds of Files

This section describes how to distinguish directories and symbolic links from ordinary files.

file-symlink-p *filename* Function

If *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 of no file at all.

If *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
```

22.6.3 Truenames

The *truename* of a file is the name that you get by following symbolic links until none remain, then expanding to get rid of '`.`' and '`..`' 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 24.3 [Buffer File Name], page 385, for related information.

22.6.4 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 22.9 [Changing File Attributes], page 361).
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 34.5 [Time of Day], page 570.)
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.
11. The file system number of the file system that the file is in. This element together with the file's inode number, 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. Unfortunately, you cannot convert this number into a time string in Emacs.

`(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.

22.7 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 value of switches 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 just the names relative to the specified directory.

If *match-regexp* is non-`nil`, this function returns only those file names that contain that regular expression—the other file names are discarded from the list.

If *nosort* is non-`nil`, that inhibits sorting 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 a directory listing for directory *dir*, formatted according to *switches*. It leaves point after the inserted text.

The argument *dir* 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 *file* is a directory and switches do not contain '`d`', so that a full listing is expected.

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`.

22.8 Creating and Deleting 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` in that case.

22.9 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*:

- A `file-already-exists` error is signaled if *ok-if-already-exists* is `nil`.
- Confirmation is requested if *ok-if-already-exists* is a number.
- No confirmation is requested if *ok-if-already-exists* is any other value, in which case the old file is removed.

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 -l fo*
-rw-rw-rw-  1 rms       29 Aug 18 20:32 foo
-rw-rw-rw-  1 rms       24 Aug 18 20:31 foo3
```

Then we evaluate the form `(add-name-to-file "~/lewis/foo" "~/lewis/foo2")`. Again we list the files. This shows two names, '`foo`' and '`foo2`'.

```
(add-name-to-file "~/lewis/foo1" "~/lewis/foo2")
     ⇒ nil

% ls -l fo*
-rw-rw-rw-  2 rms       29 Aug 18 20:32 foo
-rw-rw-rw-  2 rms       29 Aug 18 20:32 foo2
-rw-rw-rw-  1 rms       24 Aug 18 20:31 foo3
```

Finally, we evaluate the following:

```
(add-name-to-file "~/lewis/foo" "~/lewis/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 "~/lewis/foo1" "~/lewis/foo3")
     ⇒ nil

% ls -l fo*
-rw-rw-rw-  3 rms       29 Aug 18 20:32 foo
-rw-rw-rw-  3 rms       29 Aug 18 20:32 foo2
-rw-rw-rw-  3 rms       29 Aug 18 20:32 foo3
```

This function is meaningless on VMS, where multiple names for one file are not allowed.

See also `file-nlinks` in Section 22.6.4 [File Attributes], page 357.

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 functions gives the new file the same last-modified time that the old one has. (This works on only some operating systems.)

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. (In Unix, a file is deletable if its directory is writable.)

See also `delete-directory` in Section 22.8 [Create/Delete Dirs], page 361.

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, *filename* and *newname* are read in the minibuffer, and *ok-if-exists* is set to the numeric prefix argument.

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 12 low 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. Only the 9 low bits of *mode* are used.

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.

22.10 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 the 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.

22.10.1 File Name Components

The operating system groups files into directories. To specify a file, you must specify the directory, and the file's name in that directory. Therefore, a file name in Emacs is considered to have 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. Version numbers 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* without any file version numbers, backup version numbers, or trailing tildes.

```
(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"
```

22.10.2 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.

These two functions take a single argument, *filename*, which must be a string. Environment variable substitutions such as '`$HOME`', and the symbols '`~`', and '`..`', are *not* expanded. Use `expand-file-name` or `substitute-in-file-name` for that (see Section 22.10.4 [File Name Expansion], page 367).

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 that a slash is appended to the string. 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 a 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.

If you wish 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.

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"))
```

22.10.3 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 an *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
```

22.10.4 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 directory to start with if *filename* is relative. (The value of *directory* should itself be an absolute, expanded file name; it should not 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"
```

'`~/`' is expanded into the user's home directory. A '`/`' or '`~`' following a '`/`' is taken to be the start of an absolute file name that overrides what precedes it, so everything before that '`/`' or '`~`' is deleted. For example:

```
(expand-file-name
 "/a1/gnu//usr/local/lib/emacs/etc/MACHINES")
     ⇒ "/usr/local/lib/emacs/etc/MACHINES"
(expand-file-name "/a1/gnu/~/foo")
     ⇒ "/xcssun/users/rms/foo"
```

In both cases, '`/a1/gnu/`' is discarded because an absolute file name follows it.

Note that `expand-file-name` does *not* expand environment variables; that is done only by `substitute-in-file-name`.

file-relative-name *filename directory* Function

This function does the inverse of expansion—it tries to return a relative name which is equivalent to *filename* when interpreted relative to *directory*. (If such a relative name would be longer than the absolute name, it returns the absolute name instead.)

```
(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 is 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 names in *filename* with the values to which they are set by the operating system. 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"
```

If a '`~`' or a '`/`' appears following a '`/`', after substitution, 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"
```

On VMS, '`$`' substitution is not done, so this function does nothing on VMS except discard superfluous initial components as shown above.

22.10.5 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 (concat "/tmp/" name-of-application))
```

Here we use the directory '`/tmp/`' because that is the standard place on Unix for temporary files. The job of `make-temp-name` is to prevent two different users or two different jobs from trying to use the same name.

make-temp-name *string* Function

This function generates string that can be used as a unique name. The name starts with the prefix *string*, and ends with a number that is different in each Emacs job.

```
(make-temp-name "/tmp/foo")
     ⇒ "/tmp/foo021304"
```

To prevent conflicts among different application libraries run in the same Emacs, each application should have its own *string*. The number added to the end of the name distinguishes between the same application running in different Emacs jobs.

22.10.6 File Name Completion

This section describes low-level subroutines for completing a file name. For other completion functions, see Section 17.5 [Completion], page 237.

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 an absolute file name.

In the following example, suppose that the current default directory, '`~rms/lewis`', 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")
```

22.11 Making Certain File Names "Magic"

You can implement special handling for certain file names. This is called making those names *magic*. You must supply a regular expression to define the class of names (all those which match the regular expression), plus a handler that implements all the primitive Emacs file operations for file names that do match.

The value of `file-name-handler-alist` is a list of handlers, together with regular expressions that decide 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 you can handle for a magic file name:

`add-name-to-file`, `copy-file`, `delete-directory`, `delete-file`, `directory-file-name`, `directory-files`, `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-readable-p`, `file-symlink-p`, `file-writable-p`, `insert-directory`, `insert-file-contents`, `load`, `make-directory`, `make-symbolic-link`, `rename-file`, `set-file-modes`, `set-visited-file-modtime`, `unhandled-file-name-directory`, `verify-visited-file-modtime`, `write-region`.

The handler function must handle all of the above operations, and possibly others to be added in the future. Therefore, it should always reinvoke the ordinary Lisp primitive when it receives 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 (file-name-handler-alist)
             (apply operation args)))))
```

find-file-name-handler *file* — Function

This function returns the handler function for file name *file*, or `nil` if there is none.

file-local-copy *filename* — Function

This function copies file *filename* to the local site, if it isn't there 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. Otherwise, it asks the handler what to do.

This is used for running a subprocess; any subprocess must have a non-magic directory to serve as its current directory.

23 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.

23.1 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.

23.1.1 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 backup files 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.

The following example shows how to change the `make-backup-files` variable only in the '`RMAIL`' buffer and not elsewhere. Setting it `nil` stops Emacs from making backups of the '`RMAIL`' file, 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 filename Variable

This variable's value is a function to be called on certain occasions to decide whether a there should be backup files for file name *filename*. If it returns `nil`, backups are disabled. Otherwise, backups are enabled (if `make-backup-files` is true).

23.1.2 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 still refer to the current 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 22.2 [Saving Buffers], page 348.

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 Variable

This variable controls whether to make backup files by copying. If it is non-`nil`, then Emacs always copies the current contents of the file into the backup file before writing the buffer to be saved to the file. (In many circumstances, this has the same effect as `file-precious-flag`.)

backup-by-copying-when-linked Variable

This variable controls whether to make backups by copying for files with multiple names (hard links). If it is non-`nil`, then Emacs uses copying to create backups for those files.

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

This variable controls whether to make backups by copying in cases where renaming would change either the owner or the group of the file. If it is non-`nil` then Emacs creates backups by copying in such cases.

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`.

23.1.3 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
Do 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.

kept-new-versions User Option

The value of this variable is the number of oldest 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.

dired-kept-versions User Option

This variable plays a role in Dired's `dired-clean-directory` (`.`) command like that played by `kept-old-versions` when a backup file 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 deleted. The function `find-backup-file-name` (see Section 23.1.4 [Backup Names], page 376) is responsible for determining which backup versions to delete, but does not delete them itself.

trim-versions-without-asking 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.

23.1.4 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 which 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 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. In the following example, `make-backup-file-name` is redefined to prepend a '`.`' as well as to append a tilde.

```
(defun make-backup-file-name (filename)
  (concat "." filename "~"))

(make-backup-file-name "backups.texi")
     ⇒ ".backups.texi~"
```

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 old backup versions should be kept (by excluding them from the list of backup files ripe for deletion). See Section 23.1.3 [Numbered Backups], page 375.

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` that file has no backup files.

Some file comparison commands use this function in order to compare a file by default with its most recent backup.

23.2 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`; that should be checked before this function is called.

```
(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 which 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* Command

This function auto-saves all buffers that need to be auto-saved. This is all buffers for which auto-saving is enabled and that have been changed since the last time they were auto-saved.

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.

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 on 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.

23.3 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 *check-auto-save 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.

If the argument *check-auto-save* is non-`nil`, and the latest auto-save file is more recent than the visited file, `revert-buffer` asks the user whether to use that instead. Otherwise, it always uses the text of the visited file itself. Interactively, *check-auto-save* is set if there is a numeric prefix argument.

When the value of the *noconfirm* argument is non-`nil`, `revert-buffer` does not ask for confirmation for the reversion action. This means that the buffer contents are deleted and replaced by the text from the file on the disk, with no further opportunities for the user to prevent it.

Since reverting works by deleting the entire text of the buffer and inserting the file contents, all the buffer's markers are relocated to point at the beginning of the buffer. This is not "correct", but then, there is no way to determine what would be correct. It is not possible to determine, from the text before and after, which characters after reversion correspond to which characters before.

If the value of the `revert-buffer-function` variable is non-`nil`, it is called as a function with no arguments to do the work.

revert-buffer-function Variable

The value of this variable is the function to use to revert this buffer; but if the value of this variable is `nil`, then the `revert-buffer` function carries out its default action. 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 contents when reverting this buffer. The function receives two arguments, first the file name to use, and second, `t` if the user has asked to read the auto-save file.

recover-file *filename* Command

This function visits *filename*, but gets the contents from its last auto-save file. This is useful after the system has crashed, to resume editing the same file without losing all the work done in the previous session.

An error is signaled if there is no auto-save file for *filename*, or if *filename* is newer than its auto-save file. If *filename* does not exist, but its auto-save file does, then the auto-save file is read as usual. This last situation may occur if you visited a nonexistent file and never actually saved it.

24 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 which 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.

24.1 Buffer Basics

Buffers in Emacs editing are objects which have distinct names and hold text that can be edited. Buffers appear to Lisp programs as a special data type. The contents of a buffer may be viewed as an extendable string; insertions and deletions may occur in any part of the buffer. See Chapter 29 [Text], page 463.

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 only accessible through special-purpose functions. For example, the width of a tab character 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 10.9 [Buffer-Local Variables], page 143.

For functions and variables related to visiting files in buffers, see Section 22.1 [Visiting Files], page 345 and Section 22.2 [Saving Buffers], page 348. For functions and variables related to the display of buffers in windows, see Section 25.6 [Buffers and Windows], page 404.

bufferp *object* Function

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

24.2 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* is required to be an actual buffer object, not a name.

Buffers that are ephemeral and generally uninteresting to the user have names starting with a space, which prevents them from being listed by the `list-buffers` or `buffer-menu` commands. (A name starting with space also initially disables recording undo information; see Section 29.9 [Undo], page 476.)

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 24.9 [Killing Buffers], page 392.

```
(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 `nil`.

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 24.8 [Creating Buffers], page 391.

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 24.8 [Creating Buffers], page 391.

24.3 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 22.1 [Visiting Files], page 345.

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. See the definition of `set-visited-file-name` in '`files.el`'; 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 22.6.3 [Truenames], page 356.

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`. See Section 22.6.3 [Truenames], page 356.

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 22.6.4 [File Attributes], page 357, 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 22.10.4 [File Name Expansion], page 367), 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* 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.

When the function `set-visited-file-name` is called interactively, it prompts for *filename* in the minibuffer.

See also `clear-visited-file-modtime` and `verify-visited-file-modtime` in Section 24.4 [Buffer Modification], page 387.

list-buffers-directory Variable

This buffer-local variable records a string to display in a buffer listing in place of the visited file name, for buffers that don't have a visited file name. Dired buffers use this variable.

24.4 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 20.3.2 [Mode Line Variables], page 329), and controls saving (see Section 22.2 [Saving Buffers], page 348) and auto-saving (see Section 23.2 [Auto-Saving], page 378).

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 29 [Text], page 463.

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. 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.

24.5 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 Emacs's record of the modification time for the file that the buffer is visiting 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.

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 is the same format that `file-attributes` uses to return time values; see Section 22.6.4 [File Attributes], page 357.)

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.

visited-file-modtime Function

This function returns the buffer's recorded last file modification time, as a list of the form `(`*high* `.` *low*`)`. Note that this is not identical to the last modification time of the file that is visited (though under normal circumstances the values are equal).

ask-user-about-supersession-threat *fn* Function

This function is used to ask a user how to proceed after an attempt to modify an obsolete buffer. 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.

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.

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 `(`*fn*`)`, in which case the proposed buffer modification is not allowed.

See also the file locking mechanism in Section 22.5 [File Locks], page 352.

24.6 Read-Only Buffers

A buffer may be designated as *read-only*. This means that the buffer's contents may not be modified, although you may change your view of the contents by scrolling, narrowing, or widening, etc.

Read-only buffers are used in two kinds of situations:

- A buffer visiting a file is made read-only if the file is write-protected.

  Here, the purpose is to show 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 the function `toggle-read-only`.

- 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 the mode in question bind `buffer-read-only` to `nil` (with `let`) around the places where they 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. The value of `buffer-read-only` does not matter when `inhibit-read-only` is non-`nil`.

If `inhibit-read-only` is `t`, all `read-only` text properties have no effect (see Section 29.17.4 [Special Properties], page 497). If `inhibit-read-only` is a list, then `read-only` text properties are ignored 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 18.3 [Interactive Call], page 260, for another way to signal an error if the current buffer is read-only.

24.7 The Buffer List

The *buffer list* is a list of all buffers that have not been killed. The order of the buffers in the list is based primarily on how recently each buffer has been displayed in the selected window. Several functions, notably `other-buffer`, make use of this ordering.

buffer-list Function

This function returns a list of all buffers, including those whose names begin with a space. The elements are actual buffers, not their names.

```
(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")
```

Buffers appear earlier in the list if they were current more recently.

This list is a copy of a list used inside Emacs; modifying it has no effect on the buffers.

other-buffer &optional *buffer-or-name visible-ok* Function

This function returns the first buffer in the buffer list other than *buffer-or-name*. Usually this is the buffer most recently shown in the selected window, aside from *buffer-or-name*. Buffers are moved to the front of the list when they are selected and to the end when they are buried. Buffers whose names start with a space are not even considered.

If *buffer-or-name* is not supplied (or if it is not a buffer), then `other-buffer` returns the first buffer on the buffer list that is not visible in any window in a visible frame.

Normally, `other-buffer` avoids returning a buffer visible in any window on any visible frame, except as a last resort. However, if *visible-ok* is non-`nil`, then a buffer displayed in some window is admissible to return.

If no suitable buffer exists, the buffer '`*scratch*`' is returned (and created, if necessary).

list-buffers &optional *files-only* Command

This function displays a listing of the names of existing buffers. It clears the buffer '`*Buffer List*`', then inserts the listing into that buffer and displays it in a window. `list-buffers` is intended for interactive use, and is described fully in *The GNU Emacs Manual*. It returns `nil`.

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, and appears last in the list displayed by `list-buffers`.

If *buffer-or-name* is `nil` or omitted, this means to bury the current buffer. In addition, this switches to some other buffer (obtained using `other-buffer`) in the selected window. If the buffer is displayed in a window other than the selected one, it remains there.

If you wish to remove a buffer from all the windows that display it, you can do so with a loop that uses `get-buffer-window`. See Section 25.6 [Buffers and Windows], page 404.

24.8 Creating Buffers

This section describes the two primitives for creating buffers. `get-buffer-create` creates a buffer if it finds no existing buffer; `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 35.7 [Temporary Displays], page 586) and `create-file-buffer` (see Section 22.1 [Visiting Files], page 345).

get-buffer-create *name* Function

This function returns a buffer named *name*. If such a buffer already exists, it is returned. If such a buffer does not exist, one is created and returned. 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 by the value of `default-major-mode`. See Section 20.1.3 [Auto Major Mode], page 321.

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*>' are added 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 by the value of `default-major-mode`. See Section 20.1.3 [Auto Major Mode], page 321.

See the related function `generate-new-buffer-name` in Section 24.2 [Buffer Names], page 383.

24.9 Killing Buffers

Killing a buffer makes its name unknown to Emacs and makes its space available for other use.

The buffer object for the buffer which 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 24.10 [Current Buffer], page 394.

The `buffer-name` of a killed buffer is `nil`. You can use this feature to test whether a buffer has been killed:

```
(defun killed-buffer-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 use as space for other buffers. (Emacs version 18 and older was unable to return the memory 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 usual meaning of `SIGHUP` is that a dialup line has been disconnected.) See Section 33.4 [Deleting Processes], page 547.

If the buffer is visiting a file when `kill-buffer` is called and the buffer has not been saved since it was last modified, the user is asked to confirm before the buffer is killed. This is done even if `kill-buffer` is not called interactively. To prevent the request for confirmation, clear the modified flag before calling `kill-buffer`. See Section 24.4 [Buffer Modification], page 387.

Just before actually killing the buffer, after asking all questions, `kill-buffer` runs the normal hook `kill-buffer-hook`. The buffer to be killed is current when the hook functions run. See Section 20.4 [Hooks], page 332.

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
```

24.10 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 29 [Text], page 463). 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 designate any buffer as current temporarily 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 one to which the command will apply. (See Chapter 18 [Command Loop], page 255.) Therefore, `set-buffer` is not usable for switching visibly to a different buffer so that the user can edit it. For this, you must use the functions described in Section 25.7 [Displaying Buffers], page 405.

However, Lisp functions that change to a different current buffer should not leave it to 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-excursion` that will restore the current buffer when your program is done (see Section 27.3 [Excursions], page 448). 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-excursion
      (set-buffer (get-buffer-create buffer))
      (insert-buffer-substring oldbuf start end))))
```

This function binds a local variable to the current buffer, and then `save-excursion` records the values of point, the mark, and the original buffer. Next, `set-buffer` makes another buffer current. Finally, `insert-buffer-substring` copies the string from the original current buffer to the new current buffer.

If the buffer appended to happens to be displayed in some window, then 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.

Changing the current buffer between the binding and unbinding of a buffer-local variable can cause it to be bound in one buffer, and then unbound in another! You can avoid this problem by using save-excursion to make sure that the buffer from which the variable was bound is current again whenever the variable is unbound.

```
(let (buffer-read-only)
  (save-excursion
    (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. However, it does not display the buffer in the currently selected window or in any other window. This means that 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.

25 Windows

This chapter describes most of the functions and variables related to Emacs windows. See Chapter 35 [Display], page 581, for information on how text is displayed in windows.

25.1 Basic Concepts of Emacs Windows

A *window* is the physical area of the screen in which a buffer is displayed. The term is also used to refer to a Lisp object which represents that screen area in Emacs Lisp. It should be clear from the context which is meant.

There is always at least one window displayed on the screen, and there is exactly one window that we call the *selected window*. The cursor is in the selected window. The selected window's buffer is usually the current buffer (except when `set-buffer` has been used.) See Section 24.10 [Current Buffer], page 394.

For all intents, a window only exists while it is displayed on the terminal. Once removed from the display, 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 25.3 [Deleting Windows], page 400.)

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
- the amount of horizontal scrolling, in columns
- point
- the mark
- how recently the window was selected

Applications use multiple windows for a variety of reasons, but most often to give different views of the same 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 term "window" in Emacs means something similar to what it means in the context of general puprpose window systems such as X, but not identical. The X Window System subdivides the screen into X windows; Emacs uses one or more X windows, called *frames* in Emacs terminology, and subdivides each of them into (nonoverlapping) Emacs windows. When you use Emacs on an ordinary display terminal, Emacs subdivides the terminal screen into Emacs windows.

Most window systems support arbitrarily located overlapping windows. In contrast, Emacs windows are *tiled*; they never overlap, and together they fill the whole of the screen or frame. Because of the way in which Emacs creates new windows and resizes them, you can't create every conceivable tiling on an Emacs screen. See Section 25.2 [Splitting Windows], page 398. Also, see Section 25.13 [Size of Window], page 416.

See Chapter 35 [Display], page 581, 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.

25.2 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 25.7 [Displaying Buffers], page 405).

The functions described here do not accept a buffer as an argument. They let the two "halves" of the split window display the same buffer previously visible in the window that was split.

one-window-p &optional *no-mini* 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.

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 side by side, keeping the leftmost *size* columns and giving the rest of the columns to the new window. Otherwise, it splits into halves one above the other, keeping the upper *size* lines and giving the rest of the lines to the new window. The original window is therefore the right-hand or upper of the two, and the new window is the left-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:

```
          column 35
             __________
            |   |      |  line 0
            | w |  w3  |
            |___|______|
            |          |  line 15
            |    w2    |
            |__________|
                          line 50
     column 0   column 80
```

split-window-vertically *size* — Command

This function splits the selected window into two windows, one above the other, leaving the selected window with *size* lines.

This function is simply an interface to `split-windows`. Here is the complete function definition for it:

```
(defun split-window-vertically (&optional arg)
  "Split selected window into two windows,
one above the other..."
  (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-windows`. 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))
```

25.3 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 25.16 [Window Configurations], page 420). 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. (In Emacs version 18, the space was divided evenly among all the siblings.)

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 the display. 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.

This function returns `nil`.

When `delete-window` is called interactively, *window* defaults to the selected window.

delete-other-windows &optional *window* Command

This function makes *window* the only window on its frame, by deleting all the other windows. If *window* is omitted or `nil`, then the selected window is used by default.

The result is `nil`.

delete-windows-on *buffer* &optional *frame* Command

This function deletes all windows showing *buffer*. If there are no windows showing *buffer*, then this function does nothing. 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 the buffer chosen by `other-buffer`. See Section 24.7 [The Buffer List], page 390.

If there are several windows showing different buffers, then those showing *buffer* are removed, and the others are expanded to fill the void.

If *frame* is a frame, then `delete-windows-on` considers just the windows on *frame*. If *frame* is `nil`, all windows on all frames are considered. If *frame* is `t`, that stands for the selected frame.

This function always returns `nil`.

25.4 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>
```

The following functions choose one of the windows on the screen, offering various criteria for the choice.

get-lru-window &optional *all-frames* 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. The minibuffer window is not considered a candidate.

The argument *all-frames* controls which set of windows are considered. If it is non-`nil`, then all windows on all frames are considered. Otherwise, only windows in the selected frame are considered.

get-largest-window &optional *all-frames* 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. The minibuffer window is not considered a candidate.

If there are two windows of the same size, then the function returns the window which is first in the cyclic ordering of windows (see following section), starting from the selected window.

The argument *all-frames* controls which set of windows are considered. If it is non-`nil`, then all windows on all frames are considered. Otherwise, only windows in the selected frame are considered.

25.5 Cycling 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, 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 *window* &optional *minibuf all-frames* Function

This function returns the window following *window* in the cyclic ordering of windows. This is the window which `C-x o` would select if done when *window* is selected. If *window* is the only window visible, then this function returns *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`.

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 minibuffer window is active while the minibuffer is in use. See Chapter 17 [Minibuffers], page 231.)

When there are multiple frames, this functions normally cycles through all the windows in the selected frame, plus the minibuffer used by the selected frame even if it lies in some other frame.

If *all-frames* is `t`, then it cycles through all the windows in all the frames that currently exist.

If *all-frames* is neither `t` nor `nil`, then it cycles through precisely the windows in the selected frame, excluding the minibuffer in use if it lies in some other frame.

This example shows two windows, which both happen to be displaying the same buffer:

```
(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 *window* &optional *minibuf* *all-frames* — Function

This function returns the window preceding *window* in the cyclic ordering of windows. The other arguments affect which windows are included in the cycle, as in `next-window`.

other-window *count* — Command

This function selects the *count*th next window in the cyclic order. If count is negative, then it selects the $-$*count*th preceding window. 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 visible windows, calling `proc` once for each window with the window as its sole argument.

The optional argument *minibuf* says whether to include minibuffer windows. A value of `t` means count the minibuffer window even if not active. A value of `nil` means count it only if active. Any other value means not to count the minibuffer even if it is active.

If the optional third argument *all-frames* is `t`, that means include all windows in all frames. If *all-frames* is `nil`, it means to cycle within the selected frame, but include the minibuffer window (if *minibuf* says so) that that frame uses, even if it is on another frame. If *all-frames* is neither `nil` nor `t`, `walk-windows` sticks strictly to the selected frame.

25.6 Buffers and Windows

This section describes low-level functions to examine windows or to show 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`.

```
(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, then 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 25.5 [Cyclic Window Ordering], page 402.

The argument *all-frames* controls which set of windows are considered.

- If it is `nil`, then windows on the selected frame are considered.
- If it is a frame, then windows on that frame are considered.
- If it is `t`, then windows on all visible frames are considered.
- If it is some other non-`nil` value, then all windows on all frames are considered.

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`.

25.7 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.

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 is gratuitous and will surprise the user. Instead, use `set-buffer` (see Section 24.10 [Current Buffer], page 394) and `save-excursion` (see Section 27.3 [Excursions], page 448), 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 24.10 [Current Buffer], page 394.

If *buffer-or-name* does not identify an existing buffer, then a new buffer by that name is created.

Normally the specified buffer is put at the front of the buffer list. This affects the operation of `other-buffer`. However, if *norecord* is non-`nil`, this is not done. See Section 24.7 [The Buffer List], page 390.

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* — 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 previously selected window is absolutely never used to display the buffer. 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.

pop-to-buffer *buffer-or-name* &optional *other-window* 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.

If the variable `pop-up-frames` is non-`nil`, `pop-to-buffer` creates a new frame to display the buffer in. Otherwise, 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 25.8 [Choosing Window], page 407.

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.

If *buffer-or-name* is a string that does not name an existing buffer, a buffer by that name is created.

An example use of this function is found at the end of Section 33.8.2 [Filter Functions], page 553.

25.8 Choosing a Window

This section describes the basic facility which 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* Function

This function 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 that the specified buffer should be displayed 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*.

Precisely how `display-buffer` finds or creates a window depends on the variables described below.

A window can be marked as "dedicated" to its buffer. Then `display-buffer` does not try to use that window.

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 *flags* is non-`nil`, and nondedicated otherwise.

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 rather replaces its buffer.

split-height-threshold — User Option

This variable determines when `display-buffer` may split a window, if there are multiple windows. `display-buffer` splits the largest window if it has at least this many lines.

If there is only one window, it is split regardless of this value, provided `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` makes a new frame. If it is `nil`, then `display-buffer` either splits a window or reuses one.

If this is non-`nil`, the variables `pop-up-windows` and `split-height-threshold` do not matter.

See Chapter 26 [Frames], page 423, for more information.

pop-up-frame-function — Variable

This variable specifies how to make a new frame if `pop-up-frame` 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 which 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 26.2 [Frame Parameters], page 424, for more information about frame parameters.

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.

25.9 Window 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 to 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 27 [Positions], page 439, for more details on 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.

25.10 The Window Start Position

Each window contains a marker used to keep track of a buffer position which 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.

```
(window-start)
     ⇒ 7058
```

For a more complicated example of use, see the description of `count-lines` in Section 27.2.4 [Text Lines], page 443.

window-end &optional *window* Function

This function returns the position of the end of the display in window *window*. If *window* is `nil`, the selected window is used.

set-window-start *window position* &optional *noforce* Function

This function sets the display-start position of *window* to *position* in *window*'s buffer.

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 with `nil` for *noforce*, it means you want display to start at *position* even if that would put the location of point off the screen. What the display routines do in this case is move point instead, to the left margin on the middle line in the window.

For example, if point is 1 and you attempt to set the start of the window to 2, then the position of point would be "above" the top of the window. The display routines would 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)))

;; 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 ----------

     ⇒ 2
```

However, when *noforce* is non-`nil`, `set-window-start` does nothing if the specified start position would make point invisible.

This function returns *position*, regardless of whether the *noforce* option caused that position to be overruled.

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. It returns `t` if *position* is out of view only because *window* has been scrolled horizontally. See Section 25.12 [Horizontal Scrolling], page 414.

25.11 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`) will have unpredictable results if the current buffer is different from the buffer that is displayed in the selected window. See Section 24.10 [Current Buffer], page 394.

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.

`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`.

The window that is scrolled is normally the one following the selected window in the cyclic ordering of windows—the window that `next-window` would return. See Section 25.5 [Cyclic Window Ordering], page 402.

If the selected window is the minibuffer, the next window is normally the one at the top left corner. However, you can specify the window to scroll by binding the variable `minibuffer-scroll-window`. This variable has no effect when any other window is selected. See Section 17.8 [Minibuffer Misc], page 251.

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-step User Option

This variable controls how scrolling is done automatically when point moves off the screen. If the value is zero, then the text is scrolled so that point is centered vertically in the window. If the value is a positive integer *n*, then if it is possible to bring point back on screen by scrolling *n* lines in either direction, that is done; otherwise, point is centered vertically as usual. The default value is zero.

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, when `scroll-up` executes, this many lines that were visible at the bottom of the window move to the top of the window. 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`, then it puts the line containing point in the middle of the window, then clears and redisplays the entire selected frame.

When `recenter` is called interactively, Emacs sets *count* to 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.

Typing `C-u 0 C-l` positions the current line at the top of the window. This action is so handy that some people bind the command to a function key. 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 "\C-cl" 'line-to-top-of-window)
```

25.12 Horizontal Scrolling

Because we read English first from top to bottom and second from left to right, horizontal scrolling is not like vertical scrolling. Vertical scrolling involves selection of a contiguous portion of text to display. 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 it is not allowed. Scrolling to the left is allowed; it causes the first columns of text to go 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`.

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`.

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 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 to 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:

```
(save-excursion
  (goto-char position)
  (and
   (>= (- (current-column) (window-hscroll window)) 0)
   (< (- (current-column) (window-hscroll window))
      (window-width window))))
```

25.13 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. For a window that does not abut the right hand edge of the screen, the column of '`|`' characters that separates it from the window on the right is included in the width.

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 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.

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.

Here is the result obtained on a typical 24-line terminal with just one window:

```
(window-edges (selected-window))
     ⇒ (0 0 80 23)
```

If *window* is at the upper left corner of its frame, *right* and *bottom* are the same as the values returned by `(window-width)` and `(window-height)` respectively, and *top* and *bottom* are zero. For example, the edges 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`'.

```
           0
            _______
        0 |       |
          |       |
          |       |
          |       |
          xxxxxxxxx  4

                  7
```

When there are side-by-side windows, any window not at the right edge of its frame has a border in its last column. This border counts as one column in the width of the window. A window never includes a border on its left, since the border there belongs to the window to the left.

In the following example, let's imagine 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`'.

```
 --- ---
|   |   |
|   |   |
xxxxxxxxx

 0  34  7
```

25.14 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 bigger, 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, then that window disappears.

If *horizontal* is non-`nil`, then 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, then that window disappears.

If the window's frame is smaller than *size* lines (or columns), 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. If it becomes shorter than `window-min-height`, it disappears.

`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.

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))
```

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 disappears. A window disappears when it becomes smaller than `window-min-height`, and no window may be created that is smaller. The absolute minimum height is two (allowing one line for the mode line, and one line for the buffer display). Actions which 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 disappears. A window disappears when it becomes narrower than `window-min-width`, and no window may be created that is narrower. The absolute minimum width is one; any value below that is ignored. The default value is 10.

25.15 Coordinates and Windows

This section describes how to compare screen coordinates with 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 screen or frame.

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 this 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 of `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.

`nil`
: The coordinates are not in any sense within *window*.

The function `coordinates-in-window-p` does not require a frame as argument because it always uses the frame that window *window* is on.

25.16 Window Configurations

A *window configuration* records the entire layout of a 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 26.10 [Frame Configurations], page 431.

current-window-configuration Function

This function returns a new object representing Emacs's current window configuration, namely 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. 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 Emacs's windows and buffers to the state specified by *configuration*. The argument *configuration* must be a value that was previously returned by `current-window-configuration`.

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 executes *forms* in sequence, preserving window sizes and contents, including the value of point and the portion of the buffer which is visible. It also preserves the choice of selected window. However, it does not restore the value of point in the current buffer; use `save-excursion` for that.

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.

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.

26 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 has just one frame, a *terminal frame*. There is no way to create another terminal frame after startup. If Emacs has an X display, it does not make a terminal frame; instead, it initially creates a single *X window frame*. You can create more; see Section 26.1 [Creating Frames], page 423.

framep *object* Function

This predicate returns `t` if *object* is a frame, and `nil` otherwise.

See Chapter 35 [Display], page 581, for related information.

26.1 Creating Frames

To create a new frame, call the function `make-frame`.

make-frame *alist* Function

This function creates a new frame, if the display mechanism permits creation of frames. (An X server does; an ordinary terminal does not.)

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 there either default from the standard X defaults file and X resources.

The set of possible parameters depends in principle on what kind of window system Emacs uses to display its the frames. See Section 26.2.3 [X Frame Parameters], page 424, for documentation of individual parameters you can specify when creating an X window frame.

default-frame-alist Variable

An alist specifying default values of frame parameters. Each element has the form:

(*parameter* . *value*)

If you use options that specify window appearance when you invoke Emacs, they take effect by adding elements to `default-frame-alist`.

26.2 Frame Parameters

A frame has many parameters that control how it displays.

26.2.1 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 of all the parameters of *frame*.

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.

26.2.2 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 to when creating the initial X window frame.

If these parameters specify a separate minibuffer-only frame, 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 to when creating an initial minibuffer-only frame—if such a frame is needed, according to the parameters for the main initial frame.

26.2.3 X Window Frame Parameters

Just what parameters a frame has depends on what display mechanism it uses. Here is a table of the parameters of an X window frame:

`name` The name of the frame.

`left` The screen position of the left edge, in pixels.

`top` The screen position of the top edge, in pixels.

`height` The height of the frame contents, in pixels.

`width` The width of the frame contents, in pixels.

`window-id`
: The number of the X window for 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, a minibuffer window (in some other frame) means the new frame uses that minibuffer.

`font` The name of the font for text in the frame. This is a string.

`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 a scroll bar for vertical scrolling (non-`nil` means yes).

`horizontal-scroll-bars`
: Whether the frame has a scroll bar for horizontal scrolling (non-`nil` means yes). (Horizontal scroll bars are not currently implemented.)

`icon-type`
: The type of icon to use for this frame when it is iconified. Non-`nil` specifies a bitmap icon, `nil` a text icon.

`foreground-color`
: The color to use for the inside of a character. We use strings to designate colors; the X server defines the meaningful color names.

`background-color`
: The color to use for the background of text.

`mouse-color`
: The color for the mouse cursor.

`cursor-color`
: The color for the cursor that shows point.

`border-color`
: The color for the border of the frame.

`cursor-type`
: The way to display the cursor. There are two legitimate values: `bar` and `box`. The value `bar` specifies a vertical bar between characters as the cursor. The value `box` specifies an ordinary black box overlaying the character after point; that is the default.

`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 26.8 [Visibility of Frames], page 430.

`menu-bar-lines`
The number of lines to allocate at the top of the frame for a menu bar. The default is zero. See Section 19.6.5 [Menu Bar], page 299.

`parent-id`
The X Window number of the window that should be the parent of this one. Specifying this lets you create an Emacs window inside some other application's window. (It is not certain this will be implemented; try it and see if it works.)

26.2.4 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`. When you create a frame, you must specify either both size parameters or neither. Likewise, you must specify either both position parameters or neither. 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, counting from the top left corner of the screen.

frame-height &optional *frame* Function
frame-width &optional *frame* Function
These functions return the height and width of *frame*, measured in characters. If you don't supply *frame*, they use the selected frame.

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, respectively, 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 with values measured in pixels, use `modify-frame-parameters` to set the `width` and `height` parameters. See Section 26.2.3 [X Frame Parameters], page 424.

The old-fashioned functions `set-screen-height` and `set-screen-width`, which were used to specify the height and width of the screen in Emacs versions that did not support multiple frames, are still usable. They apply to the selected frame. See Section 35.2 [Screen Size], page 581.

x-parse-geometry *geom* Function

The function `x-parse-geometry` converts a standard X windows geometry string to an alist which you can use as part of the argument to `x-create-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`.

```
(x-geometry "35x70+0-0")
     ⇒ ((width . 35) (height . 70) (left . 0) (top . -1))
```

26.3 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 26.10 [Frame Configurations], page 431); 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.

26.4 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.

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.

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.

26.5 Frames and Windows

All the non-minibuffer windows in a frame are arranged in a tree of subdivisions; the root of this tree is available via the function `frame-root-window`. Each window is part of one and only one frame; you can get the frame with `window-frame`.

frame-root-window *frame* Function

This returns the root window of frame *frame*.

window-frame *window* Function

This function returns the frame that *window* is on.

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* which is selected within *frame*.

Conversely, selecting a window for Emacs with `select-window` also makes that window selected within its frame. See Section 25.4 [Selecting Windows], page 401.

26.6 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 17.8 [Minibuffer Misc], page 251).

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 frame on which to find the minibuffer to use. 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 which does have a minibuffer.

26.7 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.

The X server normally directs keyboard input to the X window that the mouse is in. Some window managers use mouse clicks or keyboard events to *shift the focus* to various X windows, overriding the normal behavior of the server.

Lisp programs can switch frames "temporarily" by calling the function `select-frame`. This does not override the window manager; rather, it escapes from the window manager's control until that control is somehow reasserted.

select-frame *frame* Function

This function selects frame *frame*, temporarily disregarding the X Windows focus. 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 X server and the window managers by arranging to select frames according to what the server and window manager ask for. It does so by generating a special kind of input event, called a *focus* event. The command loop handles a focus event by calling `internal-select-frame`. See Section 18.5.7 [Focus Events], page 270.

internal-select-frame *frame* Function

This function selects frame *frame*, assuming that the X server focus already points to *frame*.

Focus events normally do their job by invoking this command. Don't call it for any other reason.

26.8 Visibility of Frames

A 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 in the screen, not even as an icon.

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 26.2.3 [X Frame Parameters], page 424.

26.9 Raising and Lowering Frames

The X window system uses 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's X windows with these functions:

raise-frame *frame* Function

This function raises frame *frame*.

lower-frame *frame* Function

This function lowers frame *frame*.

You can also specify 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 26.2.3 [X Frame Parameters], page 424.

26.10 Frame Configurations

current-frame-configuration Function

This function returns a *frame configuration* list which describes the current arrangement of frames, all their properties, and the window configuration of each one.

set-frame-configuration *configuration* Function

This function restores the state of frames described in *configuration*.

26.11 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

Execute *body*, meanwhile generating input events for mouse motion. The code in *body* can read these events with `read-event` or `read-key-sequence`. See Section 18.5.6 [Motion Events], page 269, for the format of mouse motion events.

The value of `track-mouse` is that of the last form in *body*.

The usual purpose of tracking mouse motion is to indicate on the screen the consequences of pushing or releasing a button at the current position.

26.12 Mouse Position

The new 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 pixels relative to the top left corner of the inside of *frame*.

set-mouse-position *frame x y* Function

Thus function *warps the mouse* to position *x*, *y* in frame *frame*. The arguments *x* and *y* are integers, giving the position in pixels relative to the top left corner of the inside of *frame*.

Warping the mouse means changing the screen position of the mouse as if the user had moved the physical mouse—thus simulating the effect of actual mouse motion.

26.13 Pop-Up Menus

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 positions measured in characters, counting from the top left corner of *window*'s frame.

The argument *menu* says what to display in the menu. It can be a keymap or a list of keymaps (see Section 19.6 [Menu Keymaps], page 296). 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.

26.14 X 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), or a cons of two markers pointing to the same buffer. In the last case, the selection is considered to be the text between the markers. The 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 Windows conventions. The default is `PRIMARY`.

x-get-selection *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, kill-ring-style.

26.15 X Server

This section describes how to access and change the overall status of the X server Emacs is using.

26.15.1 X Connections

You can close the connection with the X server with the function `x-close-current-connection`, and open a new one with `x-open-connection` (perhaps with a different server and display).

x-close-current-connection Function

This function closes the connection to the X server. It deletes all frames, making Emacs effectively inaccessible to the user; therefore, a Lisp program that closes the connection should open another one.

x-open-connection *display* &optional *resource-string* Function

This function opens a connection to an X server, for use of display *display*.

The optional argument *resource-string* 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. Here's an example of what this string might look like:

```
"*BorderWidth: 3\n*InternalBorder: 2\n"
```

x-color-display-p Function

This returns `t` if the connected X display has color, and `nil` otherwise.

x-color-defined-p *color* Function

This function reports whether a color name is meaningful and supported on the X display Emacs is using. It returns `t` if the display supports that color; otherwise, `nil`.

Black-and-white displays support just two colors, `"black"` or `"white"`. Color displays support many other colors.

x-synchronize *flag* Function

The function `x-synchronize` enables or disables synchronous communication with the X server. It enables synchronous communication if *flag* is non-`nil`, and disables it if *flag* is `nil`.

In synchronous mode, Emacs waits for a response to each X protocol command before doing anything else. This is useful for debugging Emacs, because protocol errors are reported right away, which helps you find the erroneous command. Synchronous mode is not the default because it is much slower.

26.15.2 Resources

x-get-resource *attribute* &optional *name class* 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 uses '`Emacs`' 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`.*subclass*'.

26.15.3 Data about the X Server

This section describes functions and a variable that you can use to get information about the capabilities and origin of the X server that Emacs is displaying its frames on.

x-display-screens Function
: This function returns the number of screens associated with the current display.

x-server-version Function
: This function returns the list of version numbers of the X server in use.

x-server-vendor Function
: This function returns the vendor supporting the X server in use.

x-display-pixel-height Function
: This function returns the height of this X screen in pixels.

x-display-mm-height Function
: This function returns the height of this X screen in millimeters.

x-display-pixel-width Function
: This function returns the width of this X screen in pixels.

x-display-mm-width Function
: This function returns the width of this X screen in millimeters.

x-display-backing-store Function
: This function returns the backing store capability of this screen. Values can be the symbols `always`, `when-mapped`, or `not-useful`.

x-display-save-under Function
: This function returns non-`nil` if this X screen supports the Save-Under feature.

x-display-planes Function
: This function returns the number of planes this display supports.

x-display-visual-class Function
: This function returns the visual class for this X screen. The value is one of the symbols `static-gray`, `gray-scale`, `static-color`, `pseudo-color`, `true-color`, and `direct-color`.

x-display-color-p Function

This function returns `t` if the X screen in use is a color screen.

x-display-color-cells Function

This function returns the number of color cells this X screen supports.

x-no-window-manager Variable

This variable's value is is `t` if no X window manager is in use.

27 Positions

A *position* is the index of a character in the text of 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, the character after a position is often said to be "at" that position.

Positions are usually represented as integers starting from 1, but can also be represented as *markers*—special objects which relocate automatically when text is inserted or deleted so they stay with the surrounding characters. See Chapter 28 [Markers], page 453.

27.1 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. Many terminals display the cursor over the character that immediately follows point; on such terminals, 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 27.4 [Narrowing], page 449), 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 25.9 [Window Point], page 409, for more details.

point Function

This function returns the position 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 1, unless narrowing is in effect, in which case it is the position of the start of the region that you narrowed to. (See Section 27.4 [Narrowing], page 449.)

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 27.4 [Narrowing], page 449).

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 27.4 [Narrowing], page 449), `point-max` returns a value one larger than this.

```
(buffer-size)
     ⇒ 35
(point-max)
     ⇒ 36
```

buffer-saved-size Variable

The value of this buffer-local variable is the former length of the current buffer, as of the last time it was read in, saved or auto-saved.

27.2 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 27.1 [Point], page 439.

27.2.1 Motion by Characters

These functions move point based on a count of characters. `goto-char` is a fundamental primitive because it is the way to move point to a specified position.

goto-char *position* Command

This function sets point in the current buffer to the value *position*. If *position* is less than 1, then point is set to the beginning of the buffer. If it is greater than the length of the buffer, then point is set to the end of the buffer.

If narrowing is in effect, then the position is still measured from the beginning of the buffer, but point cannot be moved outside of the accessible portion. Therefore, if *position* is too small, point is set to the beginning of the accessible portion of the text; if *position* is too large, point is set to the end.

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 forward, towards the end of the buffer, *count* characters (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 backward, towards the beginning of the buffer, *count* characters (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.

27.2.2 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 31 [Syntax Tables], page 523.

forward-word *count* Command

This function moves point forward *count* words (or backward if *count* is negative). Normally it returns `t`. If this motion encounters the beginning or end of the buffer, or the limits of the accessible portion when narrowing is in effect, point stops there and the value is `nil`.

In an interactive call, *count* is set to the numeric prefix argument.

backward-word *count* Command

This function 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 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.

27.2.3 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 which 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`.

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.

In an interactive call, *n* is the numeric prefix argument, if provided; otherwise *n* defaults to `nil`.

Don't use this function in Lisp programs!

27.2.4 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, 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 buffer. If *line* is less than 1, then point is set to the beginning of the buffer. If *line* is greater than the number of lines in the buffer, then point is set to the *end of the last line* of the buffer.

If narrowing is in effect, then *line* still counts from the beginning of the buffer, but point cannot go outside the accessible portion. So point is set at the beginning or end of the accessible portion of the text if the line number specifies a position that is inaccessible.

The return value of `goto-line` is the difference between *line* and the line number of the line to which point actually was able move (before taking account of any narrowing). Thus, the value is positive if the scan encounters the 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 at the beginning of the last line. 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 at the end of the last line. 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 the line.

If the beginning or end of the buffer (or of the accessible portion) is encountered before that many lines are found, then point stops at the beginning or end. 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 will positioned 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
in the selected window.  Top line is 0.
Counts each text line only once, even if it wraps."
  (+ (count-lines (window-start) (point))
     (if (= (current-column) 0) 1 0)
     -1))
```

Also see the functions `bolp` and `eolp` in Section 29.1 [Near Point], page 463. These functions do not move point, but test whether it is already at the beginning or end of a line.

27.2.5 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. Then `vertical-motion` moves point just like `forward-line`. See Section 35.3 [Truncation], page 582.

Because the width of a given string depends on the flags which control the appearance of certain characters, `vertical-motion` will behave differently on a given piece of text found in different buffers. It will even act differently in different windows showing the same buffer, because the width may differ and so may the truncation flag. See Section 35.12 [Usual Display], page 595.

vertical-motion *count* 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.

This function returns the number of lines moved. The value may be less in absolute value than *count* if the beginning or end of the buffer was reached.

move-to-window-line *count* Command

This function moves point with respect to the text currently displayed in the selected window. Point is moved to the beginning of the screen line *count* screen lines from the top of the window. If *count* is negative, point moves either to the beginning of the line −*count* lines from the bottom or else to 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 which 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, with the top line in the window numbered 0.

27.2.6 The User-Level Vertical Motion Commands

A goal column is useful if you want to edit text such as a table in which you want to move point to a certain column on each line. The goal column affects the vertical text line motion commands, `next-line` and `previous-line`. See section "Basic Editing Commands" in *The GNU Emacs Manual*.

goal-column User Option

This variable holds an explicitly specified goal column for vertical line motion commands. If it is an integer, it specifies a column, and these commands try to move to that column on each line. If it is `nil`, then the commands set their own goal columns. Any other value is invalid.

temporary-goal-column Variable

This variable holds the temporary goal column during a sequence of consecutive vertical line motion commands. It is overridden by `goal-column` if that is non-`nil`. It is set each time a vertical motion command is invoked, unless the previous command was also a vertical motion command.

track-eol User Option

This variable controls how the vertical line motion commands operate when starting at the end of a line. If `track-eol` is non-`nil`, then vertical motion starting at the end of a line will keep to the ends of lines. This means moving to the end of each line moved onto. The value of `track-eol` has no effect if point is not at the end of a line when the first vertical motion command is given.

`track-eol` has its effect by causing `temporary-goal-column` to be set to 9999 instead of to the current column.

set-goal-column *unset* Command

This command sets the variable `goal-column` to specify a permanent goal column for the vertical line motion commands. If *unset* is `nil`, then `goal-column` is set to the current column of point. If *unset* is non-`nil`, then `goal-column` is set to `nil`.

This function is intended for interactive use; and in an interactive call, *unset* is the raw prefix argument.

27.2.7 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 31 [Syntax Tables], page 523. See Section 31.4 [Parsing Expressions], page 530, for lower-level primitives for scanning sexps or parts of sexps. For user-level commands, see section "Lists and Sexps" in *GNU Emacs Manual*.

forward-list *arg* Command

Move forward across *arg* balanced groups of parentheses. (Other syntactic entities such as words or paired string quotes are ignored.)

backward-list *arg* Command

Move backward across *arg* balanced groups of parentheses. (Other syntactic entities such as words or paired string quotes are ignored.)

up-list *arg* Command

Move forward out of *arg* levels of parentheses. A negative argument means move backward but still to a less deep spot.

down-list *arg* Command

Move forward down *arg* levels of parentheses. A negative argument means move backward but still go down *arg* level.

forward-sexp *arg* Command

Move 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

Move backward across *arg* balanced expressions.

27.2.8 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 31.3 [Motion and Syntax], page 529.

skip-chars-forward *character-set* &optional *limit* Function

This function moves point in the current buffer forward, skipping over a given set of characters. Emacs first examines the character following point; if it matches *character-set*, then point is advanced and the next character is examined. This continues until a character is found that does not match. The function returns `nil`.

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 30.2 [Regular Expressions], page 507.

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*. It just like `skip-chars-forward` except for the direction of motion.

27.3 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 25.16 [Window Configurations], page 420, and see Section 26.10 [Frame Configurations], page 431).

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 119).

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 500 times in the Lisp sources of Emacs.

The values of point and the mark for other buffers are not saved by `save-excursion`, so any changes made to point and the mark in the other buffers will 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 25.16 [Window Configurations], page 420).

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)))
```

27.4 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 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 continue to do so, but the functions which use them will refuse to operate on text that is inaccessible.

The commands for saving buffers are unaffected by narrowing; the entire buffer is saved regardless of the 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.

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 119). 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` function. 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 they came from), but it does not restore the identity of the current buffer.

Point and the mark are *not* restored by this special form; 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 technique yields correct results if *body* does further narrowing. However, `save-restriction` can become confused if they widen and then make changes outside the area 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 ----------
```

28 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.

28.1 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 27 [Positions], page 439, for a complete description of positions.

A marker has two attributes: the marker position, and the marker buffer. The marker position is an integer which is equivalent (at the moment) to the marker as a position in that buffer; however, as text is inserted or deleted in the buffer, the marker is relocated, so that its integer equivalent changes. The idea is that a marker positioned between two characters in a buffer will remain between those two characters despite any changes made to the contents of the buffer; thus, a marker's offset from the beginning of a buffer may change often during the life of the marker.

If the text around a marker is deleted, the marker is repositioned between the characters immediately before and after the deleted text. If text is inserted at the position of a marker, the marker remains in front of the new text unless it is inserted with `insert-before-markers` (see Section 29.4 [Insertion], page 466). When text is inserted or deleted somewhere before the marker position (not next to the marker), the marker moves back and forth with the two neighboring characters.

When a buffer is modified, all of its markers must be checked so that they can be relocated 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 will eventually be garbage collected, but until then will continue to be updated if they do point somewhere.

Because it is quite common to perform arithmetic operations on a marker position, most of the arithmetic operations (including `+` and `-`) accept markers as arguments. In such cases, the current position of the marker is used.

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 100th and 101st 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>
```

28.2 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 when you are using the arithmetic functions that work with both markers and integers.

markerp *object* — Function

This function returns `t` if *object* is a marker, `nil` otherwise. In particular, 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 (of any type) or a marker, `nil` otherwise.

28.3 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 functions returns a newly allocated 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 27.1 [Point], page 439. 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 27.4 [Narrowing], page 449.

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 27.4 [Narrowing], page 449.

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* 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.

If passed an argument that is an integer whose value is less than 1, `copy-marker` returns a new marker that points to the beginning of the current buffer. If passed an argument that is an integer whose value is greater than the length of the buffer, then `copy-marker` returns a new marker that points to the end of the buffer.

An error is signaled if *marker* is neither a marker nor an integer.

```
(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

(copy-marker 0)
     ⇒ #<marker at 1 in markers.texi>

(copy-marker 20000)
     ⇒ #<marker at 7572 in markers.texi>
```

28.4 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
```

Two distinct markers will be found `equal` (even though not `eq`) to each other if they have the same position and buffer, or if they both point nowhere.

28.5 Changing Markers

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 the value of *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`.

28.6 The Mark

A special marker in each buffer is designated *the mark*. It records a position for the user for the sake of commands such as `C-w` and `C-x TAB`. 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 will provide the values of point and the mark as arguments to the command in an interactive call, but will permit other Lisp programs to specify arguments explicitly. See Section 18.2.2 [Interactive Codes], page 257.

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 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 deactivation only follows 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 35 [Display], page 581.

In addition to the mark, each buffer has a *mark ring* which is a list of markers that are the 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 mark ring may contain no more than the maximum number of entries specified by the variable `mark-ring-max`; excess entries are discarded on a first-in-first-out basis.

mark &optional *force* Function

This function returns the position of the current buffer's mark as an integer.

Normally, if the mark is inactive `mark` signals an error. However, if *force* is non-`nil`, then it 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 which 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, if 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))).
```

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`, it discards marks on a first-in, first-out basis.

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 change the buffer's point, and does nothing if `mark-ring` is empty. It deactivates the mark.

The return value is not useful.

transient-mark-mode User Option

This variable 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 cause the mark to become inactive.

deactivate-mark Variable

If an editor command sets this variable non-`nil`, then the editor command loop deactivates the mark after the command returns.

mark-active Variable

The mark is active when this variable is non-`nil`. This variable is always 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 the region may have changed.

28.7 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 instead use `interactive` with the '`r`' specification, so that the same function can be called with explicit bounds arguments from programs. (See Section 18.2.2 [Interactive Codes], page 257.)

29 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 29.9 [Undo], page 476).

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 28 [Markers], page 453) or or numeric character positions (see Chapter 27 [Positions], page 439). 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)` perform identically. 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.

29.1 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 30.3 [Regexp Search], page 513.

char-after *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`.

Remember that point is always between characters, and the terminal cursor normally appears over the character following point. Therefore, the character returned by `char-after` is the character the cursor is over.

In the following example, assume that the first character in the buffer is '`@`':

```
(char-to-string (char-after 1))
     ⇒ "@"
```

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 the result of `following-char` is 0.

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, then the result of `preceding-char` is 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 27.1 [Point], page 439.

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 27.1 [Point], page 439.

bolp Function

This function returns `t` if point is at the beginning of a line. See Section 27.2.4 [Text Lines], page 443.

eolp Function

This function returns `t` if point is at the end of a line. The end of the buffer is always considered the end of a line.

29.2 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, Emacs 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.

```
---------- 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-string Function

This function returns the contents of the accessible portion of the current buffer as a string. This is the portion between `(point-min)` and `(point-max)` (see Section 27.4 [Narrowing], page 449).

```
---------- Buffer: foo ----------
This is the contents of buffer foo

---------- Buffer: foo ----------

(buffer-string)
     ⇒ "This is the contents of buffer foo
"
```

29.3 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`.

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
```

This function does not exist in Emacs version 18 and earlier.

29.4 Insertion

Insertion takes place at point. Markers pointing at positions after the insertion point are relocated with the surrounding text (see Chapter 28 [Markers], page 453). When a marker points at the place of insertion, it is normally not relocated, so that it points to the beginning of the inserted text; however, when `insert-before-markers` is used, all such markers are relocated to point after the inserted text.

Point may end up either before or after inserted text, depending on the function used. If point is left after the inserted text, we speak of insertion *before point*.

Each of these functions signals an error if the current buffer is read-only.

insert &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`.

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 a marker whose position initially equals point is relocated to come after the newly inserted text.

insert-char *character count* Function

This function inserts *count* instances of *character* into the current buffer before point. *count* must be a number, and *character* must be a character. The value is `nil`.

insert-buffer-substring *from-buffer-or-name* &optional *start end* Function

This function inserts a substring of the contents of buffer *from-buffer-or-name* (which must already exist) into the current buffer before point. The text inserted consists of the characters in the region defined by *start* and *end* (These arguments default to the beginning and end of the accessible portion of that buffer). The 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 ----------
```

29.5 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 function 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 function inserts the last character typed *count* times 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 function calls `auto-fill-function` if the current column number is greater than the value of `fill-column` and the character inserted is a space (see Section 29.12 [Auto Filling], page 481).

This function performs abbrev expansion if Abbrev mode is enabled and the inserted character does not have word-constituent syntax. (See Chapter 32 [Abbrevs], page 535, and Section 31.1.1 [Syntax Class Table], page 524.)

This function is also responsible for calling `blink-paren-function` when the inserted character has close parenthesis syntax (see Section 35.10 [Blinking], page 593).

newline &optional *number-of-newlines* Command

This function inserts newlines into the current buffer before point. If *number-of-newlines* is supplied, that many newline characters are inserted.

In Auto Fill mode, `newline` can break the preceding line if *number-of-newlines* is not supplied. When this happens, it actually inserts 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`.

The value returned is `nil`. In an interactive call, *count* is the numeric prefix argument.

split-line Command

This function 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: a non-`nil` value enables the mode. It is automatically made buffer-local when set in any fashion.

29.6 Deletion of Text

All of the deletion functions operate on the current buffer, and all return a value of `nil`. In addition to these functions, you can also delete text using the "kill" functions that save it in the kill ring; some of these functions save text in the kill ring in some cases but not in the usual case. See Section 29.8 [The Kill Ring], page 472.

erase-buffer Function

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. The value is always `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 function deletes the text in the current buffer in the region defined by *start* and *end*. The value is `nil`.

delete-char *count* &optional *killp* — Command

This function 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 function 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 function 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.

If *count* is negative, then tabs are not changed to spaces, and the characters are deleted by calling `delete-backward-char` with *count*.

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`.

29.7 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, assume that `delete-horizontal-space` is called four times, once on each line, with point between the second and third characters on the line.

```
---------- 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 value is `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.

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 white space between the objects on either side of point with either one space or no space as appropriate. It returns `nil`.

The appropriate amount of space is none at the beginning or end of the line. Otherwise, it is one space except when point is before a character with close parenthesis syntax or after a character with open parenthesis or expression-prefix syntax. See Section 31.1.1 [Syntax Class Table], page 524.

In the example below, when `fixup-whitespace` is called the first time, point is before the word '`spaces`' in the first line. It is located directly after the '`(`' for the second invocation.

```
---------- 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`.

29.8 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. When deleting 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 29.6 [Deletion], page 468.

Emacs saves the last several batches of killed text in a list. We call it the *kill ring* because, in yanking, the elements are considered to be 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, which treat it as a ring.

Some people think use of the word "kill" in Emacs is unfortunate, since it refers to processes which 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 now.

29.8.1 Kill Ring Concepts

The kill ring records killed text as strings in a list. A short kill ring, for example, might look like this:

```
("some text" "a different piece of text" "yet more text")
```

New entries in the kill ring go at the front of the list. 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, the killed portions of text are put into separate entries in the kill ring. But when two or more kill commands are executed in succession, the text they kill forms a single entry, because the second and subsequent consecutive kill commands append to the entry made by the first one.

The user can reinsert or *yank* text from any element in the kill ring. One of the entries in the ring is considered the "front", and the simplest yank command yanks that entry. Other yank commands "rotate" the ring by designating other entries as the "front".

29.8.2 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 uses the `last-command` variable to keep track of whether the previous was a kill command, and in such cases 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. 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.

copy-region-as-kill *start end* Command

This function saves the region defined by *start* and *end* on the kill ring, 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.

Don't use this command in Lisp programs; use `kill-new` or `kill-append` instead. See Section 29.8.4 [Low Level Kill Ring], page 474.

In an interactive call, *start* and *end* are point and the mark.

29.8.3 Functions for Yanking

yank &optional *arg* Command

This function inserts the text in the first entry in the kill ring directly before point. After the yank, the mark is positioned at the beginning and point is positioned after the end of the inserted text.

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.

`yank` does not alter the contents of the kill ring or rotate it. It returns `nil`.

yank-pop *arg* Command

This function replaces the just-yanked text with another batch of killed text—another element of the kill ring.

This command is allowed only immediately after a `yank` or a `yank-pop`. At such a time, the region contains text that was just inserted by the previous `yank`. `yank-pop` deletes that text and inserts in its place a different stretch of killed text. The text that is deleted is not inserted into the kill ring, since it is already in the kill ring somewhere.

If *arg* is `nil`, then the existing region contents are replaced with the previous element of the kill ring. If *arg* is numeric, then the *arg*th previous kill is the replacement. 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 value is always `nil`.

29.8.4 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. They take care of interaction with X Window selections. They do not exist in Emacs version 18.

current-kill *n* &optional *do-not-move* Function

The function `current-kill` rotates the yanking pointer in the kill ring by *n* places, 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 forward 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 also 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, it is called when the "most recent kill" value is called for. If the function returns a non-`nil` values, then that value is used as the "most recent kill". If it returns `nil`, then the first element of the kill ring is used.

interprogram-cut-function Variable

This variable provides a way of communicating killed text to and from 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, it is called whenever the "most recent kill" is changed, with the new string of killed text as an argument.

29.8.5 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 that *yank* functions should copy. 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. Rotating the ring 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. Commands which change the text in the kill ring also set this variable from `kill-ring`. The effect is to rotate the ring so that the newly killed text is at 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 more text")`.

```
kill-ring       kill-ring-yank-pointer
  |               |
  |     ___ ___    --->  ___ ___      ___ ___
   --> |___|___|------> |___|___|--> |___|___|--> nil
         |                |            |
         |                |            |
         |                |             -->"yet more text"
         |                |
         |                 --> "a different piece of text"
         |
          --> "some text"
```

This circumstance might occur after `C-y` (`yank`) immediately followed by `M-y` (`yank-pop`).

kill-ring Variable

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 sublist 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.

29.9 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 which don't have one are usually special-purpose buffers for which Emacs assumes that undoing is not useful.) All the primitives which modify the text in the buffer automatically add elements to the front of the undo list, which you can find 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:

integer This kind of element records a previous value of point. Ordinary cursor motion does not get any sort of undo record, but these entries are used to record where point was before a deletion.

`(`*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.

`(`*pos* `.` *deleted*`)`
This kind of element indicates how to reinsert text that was deleted. The deleted text itself is the string *deleted*. The place to reinsert it is *pos*.

`(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)
```

`nil` This element is a boundary. The function `undo-boundary` adds these elements. 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 between keystroke commands. Thus, each undo normally undoes the effects of one command. Calling this function explicitly is useful for splitting the effects of a command into more than one unit. For example, `query-replace` calls this function 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. 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 never get into the saved value, so they don't cause any trouble.

29.10 Maintaining Undo Lists

This section describes how to enable and disable undo information for a given buffer. It also explains how data from the undo list is discarded 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 function 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 *buffer* Function
buffer-flush-undo *buffer* Function

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`. It cannot be called interactively.

The name `buffer-flush-undo` is not considered obsolete, but the preferred name `buffer-disable-undo` was not provided in Emacs versions 18 and earlier.

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

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 subsequent changes). There is one exception: garbage collection always keeps the very last change group no matter how big it is.

29.11 Filling

Filling means adjusting the lengths of lines (by moving words between them) so that they are nearly (but no greater than) a specified maximum width. Additionally, lines can be *justified*, which means that spaces are inserted between words to make the line exactly the specified width. 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 29.12 [Auto Filling], page 481) 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 functions in this section return values that are not meaningful.

fill-paragraph *justify-flag* Command

This function fills the paragraph at or after point. If *justify-flag* is non-`nil`, each line is justified as well. It uses the ordinary paragraph motion commands to find paragraph boundaries.

fill-region *start end* &optional *justify-flag* Command

This function fills each of the paragraphs in the region from *start* to *end*. It justifies as well if *justify-flag* is non-`nil`. (In an interactive call, this is true if there is a prefix argument.)

The variable `paragraph-separate` controls how to distinguish paragraphs.

fill-individual-paragraphs *start end* &optional *justify-flag mail-flag* Command

This function fills each paragraph in the region according to its individual fill prefix. Thus, if the lines of a paragraph are indented with spaces, the filled paragraph will continue to be indented in the same fashion.

The first two arguments, *start* and *end*, are the beginning and end of the region that will be filled. The third and fourth arguments, *justify-flag* and *mail-flag*, are optional. If *justify-flag* is non-`nil`, the paragraphs are justified as well as filled. If *mail-flag* is non-`nil`, the function is told that it 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 paragraphs with extra 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-flag* Command

This function considers a region of text as a 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-flag* is non-`nil`. In an interactive call, any prefix argument requests justification.

In Adaptive Fill mode, which is enabled by default, `fill-region-as-paragraph` on an indented paragraph when there is no fill prefix uses the indentation of the second line of the paragraph as the fill prefix.

justify-current-line Command

This function inserts spaces between the words of the current line so that the line ends exactly at `fill-column`. It returns `nil`.

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 29.12 [Auto Filling], page 481).

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.

29.12 Auto Filling

Filling breaks text into lines that are no more than a specified number of columns wide. Filled lines end between words, and therefore may have to be shorter than the maximum width.

Auto Fill mode is a minor mode in which Emacs fills lines automatically as text as inserted. This section describes the hook and the two variables used by Auto Fill mode. For a description of functions that you can call manually to fill and justify text, see Section 29.11 [Filling], page 479.

auto-fill-function Variable

The value of this variable should be a function (of no arguments) to be called after self-inserting a space at a column beyond `fill-column`. It may be `nil`, in which case nothing special is done.

The default value for `auto-fill-function` is `do-auto-fill`, 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.

29.13 Sorting Text

The sorting commands 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 78). The values returned by these commands are not meaningful.

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 values. Since Emacs uses the ASCII character set, the ordering in that set determines alphabetical order.

The value of the *record-regexp* argument specifies the textual units or *records* that should be sorted. At the end of each record, a search is done for this regular expression, and the text that matches it is 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 30.2 [Regular Expressions], page 507, 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 to be compared against the other records. 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 used for sorting.

'`\&`' then the whole record is used for sorting.

a regular expression
: then the function searches for a match for the regular expression within the record. If such a match is found, it is used for sorting. If a match for *key-regexp* is not found 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, you are prompted for *record-regexp* and *key-regexp* in the minibuffer.

sort-subr *reverse nextrecfun endrecfun* &optional *startkeyfun endkeyfun* — Command

This command is the general text sorting routine that divides a buffer into records and sorts them. The functions `sort-lines`, `sort-paragraphs`, `sort-pages`, `sort-fields`, `sort-regexp-fields` and `sort-numeric-fields` all use `sort-subr`.

To understand how `sort-subr` works, consider the whole accessible portion of the buffer as being divided into disjoint pieces called *sort records*. A portion of each sort record (perhaps all of it) is designated as the sort key. The records are rearranged in the buffer in order by their sort keys. The records may or may not be contiguous.

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 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),
and BEG and END (the region to sort)."
  (interactive "P\nr")
  (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 ()
              (skip-chars-forward "\n \t\f")))
           'forward-paragraph)
```

sort-lines *reverse start end* Command

This command sorts lines in the region between *start* and *end* alphabetically. If *reverse* is non-`nil`, the sort is in reverse order.

sort-paragraphs *reverse start end* Command

This command sorts paragraphs in the region between *start* and *end* alphabetically. If *reverse* is non-`nil`, the sort is in reverse order.

sort-pages *reverse start end* Command

This command sorts pages in the region between *start* and *end* alphabetically. If *reverse* is non-`nil`, the sort is in reverse order.

sort-fields *field start end* Command

This command sorts lines in the region between *start* and *end*, comparing them alphabetically by the *field*th field of each line. Fields are separated by whitespace and numbered starting from 1. If *field* is negative, sorting is by the −*field*th field from the end of the line. This command is useful for sorting tables.

sort-numeric-fields *field start end* Command

This command sorts lines in the region between *start* and *end*, comparing them numerically by the *field*th field of each line. Fields are separated by whitespace and numbered starting from 1. The specified field must contain a number in each line of the region. 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.

The `sort-columns` function did not work on VMS prior to Emacs 19.

29.14 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. Indentation always counts from zero at the left margin.

29.14.1 Indentation Primitives

This section describes the primitive functions used to count and insert indentation. The functions in the following sections use these primitives.

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*. The value is the column at which the inserted indentation ends.

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 local to the current buffer.

29.14.2 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, except that if that function is `indent-to-left-margin`, `insert-tab` is called instead. (That is a trivial command which inserts a tab character.)

left-margin Variable

This variable is the column to which the default `indent-line-function` will indent. (That function is `indent-to-left-margin`.) In Fundamental mode, `LFD` indents to this column. This variable automatically becomes buffer-local when set in any fashion.

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.

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.

Indentation is done using 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 reindents the new line (the one following the newline just inserted).

Indentation of both lines is done according to the current major mode; this means that the current value of `indent-line-`

`function` is called. 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`.

29.14.3 Indenting an Entire Region

This section describes commands which 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. 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: 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. If indenting a line individually is fast, there is no need for a short cut.

`indent-region` with a non-`nil` argument has a different definition 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`).

29.14.4 Indentation Relative to Previous Lines

This section describes two commands which indent the current line based on the contents of previous lines.

indent-relative &optional *unindented-ok* — Command

This function 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), this function 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 function moves point to the next tab stop by inserting whitespace.

This command returns an unpredictable value.

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 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. The function consists of a call to `indent-relative` with a non-`nil` value passed to the *unindented-ok* optional argument. The value is unpredictable.

If the previous line has no indentation, the current line is given no indentation (any existing indentation is deleted); if the previous nonblank line has no indent points beyond the column at which point starts, nothing is changed.

29.14.5 Adjustable "Tab Stops"

This section explains the mechanism for user-specified "tab stops" and the mechanisms which 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 designated position, like the other indentation functions; it does not affect the display of tab characters in the buffer (see Section 35.12 [Usual Display], page 595). 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 Function

This function inserts spaces or tabs 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. If no such element is found, then nothing is done.

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.

29.14.6 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`.

29.15 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).

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.

A character counts 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 35.12 [Usual Display], page 595.

current-column Function

This function returns the horizontal position of point, measured in columns, counting from 0 at the left margin. The column count is calculated by adding together the widths of all the displayed representations of the characters between the start of the current line and point.

For a more complicated example of the use of `current-column`, see the description of `count-lines` in Section 27.2.4 [Text Lines], page 443.

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 all the displayed representations of the characters between the start of the line and point.

If the argument *column* is greater than the column position of 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*.

The argument *force* also has an effect if the line isn't long enough to reach column *column*; in that case, it says to indent 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.

29.16 Case Changes

The case change commands described here work on text in the current buffer. See Section 4.7 [Character Case], page 62, for case conversion commands that work on strings and characters. See Section 4.8 [Case Table], page 63, 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 word the before point (if moving forward) or after point (if operating backward) is ignored. 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.

29.17 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. 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, which is usually a symbol, and an associated value, which can be any Lisp object—just as for properties of symbols (see Section 7.4 [Property Lists], page 100).

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`.

29.17.1 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 29.17.3 [Property Search], page 496, 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 which is a symbol, then `get-text-property` returns the *prop* property of that symbol.

text-properties-at *position* &optional *object* Function

This function returns the list of properties held by the character at *position* in the string or buffer *object*. If *object* is `nil`, it defaults to the current buffer.

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*.

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* has a property *prop* whose value differs from *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*.

29.17.2 Changing Text Properties

The primitives for changing properties apply to a specified range of text. 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 buffer's contents, and can affect how the buffer looks on the screen, any change in the text properties is considered a buffer modification. Buffer text property changes are undoable.

add-text-properties *start end props* &optional *object* Function

This function modifies the 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 change. It should have the form of a property list (see Section 7.4 [Property Lists], page 100): 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` property to `t` for a range of text:

```
(add-text-properties (region-beginning)
                     (region-end)
                     (list 'comment t))
```

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.

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 100): a list whose elements include the property names followed by the corresponding values. The property names mentioned in *props* are the ones deleted from the text. The values associated in *props* with these names do not matter.

The return value is `t` if the function actually changed some property's value; `nil` otherwise (if *props* is `nil` or if none of the text had any of those properties).

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 have the form of a list whose elements include the property names followed by the 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 (region-beginning)
                     (region-end)
                     nil)
```

29.17.3 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. In all cases, *object* defaults to the current buffer.

next-property-change *pos* &optional *object* 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*.

The value is `nil` if the properties remain unchanged all the way to the end of *object*. If the value is non-`nil`, it is a position greater than *pos*, never equal.

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* 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*.

The value is `nil` if the properties remain unchanged all the way to the end of *object*. If the value is non-`nil`, it is a position greater than *pos*, never equal.

previous-property-change *pos* &optional *object* 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 always strictly less than *pos*.

previous-single-property-change *pos prop* &optional *object* — 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 always strictly less than *pos*.

29.17.4 Special Properties

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.

You can use the property `face` to control the font and color of text. See Section 35.9 [Faces], page 591, for more information. This feature is temporary; in the future, we may replace it with other ways of specifying how to display text.

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.

You can specify a different keymap for a portion of the text by means of a `local-map` property. The property's value, for the character after point, replaces the buffer's local map. See Section 19.7 [Active Keymaps], page 300.

If a character has the property `read-only`, then modifying that character is not allowed. Any command that would do so gets an error.

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 24.6 [Read Only Buffers], page 389.

A non-`nil` `invisible` property means a character does not appear on the screen. This works much like selective display. Details of this feature are likely to change in future versions, so check the '`etc/NEWS`' file in the version you are using.

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.

Insertion of text does not, strictly speaking, change any existing character, so there is a special rule for insertion. It compares the `read-only` properties of the two surrounding characters; if they are non-`nil` and `eq` to each other, then the insertion is not allowed. Assuming insertion is allowed, it then 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.

See also Section 29.21 [Change Hooks], page 503, for other hooks that are called when you change text in a buffer.

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). The `point-left` functions are always called before the `point-entered` functions.

A primitive function may 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.

29.17.5 Stickiness of Text Properties

Inserting a string with no text properties into the buffer normally gives the inserted text the same properties as the preceding character. You can control this copying of properties by setting the `front-sticky` and `rear-nonsticky` properties of a character.

If you make a character's `front-sticky` property `t`, then insertion before the character receives its properties. If you make the `rear-nonsticky` property `t`, then insertion after that character does *not* receive its properties. You can regard characters as being normally "rear-sticky" by default, but not "front-sticky"; thus, by default, insertion normally receives properties from the previous character only.

If neither side of an insertion is suitably sticky, then the inserted text gets no properties. If both sides are sticky, then the inserted text gets the properties of both sides, with the previous character's properties taking precedence when both sides have a property in common.

You can also specify stickiness for individual properties. To do so, use a list of property names as the value of the `front-sticky` property or the `rear-nonsticky` property. For example, if a character has a `rear-nonsticky` property whose value is `(face read-only)`, then insertion after the character does not receive its `face` property its or `read-only` property (if any), but does receive any other properties it has.

The merging of properties when both sides of the insertion are sticky takes place one property at a time. If the preceding character is `rear-sticky` for the property, and the property is non-`nil`, it dominates. Otherwise, the following character's property value is used if it is `front-sticky` for that property.

29.17.6 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 undo the kill, you get two intervals with the same properties. Thus, the distinction can't be preserved when editing happens.

But 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 if it was two intervals, and the killed text equals one of them, then undoing the kill yields just one interval. Again, the distinction can't be preserved.

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.

For practical purposes, the property search functions serve 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 29.17.3 [Property Search], page 496.

Emacs also provides explicit intervals as a presentation feature; see Section 35.8 [Overlays], page 588.

29.18 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 useful for changes which are not considered significant, such as when Outline mode changes visible lines to invisible lines and vice versa.

`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 which were actually changed by the translation. This does not count characters which were mapped into themselves in the translation table.

This function is available in Emacs versions 19 and later.

29.19 Underlining

The underlining commands are somewhat obsolete. The `underline-region` function actually inserts '`_^H`' before each appropriate character in the region. This command provides a minimal text formatting feature that might work on your printer; however, we recommend instead that you use more powerful text formatting facilities, such as Texinfo.

underline-region *start end* Command

This function underlines all nonblank characters in the region defined by *start* and *end*. That is, an underscore character and a backspace character are inserted just before each non-whitespace character in the region. The backspace characters are intended to cause overstriking, but in Emacs they display as either '`\010`' or '`^H`', depending on the setting of `ctl-arrow`. There is no way to see the effect of the overstriking within Emacs. The value is `nil`.

ununderline-region *start end* Command

This function removes all underlining (overstruck underscores) in the region defined by *start* and *end*. The value is `nil`.

29.20 Registers

A register is a sort of variable used in Emacs editing that can hold a marker, a string, a rectangle, a window configuration (of one frame), or a frame configuration (of all frames). Each register is named by a single character. All characters, including control and meta characters (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 a character which is its name.

The functions in this section return unpredictable values unless otherwise stated.

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 object *contents* is a string, marker, or list representing the register contents. A string represents text stored in the register. A marker represents a position. A list represents a rectangle; its elements are strings, one per line of the rectangle.

view-register *reg* — Command

This command displays what is contained in register *reg*.

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*.

point-to-register *reg* — Command

This command stores both the current location of point and the current buffer in register *reg* as a marker.

jump-to-register *reg* — Command
register-to-point *reg* — Command

This command restores the status recorded in register *reg*.

If *reg* contains a marker, it moves point to the position stored in the marker. Since both the buffer and the location within the buffer are stored by the `point-to-register` function, this command can switch you to another buffer.

If *reg* contains a window configuration or a frame configuration. `jump-to-register` restores that configuration.

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.

copy-to-register *reg start end* &optional *delete-flag* — Command

This command copies the region from *start* to *end* into register *reg*. If *delete-flag* is non-`nil`, it deletes the region from the buffer after copying it into the register.

prepend-to-register *reg start end* &optional *delete-flag* Command

This command prepends the region from *start* to *end* into register *reg*. If *delete-flag* is non-`nil`, it deletes the region from the buffer after copying it to the register.

append-to-register *reg start end* &optional *delete-flag* Command

This command appends the region from *start* to *end* to the text already in register *reg*. If *delete-flag* is non-`nil`, it deletes the region from the buffer after copying it to the register.

copy-rectangle-to-register *reg start end* &optional *delete-flag* Command

This command copies a rectangular region from *start* to *end* into register *reg*. If *delete-flag* is non-`nil`, it deletes the region from the buffer after copying it to the register.

window-configuration-to-register *reg* Command

This function stores the window configuration of the selected frame in register *reg*.

frame-configuration-to-register *reg* Command

This function stores the current frame configuration in register *reg*.

29.21 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 29.17.4 [Special Properties], page 497, 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-function Variable

If this variable is non-`nil`, then it should be a function; the function is called before any buffer modification. Its arguments are the beginning and end of the region that is going to change, represented as integers. The buffer that's about to change is always the current buffer.

after-change-function Variable

If this variable is non-`nil`, then it should be a function; the function is called after any buffer modification. It receives three arguments: the beginning and end of the region just changed, and the length of the text that existed before the change. (To get the current length, subtract the region beginning from the region end.) All three arguments are integers. The buffer that's about to change is always the current buffer.

Both of these variables are temporarily bound to `nil` during the time that either of these hooks is running. This means that if one of these functions changes the buffer, that change won't run these functions. If you do want the hook function to be run recursively, write your hook functions to bind these variables back to their usual values.

first-change-hook Variable

This variable is a normal hook; its hook functions are run using `run-hooks` whenever a buffer is changed that was previously in the unmodified state.

The variables described in this section are meaningful only starting with Emacs version 19.

30 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 identify the text matched by parts of the regular expression by examining the *match data*.

30.1 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.

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 positioned at the beginning of the line. Then `(search-forward "fox")` is evaluated in the minibuffer and point is left after the last letter of '`fox`':

```
---------- Buffer: foo ----------
⋆The quick brown fox jumped over the lazy dog.
---------- Buffer: foo ----------

(search-forward "fox")
     ⇒ t

---------- 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 programs may depend on a value of `nil`.)

If *repeat* is non-`nil`, then the search is repeated that many times. Point is positioned at the end of the last match.

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.

A word search differs from a simple string search in that a word search **requires** that the words it searches for are present as entire words (searching for the word '`ball`' does not match the word '`balls`'), and punctuation and spacing are ignored (searching for '`ball boy`' does match '`ball. Boy!`').

In this example, point is first placed 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.")
     ⇒ t

---------- 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 `t`, then `word-search-forward` returns `nil` when a search fails, instead of signaling an error. If *noerror* is neither `nil` nor `t`, then `word-search-forward` 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.

30.2 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.

30.2.1 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 which 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 which 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 which begins with '`a`' and ends with '`b`'.

`*`
: is not a construct by itself; it is a suffix that means the preceding regular expression is to be repeated as many times as possible. In '`fo*`', the '`*`' applies to the '`o`', so '`fo*`' matches one '`f`' followed by any number of '`o`'s. The case of zero '`o`'s is allowed: '`fo*`' does match '`f`'.

'*' always applies to the *smallest* possible preceding expression. Thus, 'fo*' has a repeating 'o', not a repeating 'fo'.

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 case that makes it possible to match the rest of the pattern. For example, matching 'ca*ar' against the string 'caaar', the 'a*' first tries to match all three 'a's; but the rest of the pattern is 'ar' and there is only 'r' left to match, so this try fails. The next alternative is for 'a*' to match only two 'a's. With this choice, the rest of the regexp matches successfully.

`+` is a suffix character similar to '*' except that it must match the preceding expression at least once. So, for example, 'ca+r' will match the strings 'car' and 'caaaar' but not the string 'cr', whereas 'ca*r' would match all three strings.

`?` is a suffix character similar to '*' except that it can match the preceding expression either once or not at all. For example, 'ca?r' will match 'car' or 'cr'; nothing else.

`[ ... ]` '[' begins a *character set*, which is terminated by a ']'. In the simplest case, the characters between the two form the set. 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.

Character ranges can also be included in a character set, by writing two characters with a '-' between them. Thus, '[a-z]' matches any lower case letter. Ranges may be intermixed freely with individual characters, as in '[a-z$%.]', which matches any lower case letter or '$', '%' or a period.

Note that the usual special characters are not special any more inside a character set. A completely different set of special characters exists inside character sets: ']', '-' and '^'.

To include a ']' in a character set, make it the first character. For example, '[]a]' matches ']' or 'a'. To include a '-', write '-' as the first or last character in the range.

To include '^', make it other than the first character in the set.

`[^ ... ]` '[^' begins a *complement character set*, 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 set unless it is the first character. The character following the '^' is treated as if it were first (thus, '-' and ']' are not special there).

Note that a complement character set can match a newline, unless newline is mentioned as one of the characters not to match.

^
: 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' which occurs at the beginning of a line.

When matching a string, '^' 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, '$' 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 which matches only '$', and '\[' is a regular expression which matches only '[', and so on.

Note that '\' also has special meaning in the read syntax of Lisp strings (see Section 2.3.7 [String Type], page 26), 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; better to 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: characters which, when preceded by '\', are special constructs. Such characters are always ordinary when encountered on their 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, '`\(foo\|bar\)x`' matches either '`foox`' or '`barx`'.
2. To enclose a complicated expression for a suffix character such as '`*`' to operate on. Thus, '`ba\(na\)*`' matches '`bananana`', etc., with any (zero or more) number 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 which 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 which is matched the *digit*th time by a previous '`\( ... \)`' 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 mean "match the same text matched the *digit*th time by the '`\( ... \)`' construct."

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.

``\` `` matches the empty string, provided it is at the beginning of the buffer or string being matched against.

`\'` matches the empty string, provided it is at the end of the buffer or string being matched against.

`\=` matches the empty string, provided it is at point. (This construct is not defined when matching against a string.)

`\b` matches the empty string, provided it is 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 the empty string, provided it is *not* at the beginning or end of a word.

`\<` matches the empty string, provided it is at the beginning of a word.

`\>` matches the empty string, provided it is at the end of a word.

`\w` matches any word-constituent character. The editor syntax table determines which characters these are. See Chapter 31 [Syntax Tables], page 523.

`\W` matches any character that is not a word-constituent.

`\s`*code* matches any character whose syntax is *code*. Here *code* is a character which represents a syntax code: thus, '`w`' for word constituent, '`-`' for whitespace, '`(`' for open parenthesis, etc. See Chapter 31 [Syntax Tables], page 523, for a list of the codes.

`\S`*code* matches any character whose syntax is not *code*.

Not every string is a valid regular expression. For example, any string with unbalanced square brackets is invalid, and so is a string that ends with a single '`\`'. If an invalid regular expression is passed to any of the search functions, an `invalid-regexp` error is signaled.

regexp-quote *string* Function

This function returns a regular expression string which 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 which is the value of `string`, surrounded by whitespace:

```
(re-search-forward
 (concat "\\s " (regexp-quote string) "\\s "))
```

30.2.2 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 enable you to distinguish the spaces from the 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 case, the tab and carriage return are the actual characters.

This regular expression contains four parts in succession and can be deciphered as follows:

`[.?!]`
: The first part of the pattern consists of three characters, a period, a question mark and an exclamation mark, within square brackets. 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 set, 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, or a tab, or two spaces. The double backslashes are needed to prevent Emacs from reading the parentheses and vertical bars as part of the search pattern; the parentheses are used to mark the group and the vertical bars are used to indicated that the patterns to either side of them are alternatives. The dollar sign is used to match the end of a line. The tab character is written using '`\t`' and the two spaces are written as themselves.

`[ \t\n]*` Finally, the last part of the pattern indicates that the end of the line or the whitespace following the period, question mark or exclamation mark may, but need not, be followed by additional whitespace.

30.3 Regular Expression Searching

In GNU Emacs, you can search for the next match for a regexp either incrementally or not. Incremental search commands are described in the *The GNU Emacs Manual*. 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`.

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 string found that does match.

If the search is successful (i.e., if text matching *regexp* is found), then point moves to the end of that text, and the function returns the new value of point.

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`, `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`.

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). The call succeeds if all these searches succeeded, and point is left at the end of the match found by the last search. Otherwise the search fails.

In the following example, point is initially located directly before the '`T`'. After evaluating the form, point is located at the end of that line (between the '`t`' of '`hat`' and before the newline).

```
---------- Buffer: foo ----------
I read "⋆The cat in the hat
comes back" twice.
---------- Buffer: foo ----------
```

```
(re-search-forward "[a-z]+" nil t 5)
     ⇒ t

---------- 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. 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 is done at a specified beginning position. Thus, true mirror-image behavior would require a special feature for matching regexps from end to beginning.

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 30.5 [Match Data], page 517.

```
(string-match
 "quick" "The quick brown fox jumped quickly." 8)
     ⇒ 27
(match-end 0)
     ⇒ 32
```

The `match-beginning` and `match-end` functions are described together; see Section 30.5 [Match Data], page 517.

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 must succeed 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` or `match-end`. See Section 30.5 [Match Data], page 517.

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
```

30.4 Replacement

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, the number of occurrences to consider. In this case, `perform-replace` returns after considering that many occurrences.

Normally, the keymap `query-replace-map` defines the possible user responses. 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 special 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. 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 don't ask any more.

`act-and-exit`
Answer this question "yes," and don't ask any more.

`act-and-show`
Answer this question "yes," but show the results—don't advance yet.

`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 item—instead of any other answer.

`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 the `y-or-n-p` functions use this answer.

`help` Display some help, then ask again.

30.5 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 extract parts of it.

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.

30.5.1 Simple Match Data Access

This section explains how to use the match data to find the starting point or ending point of the text that was matched by a particular search, or by a particular parenthetical subexpression of a regular expression.

match-beginning *count* Function

This function returns the position of the start of text matched by the last regular expression searched for. *count*, a number, specifies a subexpression whose start position is the value. If *count* is zero, then the value is the position of the text matched by the whole regexp. If *count* is greater than zero, then the value is the position of the beginning of the text matched by the *count*th subexpression, regardless of whether it was used in the final match.

Subexpressions of a regular expression are those expressions grouped inside of 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.

The value is `nil` for a parenthetical grouping inside of a '`\|`' alternative that wasn't used in the match.

The `match-end` function is similar to the `match-beginning` function except that it returns the position of the end of the matched text.

Here is an example, with a comment showing the numbers of the positions in the text:

```
(string-match
 "\\(qu\\)\\(ick\\)" "The quick fox jumped quickly.")
     ⇒ 4             ;^^^^^^^^^^
                     ;0123456789

(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. Before the form is evaluated, point is located at the beginning of the line. After evaluating the search form, point is located on the line 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))
    ⇒ (t 9 13)

---------- Buffer: foo ----------
I read "The cat ⋆in the hat comes back" twice.
        ^   ^
        9  13
---------- Buffer: foo ----------
```

(Note that in this case, the index returned is a buffer position; the first character of the buffer counts as 1.)

match-end *count* Function

This function returns the position of the end of text matched by the last regular expression searched for. This function is otherwise similar to `match-beginning`.

30.5.2 Replacing the Text That Matched

This function replaces the text matched by the last search with *replacement*.

replace-match *replacement* &optional *fixedcase* *literal* — Function

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, except when all of the words in the original text are only one character long. In that event, the replacement text is capitalized. If *all* of the words in the original text are capitalized, then all of the words in the replacement text are capitalized.

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*' stands for the *n*th subexpression in the original regexp. Subexpressions are those expressions grouped inside of '`\(...\)`'. *n* is a digit.

'`\\`' — '`\\`' stands for a single '`\`' in the replacement text.

`replace-match` leaves point at the end of the replacement text, and returns `t`.

30.5.3 Accessing the Entire Match Data

The functions `match-data` and `store-match-data` let you read or write the entire match data, all at once.

match-data — Function

This function returns a new 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. 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`. (In Emacs 18 and earlier versions, markers were used even for matching on a string, except in the case of the integer 0.)

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>)
```

store-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.

30.5.4 Saving and Restoring the Match Data

All asynchronous process functions (filters and sentinels) and functions that use `recursive-edit` should save and restore the match data if they do a search or if they let the user type arbitrary commands. Saving the match data is useful in other cases as well—whenever you want to access the match data resulting from an earlier search, notwithstanding another intervening search.

This example shows the problem that can arise if you fail to attend to this requirement:

```
(re-search-forward "The \\(cat \\)")
     ⇒ 48
(foo)                      ; Perhaps foo does
                           ;    more searching.
(match-end 0)
     ⇒ 61                  ; Unexpected result—not 48!
```

In Emacs versions 19 and later, you can save and restore the match data with `save-match-data`:

save-match-data *body*... Special Form

This special form executes *body*, saving and restoring the match data around it. This is useful if you wish to do a search without altering the match data that resulted from an earlier search.

You can use `store-match-data` together with `match-data` to imitate the effect of the special form `save-match-data`. This is useful for writing code that can run in Emacs 18. Here is how:

```
(let ((data (match-data)))
  (unwind-protect
      ...   ; May change the original match data.
    (store-match-data data)))
```

30.6 Standard Regular Expressions Used in Editing

Here are the regular expressions standardly used in editing:

page-delimiter Variable

This is the regexp describing line-beginnings that separate pages. The default value is `"^\014"` (i.e., `"^^L"` or `"^\C-l"`).

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 is a line that consists entirely of spaces, tabs, and form feeds.

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.

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 by a closing brace, followed by tabs, spaces or new lines.

For a detailed explanation of this regular expression, see Section 30.2.2 [Regexp Example], page 512.

30.7 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. Regexps, and in particular character sets, are included: 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 per-buffer-local variable; altering the variable affects only the current buffer. (See Section 10.9.1 [Intro to Buffer-Local], page 144.) 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.

case-replace User Option

This variable determines whether `query-replace` should preserve case in replacements. If the variable is `nil`, then case need not be preserved.

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)`.

31 Syntax Tables

A *syntax table* provides Emacs with the information that determines the syntactic use of each character in a buffer. This information is used by the parsing commands, 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 27.2.2 [Word Motion], page 441) and the list motion functions (see Section 27.2.7 [List Motion], page 446) as well as the functions in this chapter.

A syntax table is a vector of 256 elements; it contains one entry for each of the 256 ASCII characters of an 8-bit byte. Each element is an integer that encodes the syntax of the character in question.

Syntax tables are used only for moving across text, not for the GNU Emacs Lisp reader. GNU Emacs Lisp uses built-in syntactic rules when reading Lisp expressions, and these rules cannot be changed.

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 which 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 20.1.2 [Example Major Modes], page 318, for an example of how to set up a syntax table.

syntax-table-p *object* Function

This function returns `t` if *object* is a vector of length 256 elements. This means that the vector may be a syntax table. However, according to this test, any vector of length 256 is considered to be a syntax table, no matter what its contents.

31.1 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 desired syntax.

Emacs defines twelve *syntax classes*. Each syntax table puts each character into one class. 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 which is frequently put in that class; however, its meaning as a designator is unvarying and independent of how it is actually classified.

A syntax descriptor is a Lisp string which specifies a syntax class, a matching character (unused except for 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.

Thus, the 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).

31.1.1 Table of Syntax Classes

Here is a summary of the classes, the characters that stand for them, their meanings, and examples of their use.

whitespace character Syntax class

Whitespace characters (designated with '` `' or '`-`') separate symbols and words from each other. Typically, whitespace characters have no other syntactic use, and multiple whitespace characters are syntactically equivalent to a single one. Space, tab, newline and formfeed are almost always considered whitespace.

word constituent Syntax class

Word constituents (designated with '`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 with '`_`') 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 (‘.’) 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 35.10 [Blinking], page 593.

The class of open parentheses is designated with ‘(’, and that of close parentheses with ‘)’.

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 with ‘"’) is used to delimit string constants in many languages, including Lisp and C. 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 with '\') 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 27.2.2 [Word Motion], page 441.

character quote Syntax class

A *character quote character* (designated with '`/`') 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 27.2.2 [Word Motion], page 441.

This class is not currently used in any standard Emacs modes.

paired delimiter Syntax class

Paired delimiter characters (designated with '`$`') are like string quote characters except that the syntactic properties of the characters between the delimiters are not suppressed. Only TeX mode uses a paired identical delimiter presently—the '`$`' that begins and ends math mode.

expression prefix Syntax class

An *expression prefix operator* (designated with '`'`') is used for syntactic operators that are part of an expression if they appear next to one but are not part of an adjoining symbol. These characters in Lisp 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 different languages to delimit comments. These classes are designated with '`<`' and '`>`', respectively.

English text has no comment characters. In Lisp, the semicolon ('`;`') starts a comment and a newline or formfeed ends one.

31.1.2 Syntax Flags

In addition to the classes, entries for characters in a syntax table can include 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 ('*/').

The flags for a character *c* are:

- '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 can now supports two comment styles simultaneously. (This is for the sake of C++.) More specifically, it can recognize two different comment-start sequences. Both must share the same first character; only the second character may differ. Mark the second character of the "b"-style comment start sequence with the 'b' flag.

  The two styles of comment can have different comment-end sequences. 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'

  Thus '/*' is a comment-start sequence for "a" style, '//' is a comment-start sequence for "b" style, '*/' is a comment-end sequence for "a" style, and newline is a comment-end sequence for "b" style.

- '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 (‘'’).

31.2 Syntax Table Functions

In this section we describe functions for creating, accessing and altering syntax tables.

make-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.

copy-syntax-table &optional *table* Function

This function is identical to `make-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 31.1 [Syntax Descriptors], page 523.

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 ?/ ".13")
     ⇒ nil
```

char-syntax *character* Function

This function returns the syntax class of *character*, represented by its mnemonic designator character. This *only* returns the class, not any matching parenthesis or flags.

An error is signaled if *char* is not a character.

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 in C-mode. This does not show the fact that it is also a comment sequence character. 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, ‘)’.

```
(char-to-string (char-syntax ?\ ))
     ⇒ " "

(char-to-string (char-syntax ?/))
     ⇒ "."

(char-to-string (char-syntax ?\())
     ⇒ "("
```

set-syntax-table *table* Function

This function makes *table* the syntax table for the current buffer. It returns *table*.

syntax-table Function

This function returns the current syntax table, which is the table for the current buffer.

31.3 Motion and Syntax

This section describes functions for moving across characters in certain syntax classes. None of these functions exists in Emacs version 18 or earlier.

skip-syntax-forward *syntaxes* &optional *limit* Function

This function moves point forward across characters whose syntax classes are mentioned in *syntaxes*. It stops when it encounters the end of the buffer, or position *lim* (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 *lim* (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 chars with expression prefix syntax. This includes both characters in the expression prefix syntax class, and characters with the '`p`' flag.

31.4 Parsing Balanced Expressions

Here are several functions for parsing and scanning balanced expressions. 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 27.2.7 [List Motion], page 446, 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 an expression in the current buffer starting at *start*, not scanning past *limit*. Parsing stops at *limit* or when certain criteria described below are met; point is set to the location where parsing stops. The value returned is a description of the status of the parse at the point where it stops.

Normally, *start* is assumed to be the top level of an expression to be parsed, such as the beginning of a function definition. Alternatively, you might wish to resume parsing in the middle of an expression. To do this, you must provide a *state* argument that describes the initial status of parsing. If *state* is omitted (or `nil`), parsing assumes that *start* is the beginning of a new parse at level 0.

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.

The fifth argument *state* is a seven-element list of the same form as the value of this function, described below. 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 seven elements describing the final state of the parse:

0. The depth in parentheses, starting at 0.
1. The character position of the start of the innermost containing parenthetical grouping; `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. (It is the character that will terminate the string.)
4. `t` if inside a comment.
5. `t` if point is just after a quote character.
6. The minimum parenthesis depth encountered during this scan.

Elements 1, 4, 5, and 6 are significant in the argument *state*.

This function is used to determine how to indent lines in programs written in languages that have nested parentheses.

scan-lists *from count depth* Function

This function scans forward *count* balanced parenthetical groupings from character number *from*. It returns the character number of the position thus found.

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 levels of parenthesis.

Comments are ignored if `parse-sexp-ignore-comments` is non-`nil`.

If the beginning or end of the buffer (or its accessible portion) is reached 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

Scan from character number *from* by *count* balanced expressions. It returns the character number of the position thus found.

Comments are ignored if `parse-sexp-ignore-comments` is non-`nil`.

If the beginning or end of (the accessible part of) the buffer is reached in the middle of a parenthetical grouping, an error is signaled. If the beginning or end is reached 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 to skip cannot exceed that many.

31.5 Some Standard Syntax Tables

Each of the major modes in Emacs has its own syntax table. 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 in use in 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.)

31.6 Syntax Table Internals

Each element of a syntax table is an integer that translates into the full meaning of the entry: class, possible matching character, and flags. However, it is not common for a programmer to work with the entries directly in this form since the Lisp-level syntax table functions usually work with syntax descriptors (see Section 31.1 [Syntax Descriptors], page 523).

The low 8 bits of each element of a syntax table indicates the syntax class.

Integer	*Class*
0	whitespace
1	punctuation
2	word
3	symbol
4	open parenthesis
5	close parenthesis
6	expression prefix
7	string quote
8	paired delimiter
9	escape
10	character quote
11	comment-start
12	comment-end

The next 8 bits are the matching opposite parenthesis (if the character has parenthesis syntax); otherwise, they are not meaningful. The next 6 bits are the flags.

32 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; 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, they will never be 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 97.

For the user-level commands for abbrevs, see section "Abbrev Mode" in *The GNU Emacs Manual*.

32.1 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 local when set in any fashion.

default-abbrev-mode Variable

This is the value `abbrev-mode` for buffers that do not override it. This is the same as `(default-value 'abbrev-mode)`.

32.2 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 `nil`s.

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 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 value is always `nil`.

If *human* is non-`nil`, a human-oriented description is inserted. Otherwise the description is a Lisp expression—a call to `define-abbrev-table` which would define *name* exactly as it is currently defined.

32.3 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*. 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 which internally represents the new abbrev, or `nil` if the user declines to redefine an existing abbrev.

define-abbrev *table name expansion hook* Function

This function defines an abbrev in *table* named *name*, to expand to *expansion*, and call *hook*. The return value is an uninterned symbol which represents the abbrev inside Emacs; its name is *name*.

The argument *name* should be a string. The argument *expansion* should be a string, or `nil`, to undefine the abbrev.

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*.

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 functioning of the functions in this section; it is examined by their callers.

32.4 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 13.1 [How Programs Do Loading], page 175). 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 which when loaded will define the same abbrevs. This function returns `nil`.

32.5 Looking Up and Expanding Abbreviations

Abbrevs are usually expanded by commands for interactive use, including `self-insert-command`. This section describes the subroutines used in writing such functions, 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. By default, this function tries first the current buffer's local abbrev table, and second the global abbrev table.

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-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; if it is specified, the abbrev is looked up in that table only.

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 last abbrev expanded. This information is left by `expand-abbrev` for the sake of the `unexpand-abbrev` command.

last-abbrev-location Variable

This is the location of the last 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 last abbrev expanded, as results from case conversion. 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 20.4 [Hooks], page 332. 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 function issues a prompt. Thus, this hook allows the user to decide whether the abbrev should be expanded, and to abort expansion if it is not desired.

```
(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 enters 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"))))
```

32.6 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. 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.

c-mode-abbrev-table Variable

This is the local abbrev table used in C mode.

lisp-mode-abbrev-table Variable

This is the local abbrev table used in Lisp mode and Emacs Lisp mode.

33 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.

33.1 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 33.3 [Asynchronous Processes], page 545). The other two, `call-process` and `call-process-region`, create a synchronous process and do not return a process object (see Section 33.2 [Synchronous Processes], page 543).

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. The actual file containing the program is found by following normal system rules: if the file name is absolute, then the program must be found in the specified file; if the name is relative, then the directories in `exec-path` are searched sequentially for a suitable file. The variable `exec-path` is initialized when Emacs is started, 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 22.10.4 [File Name Expansion], page 367).

Each of the subprocess-creating functions has a *buffer-or-name* argument which specifies where the standard output from the program will go. If *buffer-or-name* is `nil`, that says to discard the output unless a filter function handles it. (See Section 33.8.2 [Filter Functions], page 553, and Chapter 16 [Streams], page 221.) 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 are not allowed in these strings, since they 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. Such arguments must be provided via *args*.

The subprocess gets its current directory from the value of `default-directory` (see Section 22.10.4 [File Name Expansion], page 367).

The subprocess inherits its environment from Emacs; but you can specify overrides for it with `process-environment`. See Section 34.3 [System Environment], page 567.

exec-directory Variable

The value of this variable is the name of a directory (a string) that contains programs that come with GNU Emacs, that are intended for Emacs to invoke. The program `wakeup` is an example of such a program; the `display-time` command uses it to get a reminder once per minute.

The default value is the name of a directory whose name ends in '`arch-lib`'. We call the directory '`emacs/arch-lib`', since its name usually ends that way. We sometimes refer to "the directory '`emacs/arch-lib`'," when strictly speaking we ought to say, "the directory named by the variable `exec-directory`." Most of the time, there is no difference.

(In earlier Emacs versions, prior to version 19, these files lived in the directory '`emacs/etc`' instead of in '`emacs/arch-lib`'.)

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.

33.2 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`, and the process is killed by sending it a `SIGKILL` signal. See Section 18.8 [Quitting], page 280.

The synchronous subprocess functions return `nil` in version 18. In version 19, they will return an indication of how the process terminated.

call-process *program* &optional *infile* *buffer-or-name* *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 process output gets inserted in buffer *buffer-or-name* before point, if that argument names a buffer. If *buffer-or-name* is `t`, output is sent to the current buffer; if *buffer-or-name* is `nil`, output is discarded.

If *buffer-or-name* is the integer 0, `call-process` returns `nil` immediately and discards any output. 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.

If *display* is non-`nil`, then `call-process` redisplays the buffer as output is inserted. Otherwise the function 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 are supplied as the 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.

The examples below are both run with the buffer '`foo`' 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 ----------
```

The `dired-readin` function contains a good example of the use of `call-process`:

```
(call-process
 "ls" nil buffer nil dired-listing-switches dirname)
```

call-process-region *start end program* &optional *delete buffer-or-name 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`, which may be useful when the output is going to be inserted back in the current buffer.

If *buffer-or-name* names a buffer, the output is inserted in that buffer at point. If *buffer-or-name* is `t`, the output is sent to the current buffer. If *buffer-or-name* is `nil`, the output is discarded. If *buffer-or-name* is the integer 0, the output is discarded and `call-process` returns `nil` immediately, just as `call-process` would.

If *display* is non-`nil`, then `call-process-region` redisplays the buffer as output is inserted. Otherwise the function 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 are supplied as the 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 *buffer-or-name* 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.
```

33.3 Creating an Asynchronous Process

After an *asynchronous process* is created, Emacs and the Lisp program can continue running immediately. The process may thereafter run in parallel with Emacs, and the two may communicate with each other using the functions described in following sections. Here we describe how to create an asynchronous process, with `start-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 for Emacs Lisp programs. The argument *name* specifies the name for the process object; if a process with this name already exists, then *name* is modified (by adding '<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 are supplied as the 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, 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.

process-connection-type — Variable

This variable controls the type of device used to communicate with asynchronous subprocesses. If it is `nil`, then pipes are used. If it is `t`, then PTYs are used (or pipes if PTYs are not supported).

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 in order to change it for just one call of `start-process`, temporarily rebind it with `let`.

```
(let ((process-connection-type nil))  ; Use a pipe.
  (start-process ...))
```

33.4 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.)

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 Variable

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*. The argument *name* may be a process, the name of a process, a buffer, or the name of a buffer. The subprocess is killed with a `SIGHUP` signal.

```
(delete-process "*shell*")
     ⇒ nil
```

process-kill-without-query *process* Function

This function declares that Emacs need not query the user if *process* is still running when Emacs is exited. The process will be deleted silently. The value is `t`.

```
(process-kill-without-query (get-process "shell"))
     ⇒ t
```

33.5 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. (Any processes listed as '`Exited`' or '`Signaled`' are actually eliminated after the listing is made.) This function 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-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-id *process* Function

This function returns the PID of *process*. This is an integer which distinguishes the process *process* from all other processes running on the same computer at the current time. The PID of a process is chosen by the operating system kernel when the process is started and remains constant as long as the process exists.

process-name *process* Function

This function returns the name of *process*.

process-status *process-name* Function

This function returns the status of *process-name* as a symbol. The argument *process-name* must be a process, a buffer, a process name (string) or a buffer name (string).

The possible values for an actual subprocess are:

`run` for a process that is running.

`stop` for a process that is stopped but continuable.

`exit` for a process that has exited.

`signal` for a process that has received a fatal signal.

`open` for a network connection that is open.

`closed` for a network connection that is closed. Once a connection is closed, you cannot reopen it, though you might be able to open a new connection to the same place.

`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`.

In earlier Emacs versions (prior to version 19), the status of a network connection was `run` if open, and `exit` if closed.

33.6 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, the subprocess will cease to read input correctly if you send an input line longer than the system can handle. You cannot avoid the problem by breaking the input into pieces and sending them separately, for the operating system will still have to put all the pieces together in the input buffer before it lets the subprocess read the line. The only solution is to put the input in a temporary file, and send the process a brief command to read that file.

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.

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.

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"
```

33.7 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. 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.

The set of signals and their names is defined by the operating system; Emacs has facilities for sending only a few of the signals that are defined. Emacs can send signals only to its own subprocesses.

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 which 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 33.3 [Asynchronous Processes], page 545.

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.

On systems with job control, the "stop character" (usually `C-z`) sends this signal (outside of Emacs). 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.

33.8 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.

33.8.1 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 33.8.2 [Filter Functions], page 553), its output is inserted in the associated buffer. The position to insert the output is determined by the `process-mark` (see Section 33.5 [Process Information], page 547), 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. If the process has no buffer and no filter function, its output is discarded.

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 marker which controls where additional output from the process will be inserted in the process buffer. When output is inserted, the marker is updated to point at the end of the output. This causes successive batches of output to be inserted consecutively.

If *process* does not insert its output into a buffer, then `process-mark` returns a marker that points nowhere.

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 is useful for distinguishing 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 will not be associated with any 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. (Presently, 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>
```

If the process's buffer is killed, the actual child process is killed with a `SIGHUP` signal (see Section 33.7 [Signals to Processes], page 550).

33.8.2 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, that would otherwise have been in a buffer, is passed to the filter. The process buffer is used for output from the process only when there is no filter.

A filter function must accept two arguments: the associated process and a string, which is the output. The function is then free to do whatever it chooses with the output.

A filter function 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 filters at random places in the middle of other Lisp programs. You may explicitly cause Emacs to wait, so that filter functions will run, by calling `sit-for`, `sleep-for` or `accept-process-output` (see Section 33.8.3 [Accepting Output], page 556). Emacs is also waiting when the command loop is reading input.

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 18.8 [Quitting], page 280.

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 use `unwind-protect` to make sure to restore the previous 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)
  (let ((old-buffer (current-buffer)))
    (unwind-protect
        (let (moving)
          (set-buffer (process-buffer proc))
          (setq moving (= (point) (process-mark proc)))
          (save-excursion
            ;; Insert the text, moving the process-marker.
            (goto-char (process-mark proc))
            (insert string)
            (set-marker (process-mark proc) (point)))
          (if moving (goto-char (process-mark proc))))
      (set-buffer old-buffer))))
```

The reason to use an explicit `unwind-protect` rather than letting `save-excursion` restore the current buffer is so as to preserve the change in point made by `goto-char`.

To make the filter force the process buffer to be visible whenever new text arrives, insert the following line just before the `unwind-protect`:

```
(display-buffer (process-buffer proc))
```

To force point to move to the end of the new output no matter where it was previously, eliminate the variable `moving` and call `goto-char` unconditionally.

All filter functions that do regexp searching or matching should save and restore the match data. Otherwise, a filter function that runs during a call to `sit-for` might clobber the match data of the program that called `sit-for`. See Section 30.5 [Match Data], page 517.

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.

set-process-filter *process filter* Function

This function gives *process* the filter function *filter*. If *filter* is `nil`, then the process will have 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
")
```

Here is another, more realistic example, which demonstrates how to use the process mark to do insertion in the same fashion as is done when there is no filter function:

```
;; Insert input in the buffer specified by my-shell-buffer
;;   and make sure that buffer is shown in some window.
(defun my-process-filter (proc str)
    (let ((cur (selected-window))
          (pop-up-windows t))
      (pop-to-buffer my-shell-buffer)
      (goto-char (point-max))
      (insert str)
      (set-marker (process-mark proc) (point-max))
      (select-window cur)))
```

33.8.3 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.

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.

33.9 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. You may explicitly cause Emacs to wait, so that sentinels will run, by calling `sit-for`, `sleep-for` or `accept-process-output` (see Section 33.8.3 [Accepting Output], page 556). Emacs is also waiting 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 18.8 [Quitting], page 280.

All sentinels that do regexp searching or matching should save and restore the match data. Otherwise, a sentinel that runs during a call to `sit-for` might clobber the match data of the program that called `sit-for`. See Section 30.5 [Match Data], page 517.

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.

33.10 Transaction Queues

You can use a *transaction queue* for more convenient communication with subprocesses 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 which 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 to match the entire answer; that's how `tq-enqueue` tells 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 33.8.2 [Filter Functions], page 553.

33.11 TCP

Emacs Lisp programs can open TCP 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. See Section 33.5 [Process Information], page 547.

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).

34 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 623, for related information. See also Chapter 35 [Display], page 581, for additional operating system status information pertaining to the terminal and the screen.

34.1 Starting Up Emacs

This section describes what Emacs does when it is started, and how you can customize these actions.

34.1.1 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 runs the normal hook `before-init-hook`.
2. It loads '`.emacs`' unless '`-q`' was specified on command line. (This is not done in '`-batch`' mode.) '`.emacs`' is found in the user's home directory; the '`-u`' option can specify the user name whose home directory should be used.
3. It loads '`default.el`' unless `inhibit-default-init` is non-`nil`. (This is not done in '`-batch`' mode or if '`-q`' was specified on command line.)
4. It runs the normal hook `after-init-hook`.
5. It loads the terminal-specific Lisp file, if any, except when in batch mode.
6. It runs `term-setup-hook`.
7. It runs `window-setup-hook`. See Section 35.15 [Window Systems], page 599.
8. It displays copyleft and nonwarranty, plus basic use information, unless the value of the variable `inhibit-startup-message` is non-`nil`.

   This display is also inhibited in batch mode, and if the current buffer is not '`*scratch*`'.
9. It processes any remaining command line arguments.

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.

34.1.2 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`' can be used to control 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*.

Emacs may also 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 13.1 [How Programs Do Loading], page 175). The Emacs distribution does not have any such file; you may create one at your site 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 will not subsequently load the '`default.el`' file.

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 14 [Byte Compilation], page 185), and make your '`.emacs`' file load the other file using `load` (see Chapter 13 [Loading], page 175).

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
after-init-hook Variable

These two normal hooks are run just before, and just after, loading of the user's init file or '`default.el`'.

34.1.3 Terminal-Specific Initialization

Each terminal type can have its own Lisp library that Emacs loads when run on that type of terminal. For a terminal type named *termtype*, the library is called '`term/`*termtype*'. Emacs finds the file by searching the `load-path` directories as it does for other files, and trying the '`.elc`' and '`.el`' suffixes. Normally, terminal-specific Lisp library is located in '`emacs/lisp/term`', a subdirectory of the '`emacs/lisp`' directory in which most Emacs Lisp libraries are kept.

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.

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 fully explain what should go in it. See Section 34.7 [Terminal Input], page 572.

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 very 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 20.4 [Hooks], page 332.

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 which Emacs runs after loading your '`.emacs`' file, the default initialization file (if any) and after loading terminal-specific Lisp files. arguments.

You can use `term-setup-hook` to override the definitions made by a terminal-specific file.

See `window-setup-hook` in Section 35.15 [Window Systems], page 599, for a related feature.

34.1.4 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.

command-line Function

This function parses the command line which Emacs was called with, processes it, loads the user's '`.emacs`' file and displays the initial nonwarranty information, etc.

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 arguments passed by the shell to Emacs, as a list of strings.

34.2 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.

34.2.1 Killing Emacs

Killing Emacs means ending the execution of the Emacs process. The parent process normally resumes control.

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.

kill-emacs &optional *no-query* Function

This function exits the Emacs process and kills it.

Normally, if there are modified files or if there are running processes, `kill-emacs` asks the user for confirmation before exiting. However, if *no-query* is supplied and non-`nil`, then Emacs exits without confirmation.

If *no-query* is an integer, then it is used as the exit status of the Emacs process. (This is useful primarily in batch operation; see Section 34.10 [Batch Mode], page 579.)

If *no-query* 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.

kill-emacs-hook Variable

This variable is a normal hook (a list of functions); the first thing `kill-emacs` does is to run this hook with `run-hooks`. That calls each of the functions in the list, with no arguments.

34.2.2 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 such as X, 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 an X client.

suspend-emacs *string* Function

This function stops Emacs and returns to the superior process. It returns `nil`.

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`. In Emacs version 18, `suspend-hook` was not a normal hook; its value was a single function, and if its value was non-`nil`, then `suspend-emacs` returned immediately without actually suspending anything.

After the user resumes Emacs, it runs the normal hook `suspend-resume-hook` using `run-hooks`. See Section 20.4 [Hooks], page 332.

The next redisplay after resumption will redraw the entire screen, unless the variable `no-redraw-on-reenter` is non-`nil` (see Section 35.1 [Refresh Screen], page 581).

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.

34.3 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-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 symbols for the operating systems that Emacs can run on up to version 19.1.

`aix-v3` AIX version 3.

`berkeley-unix`
Berkeley BSD 4.1, 4.2, or 4.3.

`hpux` Hewlett-Packard operating system, version 5, 6, or 7.

`irix` Silicon Graphics Irix system.

`rtu` RTU 3.0, UCB universe.

`unisoft-unix`
UniSoft's UniPlus 5.0 or 5.2.

`usg-unix-v`
AT&T's System V.0, System V Release 2.0, 2.2, or 3.

`vax-vms` VAX VMS version 4 or 5.

`xenix` SCO Xenix 386 Release 2.2.

We do not wish to add new symbols to make finer distinctions unless it is absolutely necessary! In fact, it would be nice to eliminate some of these alternatives in the future.

system-name Function

This function returns the name of the machine you are running on.

```
(system-name)
     ⇒ "prep.ai.mit.edu"
```

getenv *var* Function

This function returns the value of the environment variable *var*, as a string. If the variable `process-environment` specifies a value for *var*, that overrides the actual 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 works by modifying `process-environment`; binding that variable with `let` is also reasonable practice.

process-environment Variable

This variable is a list of strings to append to the environment of processes as they are created. Each string assigns a value to a shell environment variable. (This applies both to asynchronous and synchronous processes.) The function `getenv` also looks at 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")
```

load-average Function

This function returns the current 1 minute, 5 minute and 15 minute load averages in a list. The values are integers that are 100 times the system load averages. (The load averages indicate the number of processes trying to run.)

```
(load-average)
     ⇒ (169 48 36)

lewis@rocky[5] % uptime
 11:55am  up 1 day, 19:37,  3 users,
 load average: 1.69, 0.48, 0.36
```

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 success, `nil` if not.

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.

34.4 User Identification

user-login-name Function

This function returns the name under which the user is logged in. This is based on the effective UID, not the real UID.

```
(user-login-name)
     ⇒ "lewis"
```

user-real-login-name Function

This function returns the name under which the user logged in. This is based on the real UID, not the effective UID. This differs from `user-login-name` only when running with the setuid bit.

user-full-name Function

This function returns the full name of the user.

```
(user-full-name)
     ⇒ "Bil Lewis"
```

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.

34.5 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 humanly-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. However, it would be wise to count the characters from the beginning of the string rather than from the end, as additional information may 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 cons cell containing two integers, or 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 22.6.4 [File Attributes], page 357).

```
(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 22.6.4 [File Attributes], page 357.

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 below) and from `file-attributes` (see Section 22.6.4 [File Attributes], page 357).

34.6 Timers

You can set up a timer to call a function at a specified future time.

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; The form '*hour*`:`*min*`:`*sec timezone month*`/`*day*`/`*year*', where all fields are numbers, works; the format that `current-time-string` returns is also allowed.

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 an integer, 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 an integer, it specifies a repetition period measured in seconds.

cancel-timer *timer* Function

Cancel the requested action for *timer*, which should be a value previously returned by `run-at-time`. This cancels the effect of that call to `run-at-time`; the arrival of the specified time will not cause anything special to happen.

34.7 Terminal Input

This section describes functions and variables for recording or manipulating terminal input. See Chapter 35 [Display], page 581, for related functions.

34.7.1 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.

If *flow* is non-`nil`, then Emacs uses XON/XOFF (`C-q`, `C-s`) flow control for output to terminal. This has no effect except in CBREAK mode. See Section 34.9 [Flow Control], page 578.

The normal setting is system dependent. Some systems always use CBREAK mode regardless of what is specified.

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 using European 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 18.8 [Quitting], page 280.

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 has no effect unless *INTERRUPT* is non-`nil`.

META
: is non-`nil` if Emacs is paying attention to the eighth bit of input characters; if nil, Emacs clears the eighth bit of every input character.

QUIT
: is the character Emacs currently uses for quitting, usually `C-g`.

meta-flag Variable

This variable used to control whether to treat the 0200 bit in keyboard input as the `META` bit. `nil` meant no, and anything else meant yes. This variable existed in Emacs versions 18 and earlier but no longer exists in Emacs 19; use `set-input-mode` instead.

34.7.2 Translating Input Events

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 you use X windows, 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 must be a string or `nil`.

If `keyboard-translate-table` is a string, then each character read from the keyboard is looked up in this string and the character in the string is used instead. If the string is of length *n*, character codes *n* and up are untranslated.

In the example below, we set `keyboard-translate-table` to a string of 128 characters. 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 34.9 [Flow Control], page 578 for more information on this subject.)

```
(defun evade-flow-control ()
  "Replace C-s with C-\ and C-q with C-^."
  (interactive)
  (let ((the-table (make-string 128 0)))
    (let ((i 0))
      (while (< i 128)
        (aset the-table i i)
        (setq i (1+ i))))

    ;; Swap C-s and C-\.
    (aset the-table ?\034 ?\^s)
    (aset the-table ?\^s ?\034)
    ;; Swap C-q and C-^.
    (aset the-table ?\036 ?\^q)
    (aset the-table ?\^q ?\036)

    (setq keyboard-translate-table the-table)))
```

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 or enlarges the translate table if necessary.

function-key-map Variable

This variable holds a keymap which describes the character sequences sent by function keys on an ordinary character terminal. This keymap uses the data structure as other keymaps, but is used differently: it specifies translations to make while reading events.

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.

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 a number of 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 34.1.3 [Terminal-Specific], page 563.

Emacs versions 18 and earlier used totally different means of detecting the character sequences that represent function keys.

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.

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.

34.7.3 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 inputs, not counting keyboard macros. (Events from keyboard macros are excluded because they are less interesting for debugging; it should be enough to see the events which 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) are 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. The function always returns 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 34.8 [Terminal Output], page 576).

34.8 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 function returns the value of the variable `baud-rate`. In Emacs versions 18 and earlier, this was the only way to find out the terminal speed.

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 which are due to incorrect Termcap entries or to undesirable settings of terminal options more often than 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 34.7 [Terminal Input], page 572.

```
(open-termscript "../junk/termscript")
     ⇒ nil
```

34.9 Flow Control

This section attempts to answer the question "Why does Emacs choose to 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 was unobjectionable. Emacs, for economy of keystrokes and portability, used nearly all the ASCII control characters, with mnemonic meanings when possible; thus, C-s for search and C-q for quote.

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 did not pay attention. 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 users, who can turn flow control off, were unwilling 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 is a rhetorical (or religious) question; it 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 (which is very old), C-s and C-q were sent by the computer to turn the punch on and off!

GNU Emacs version 19 provides a convenient way of enabling flow control if you want it: call the function enable-flow-control.

enable-flow-control Function

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 34.7.2 [Translating Input], page 573).

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 kernel 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` were typed. 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. For example:

```
(setq keyboard-translate-table (make-string 128 0))
(let ((i 0))
  ;; Map most characters into themselves.
  (while (< i 128)
    (aset keyboard-translate-table i i)
    (setq i (1+ i))))
  ;; Map C-\ to C-s.
  (aset the-table ?\034 ?\^s)
  ;; Map C-^ to C-q.
  (aset the-table ?\036 ?\^q)))
```

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 34.8 [Terminal Output], page 576.

34.10 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 output 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.

35 Emacs Display

This chapter describes a number of features related to the display that Emacs presents to the user.

35.1 Refreshing the Screen

The function `redraw-frame` redisplays the entire contents of a given frame. See Chapter 26 [Frames], page 423.

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.

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. See Section 34.2.2 [Suspending Emacs], page 566.

no-redraw-on-reenter Variable

This variable controls whether Emacs redraws the entire screen after it has been suspended and resumed. Non-`nil` means yes, `nil` means no.

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.

35.2 Screen Size

The screen size functions report or tell Emacs the height or width of the terminal. When you are using multiple frames, they apply to the selected frame (see Chapter 26 [Frames], page 423).

screen-height Function

This function returns the number of lines on the screen that are available for display.

```
(screen-height)
     ⇒ 50
```

screen-width Function

This function returns the number of columns on the screen that are available for display.

```
(screen-width)
     ⇒ 80
```

set-screen-height *lines* &optional *not-actual-size* Function

This function declares that the terminal can display *lines* lines. The sizes of existing windows are altered proportionally to fit.

If *not-actual-size* is non-`nil`, then Emacs displays *lines* lines of output, but does not change its value for the actual height of the screen. (Knowing the correct actual size may be necessary for correct cursor positioning.) 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.

If *lines* is different from what it was previously, then the entire screen is cleared and redisplayed using the new size.

This function returns `nil`.

set-screen-width *columns* &optional *not-actual-size* Function

This function declares that the terminal can display *columns* columns. The details are as in `set-screen-height`.

35.3 Truncation

When a line of text extends beyond the right edge of a window, the line can either be truncated or continued on the next line. When a line is truncated, this is shown with a '`$`' in the rightmost column of the window. When a line is continued or "wrapped" onto the next line, this is shown with a '`\`' on the rightmost column of the window. The additional screen lines used to display a long text line are called *continuation* lines. (Note that wrapped lines are not filled; filling has nothing to do with continuation and truncation. See Section 29.11 [Filling], page 479.)

truncate-lines User Option

This buffer-local variable controls how Emacs displays lines that extend beyond the right edge of the window. If it is non-`nil`, then Emacs does not display continuation lines; rather each line of text occupies exactly one screen line, and a dollar sign appears at the edge of any line that extends to or beyond the edge of the window. The default is `nil`.

If the variable `truncate-partial-width-windows` is non-`nil`, then truncation is used for windows that are not the full width of the screen, regardless of the value of `truncate-lines`.

default-truncate-lines Variable

This variable is the default value for `truncate-lines` in buffers that do not have local values for it.

truncate-partial-width-windows User Option

This variable determines how lines that are too wide to fit on the screen are displayed in side-by-side windows (see Section 25.2 [Splitting Windows], page 398). If it is non-`nil`, then wide lines are truncated (with a '`$`' at the end of the line); otherwise they wrap to the next screen line (with a '`\`' at the end of the line).

You can override the images that indicate continuation or truncation with the display table; see Section 35.13 [Display Tables], page 596.

35.4 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 122.

You can write output in the echo area by using the Lisp printing functions with `t` as the stream (see Section 16.5 [Output Functions], page 227), or as follows:

message *string* &rest *arguments* Function

This function prints a one-line message in the echo area. The argument *string* is similar to a C language `printf` control string. See `format` in Section 4.5 [String Conversion], page 58, for the details on the conversion specifications. `message` returns the constructed string.

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."
```

```
---------- Echo Area ----------
Minibuffer depth is 0.
---------- Echo Area ----------
```

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.

35.5 Selective Display

Selective display is a class of minor modes in which specially marked lines do not appear on the screen, or in which highly indented lines do not appear.

The first variant, explicit selective display, is designed for use in a Lisp program. The program controls which lines are hidden by altering the text. Outline mode uses this variant. In the second variant, the choice of lines to hide is made automatically based on indentation. This variant is designed as a user-level feature.

The way you control explicit selective display is by replacing a newline (control-j) with a control-m. The text which 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 space. 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 which make parts of the text visible or invisible.

When you write a selectively displayed buffer into a file, all the control-m's are replaced by their original 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.

- If the value of `selective-display` is a positive integer, then lines that start with more than `selective-display` 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, what is shown is the *display* 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 35.13 [Display Tables], page 596.

35.6 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 as an arrow, or `nil` if the arrow feature is not in use.

overlay-arrow-position Variable

This variable holds a marker which indicates where to display the arrow. It should point at the beginning of a line. The arrow text is displayed 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 which this marker points into. Thus, only one buffer can have an overlay arrow at any given time.

35.7 Temporary Displays

Temporary displays are used by commands to put output into a buffer and then present it to the user for perusal rather than for editing. Many of the 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 output in the general sense of the word, are not affected). See Section 16.5 [Output Functions], page 227.

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

The value of this variable, if non-`nil`, is called as a function to display a help buffer. This variable is used by `with-output-to-temp-buffer`.

In Emacs versions 18 and earlier, this variable was called `temp-buffer-show-hook`.

momentary-string-display *string position* &optional *char message* Function

This function momentarily displays *string* in the current buffer at *position* (which is a character offset from the beginning of the buffer). The display remains until the next character is typed.

If the next character the user types is *char*, Emacs ignores it. Otherwise, that character 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 *message* is non-`nil`, it is displayed in the echo area while *string* is displayed in the buffer. If it is `nil`, then instructions to type *char* are displayed there, e.g., '`Type RET to continue editing`'.

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 ----------
```

This function works by actually changing the text in the buffer. As a result, if you later undo in this buffer, you will see the message come and go.

35.8 Overlays

You can use *overlays* to alter the appearance of a buffer's text on the screen. An overlay is an object which belongs to a particular buffer, and has a specified beginning and end. It also has properties which you can examine and set; these affect the display of the text within the overlay.

35.8.1 Overlay Properties

Overlay properties are like text properties in some respects, but the differences are more important than the similarities. 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.

`face`
: This property controls the font and color of text. See Section 35.9 [Faces], page 591, for more information. This feature is temporary; in the future, we may replace it with other ways of specifying how to display text.

`mouse-face`
: This property is used instead of `face` when the mouse is within the range of the overlay. This feature is not yet implemented, and may be temporary. It is documented here because we are likely to implement it this way at least for a while.

`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.

`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. This is not yet implemented, but will be.

`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. This is not yet implemented, but will be.

`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. Each function receives two arguments: the beginning and end of the part of the buffer being modified.

`insert-in-front-hooks`
This property's value is a list of functions to be called if text is inserted right at the beginning of the overlay.

`insert-behind-hooks`
This property's value is a list of functions to be called if text is inserted right at the end of the overlay.

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 *overlay* does not record any value for that property, then the value is `nil`.

overlay-put *overlay prop value* Function
This function set the value of property *prop* recorded in *overlay* to *value*. It returns *value*.

35.8.2 Managing Overlays

make-overlay *start end* &optional *buffer* Function
This function creates and returns an overlay which 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 return value is the overlay itself.

overlay-start *overlay* Function
This function returns the position at which *overlay* starts.

overlay-end *overlay* Function
This function returns the position at which *overlay* ends.

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 part of the buffer it belonged to, and ceases to have any effect on display.

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*.

next-overlay-change *pos* Function

This function returns the buffer position of the next beginning or end of an overlay, after *pos*.

35.9 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 id number* which distinguishes faces at low levels within Emacs. However, for most purposes, you can refer to faces in Lisp programs by their names.

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.

The face named `default` is used for ordinary text. The face named `modeline` is used for displaying the mode line and menu bars. The face named `region` is used for highlighting the region (in Transient Mark mode only).

35.9.1 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*, whose id number is zero, which is used for all text that doesn't somehow specify another face.
- With text properties. A character may have a `face` property; if so, it's displayed with that face. 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. See Section 29.17.4 [Special Properties], page 497.

- With overlays. An overlay may have `face` and `mouse-face` properties too; they apply to all the text covered by the overlay.
- With special glyphs. Each glyph can specify a particular face id number. See Section 35.13.3 [Glyphs], page 597.

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 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 35.8 [Overlays], page 588.

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.

35.9.2 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 a new face named *new-name* which is a copy of the existing face named *old-face*. If there is already a face named *new-name*, then it alters the face to have the same attributes as *old-face*.

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 that 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 (respectively, background) color of face *face* to *color*. The argument *color* color should be a string, the name of a color.

set-face-font *face font* &optional *frame* — Function

This function sets the font of face *face*. The argument *font* should be a string.

set-face-underline-p *face underline-p* &optional *frame* — Function

This function sets the underline attribute of face *face*.

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.

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 (respectively, background) color of face *face*. The argument *color* color should be a string, the name of a color.

face-font *face* &optional *frame* — Function

This function returns the name of the font of face *face*.

face-underline-p *face* &optional *frame* — Function

This function returns the underline attribute of face *face*.

face-id-number *face* — Function

This function returns the id number of face *face*.

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 id 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 28.6 [The Mark], page 457, for more information about Transient Mark mode.

Normally, the value is the id number of the face named `region`.

35.10 Blinking

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 char with close parenthesis syntax is inserted. The value of `blink-paren-function` may be `nil`, in which case nothing is done.

> **Please note:** this variable was named `blink-paren-hook` in older Emacs versions, but since it is not called with the standard convention for hooks, it was renamed to `blink-paren-function` in version 19.

blink-matching-paren — Variable

If this variable is `nil`, then `blink-matching-open` does nothing.

blink-matching-paren-distance — Variable

This variable specifies the maximum distance to scan for a matching parenthesis before giving up.

blink-matching-open — Function

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, then its context is shown by displaying it 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)))
```

35.11 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 (under X, this uses the face named `modeline`, which you can set as you wish). Otherwise, mode lines are displayed normally, just like text. The default is `t`.

35.12 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 35.13 [Display Tables], page 596). 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 is non-`nil`, these codes map to sequences of two glyphs, where the first glyph is the ASCII code for '^'. 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 code in octal.

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 the characters for which you want unusual 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.

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. This is the same as executing the following expression:

```
(default-value 'ctl-arrow)
```

See Section 10.9.3 [Default Value], page 147.

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 from the user-settable tab stops used by the command `tab-to-tab-stop`. See Section 29.14.5 [Indent Tabs], page 490.

35.13 Display Tables

You can use the *display table* feature to control how all 256 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*.

35.13.1 Display Table Format

A display table is actually an array of 261 elements.

make-display-table Function

This creates and returns a display table. The table initially has `nil` in all elements.

The first 256 elements correspond to character codes; the *n*th element says how to display the character code *n*. The value should be `nil` or a vector of glyph values (see Section 35.13.3 [Glyphs], page 597). If an element is `nil`, it says to display that character according to the usual display conventions (see Section 35.12 [Usual Display], page 595).

The remaining five elements of a display table serve special purposes, and `nil` means use the default stated below.

256 The glyph for the end of a truncated screen line (the default for this is '`$`'). See Section 35.13.3 [Glyphs], page 597.

257 The glyph for the end of a continued line (the default is '`\`').

258 The glyph for indicating a character displayed as an octal character code (the default is '`\`').

259 The glyph for indicating a control character (the default is '`^`').

260 A vector of glyphs for indicating the presence of invisible lines (the default is '`...`'). See Section 35.5 [Selective Display], page 584.

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 ?^ ??)))
```

35.13.2 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 local in all buffers; its value in a particular buffer is the display table for that buffer, or `nil` if the buffer does not have any assigned display table.

standard-display-table Variable

This variable's value is the default display table, used when neither the current buffer nor the window displaying it has an assigned display table. This variable is `nil` by default.

If neither the selected window nor the current buffer has a display table, and if the variable `standard-display-table` is `nil`, then Emacs uses the usual display conventions. See Section 35.12 [Usual Display], page 595.

35.13.3 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 *g*th 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:

integer
: Define this glyph code as an alias for code *integer*. This is used with X Windows to specify a face code.

string
: Send the characters in *string* to the terminal to output this glyph. This alternative is available on character terminals, but not under X.

`nil`
: This glyph is simple. On an ordinary terminal, the glyph code mod 256 is the character to output. With X, the glyph code mod 256 is character to output, and the glyph code divided by 256 specifies the *face id number* to use while outputting it. See Section 35.9 [Faces], page 591.

If a glyph code is greater than or equal to the length of the glyph table, that code is automatically simple.

35.13.4 ISO Latin 1

If you have a terminal that can handle the entire ISO Latin 1 character set, you can arrange to use that character set as follows:

```
(require 'disp-table)
;; Set char codes 160–255 to display as themselves.
;; (Codes 128–159 are the additional control characters.)
(standard-display-8bit 160 255)
```

If you are editing buffers written in the ISO Latin 1 character set and your terminal doesn't handle anything but ASCII, you can load the file '`iso-ascii`' to set up a display table which makes the other ISO characters display as sequences of ASCII characters. For example, the character "o with umlaut" displays as '`{"o}`'.

Some European countries have terminals that don't support ISO Latin 1 but do support the special characters for that country's language. You can define a display table to work one language using such terminals. For an example, see '`lisp/iso-swed.el`', which handles certain Swedish terminals.

You can load the appropriate display table for your terminal automatically by writing a terminal-specific Lisp file for the terminal type.

35.14 Beeping

You can make Emacs ring a 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 beeping alone when signaling an error is appropriate. (See Section 9.5.3 [Errors], page 122.)

ding &optional *dont-terminate* — Function

This function beeps, or flashes the screen (see `visible-bell` below). It also terminates any keyboard macro currently executing unless *dont-terminate* is non-`nil`.

beep &optional *dont-terminate* — Function

This is a synonym for `ding`.

visible-bell — Variable

This variable determines whether Emacs should flash the screen to represent a bell. Non-`nil` means yes, `nil` means no. This is effective only if the Termcap entry for the terminal in use has the visible bell flag ('`vb`') set.

35.15 Window Systems

Emacs works with several window systems, most notably the X Window Syste,. Note that 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. Its value should be a symbol such as `x` (if Emacs is running under X) or `nil` (if Emacs is running on an ordinary terminal).

window-system-version Variable

This variable distinguishes between different versions of the X Window System. Its value is 10 or 11 when using X; `nil` otherwise.

window-setup-hook Variable

This variable is a normal hook which Emacs runs after loading your '`.emacs`' file and the default initialization file (if any), after loading terminal-specific Lisp code, and after 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.

36 Customizing the Calendar and Diary

There are many customizations that you can use to make the calendar and diary suit your personal tastes.

36.1 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)
```

they display 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 place a plus sign ('`+`') beside any dates with diary entries. Whenever the calendar window is displayed or redisplayed, the diary entries are automatically marked for holidays.

Similarly, setting the variable `mark-holidays-in-calendar` to `t` places an asterisk ('`*`') after all holiday dates visible in the calendar window.

There are many customizations that you can make with the hooks provided. For example, the variable `calendar-load-hook`, whose default value is `nil`, is a normal hook run when the calendar package is first loaded (before actually starting to display the calendar).

The variable `initial-calendar-window-hook`, whose default value is `nil`, is a normal hook run the first time the calendar window is displayed. The function is invoked only when you first enter Calendar mode, not when you redisplay an existing Calendar window. But if you leave the calendar with the `q` command and reenter it, the hook runs again.

The variable `today-visible-calendar-hook`, whose default value is `nil`, 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; a function `calendar-star-date` is included for this purpose. In your '`.emacs`' file, put:

```
(setq today-visible-calendar-hook 'calendar-star-date)
```

Another standard hook function adds asterisks around the current date. Here's how to use it:

```
(setq today-visible-calendar-hook 'calendar-mark-today)
```

A corresponding variable, `today-invisible-calendar-hook`, whose default value is `nil`, is a normal hook run after the calendar buffer text has been prepared, if the current date is *not* visible in the window.

36.2 Customizing the Holidays

Emacs knows about holidays defined by entries on one of several lists. You can customize theses lists of holidays to your own needs, adding holidays or deleting lists of 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 consider all the holidays of these religions, 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 your 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). Holiday forms may have the following formats:

`(fixed` *month day string*`)`
: A fixed date on the Gregorian calendar. *month* and *day* are numbers, *string* is the name of the 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. *string* is the name of the holiday.

`(hebrew` *month day string*`)`
: A fixed date on the Hebrew calendar. *month* and *day* are numbers, *string* is the name of the holiday.

`(islamic` *month day string*`)`
: A fixed date on the Islamic calendar. *month* and *day* are numbers, *string* is the name of the holiday.

`(julian` *month day string*`)`
: A fixed date on the Julian calendar. *month* and *day* are numbers, *string* is the name of the holiday.

`(sexp` *sexp string*`)`
: *sexp* is a Lisp expression that should use the variable `year` to compute the date of a holiday, or `nil` if the holiday doesn't happen this year. The value represents the date as a list of the form `(`*month day year*`)`. *string* is the name of the holiday.

`(if` *boolean holiday-form* `&optional` *holiday-form*`)`
: A choice between two holidays based on the value of *boolean*.

`(`*function* `&optional` *args*`)`
: Dates requiring special computation; *args*, if any, are passed in a list to the function `calendar-holiday-function-`*function*.

For example, suppose you want to add Bastille Day, celebrated in France on July 14. You can do this by adding the following line to your '`.emacs`' file:

```
(setq other-holidays '((fixed 7 14 "Bastille Day")))
```

The holiday form `(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:

```
(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
      '((hebrew 10 2 "Last day of Hanukkah")
        (islamic 3 12 "Mohammed's Birthday")
        (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 the '`if`' or the '`sexp`' form. For example, American presidential elections occur on the first Tuesday after the first Monday in November of years divisible by 4:

```
(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 of the sun, for example, add `(eclipses)` to `other-holidays` and write an Emacs Lisp function `calendar-holiday-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") ... )
```

36.3 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 is a list of expressions that can involve the variables `month`, `day`, and `year`, all numbers in string form, and `monthname` and `dayname`, 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))
```

36.4 Time Display Format

In the calendar, diary, and related buffers, Emacs displays 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 "military" (European) style of writing times—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`, all numbers in string form, and `am-pm` and `time-zone`, both alphabetic strings. The default definition of `calendar-time-display-form` is as follows:

```
(12-hours ":" minutes am-pm
          (if time-zone " (") time-zone (if time-zone ")"))
```

Setting `calendar-time-display-form` to

```
(24-hours ":" minutes
          (if time-zone " (") time-zone (if time-zone ")"))
```

gives military-style times like '`21:07 (UT)`' if time zone names are defined, and times like '`21:07`' if they are not.

36.5 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 default starting and stopping dates for daylight savings time are the present-day American rules of the first Sunday in April until the last Sunday in October, but you can specify whatever rules you want by setting `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*`)`.

Emacs uses these expressions to determine the starting date of daylight savings time for the holiday list and for correcting times of day in the solar and lunar calculations.

The default value of `calendar-daylight-savings-starts` is this,

```
(calendar-nth-named-day 1 0 4 year)
```

which computes the first 0th day (Sunday) of the fourth month (April) in the year specified by `year`. If daylight savings time were changed to start on October 1, you would set `calendar-daylight-savings-starts` to

```
(list 10 1 year)
```

For a more complex example, suppose daylight savings time begins on the first of Nisan on the Hebrew calendar. You would set `calendar-daylight-savings-starts` to

```
    (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`.

36.6 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: 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 hidden.) 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 forms of dates recognized in the diary file. Each form is a list of regular expressions (see Section 30.2 [Regular Expressions], page 507) and the variables `month`, `day`, `year`, `monthname`, and `dayname`. The variable `monthname` matches the name of the month, capitalized or not, or its three-letter abbreviation, followed by a period or not; it matches '`*`'. Similarly, `dayname` matches the name of the day, capitalized or not, or its three-letter abbreviation, followed by a period or not. The variables month, day, and year match those numerical values, preceded by arbitrarily many zeros; they also match '*'. The default value of diary-date-forms in the American style is

```
((month "/" day "[^/0-9]")
 (month "/" day "/" year "[^0-9]")
 (monthname " *" day "[^,0-9]")
 (monthname " *" day ", *" year "[^0-9]")
 (dayname "\\W"))
```

Emacs matches of the diary entries with the date forms is done with the standard syntax table from Fundamental mode (see Chapter 31 [Syntax Tables], page 523), but with the '*' changed so that it is a word constituent.

The forms on the list must be *mutually exclusive* and must not match any portion of the diary entry itself, just the date. If, to be mutually exclusive, the pattern must match a portion of the diary entry itself, the first element of the form *must* be backup. This causes the date recognizer to back up to the beginning of the current word of the diary entry. Even if you use backup, the form 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 middle form because part of the diary entry must be matched to distinguish this form from the following one.

36.7 Hebrew- and Islamic-Date Diary Entries

Your diary file can have entries based on Hebrew or Islamic dates, as well as entries based on our usual Gregorian calendar. However, because the processing of such entries is time-consuming and most people don't need them, you must customize the processing of your diary file to specify that you want such entries recognized. If you want Hebrew-date diary entries, for example, you must include these lines in your '.emacs' file:

```
(setq nongregorian-diary-listing-hook 'list-hebrew-diary-entries)
(setq nongregorian-diary-marking-hook 'mark-hebrew-diary-entries)
```

If you want Islamic-date entries, include these lines in your '.emacs' file:

```
(setq nongregorian-diary-listing-hook 'list-islamic-diary-entries)
(setq nongregorian-diary-marking-hook 'mark-islamic-diary-entries)
```

If you want both Hebrew- and Islamic-date entries, include these lines:

```
(setq nongregorian-diary-listing-hook
      '(list-hebrew-diary-entries list-islamic-diary-entries))
(setq nongregorian-diary-marking-hook
      '(mark-hebrew-diary-entries mark-islamic-diary-entries))
```

Hebrew- and Islamic-date diary entries have the same formats as Gregorian-date diary entries, except that the date must be preceded with an 'H' for Hebrew dates and an 'I' for Islamic dates. 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

```
HHeshvan 25 Happy Hebrew birthday!
```

and would appear in the diary for any date that corresponds to Heshvan 25 on the Hebrew calendar. Similarly, an Islamic-date diary entry might be

```
IDhu al-Qada 25 Happy Islamic birthday!
```

and would appear in the diary for any date that corresponds to Dhu al-Qada 25 on the Islamic calendar.

As with Gregorian-date diary entries, Hebrew- and Islamic-date entries are nonmarking if they are preceded with an ampersand ('&').

There are commands to help you in making Hebrew- and Islamic-date entries to your diary:

`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`).

`i h y`
: Add a diary entry for the day of the Hebrew year corresponding to the selected date (`insert-yearly-hebrew-diary-entry`).

`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 exactly like the corresponding commands for ordinary diary entries: Move point to a date in the calendar window and the above commands insert the Hebrew or Islamic date (corresponding to the date indicated by point) at the end of your diary file and you can then type the diary entry. If you want the diary entry to be nonmarking, give a numeric argument to the command.

36.8 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 hides the irrelevant diary entries and then displays the buffer (`simple-diary-display`). However, if you specify the hook as follows,

```
(add-hook 'diary-display-hook 'fancy-diary-display)
```

then fancy mode displays diary entries and holidays by copying them into a special buffer that exists only for display. Copying 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. Add this line to your '`.emacs`' file:

```
(add-hook 'list-diary-entries-hook 'sort-diary-entries)
```

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.

36.9 Included Diary Files

If you use the fancy diary display, you can have diary entries from other files included with your own by an "include" mechanism. This facility makes possible the sharing of common diary files among groups of users. Lines in the diary file of this form:

`#include "`*filename*`"`

includes the diary entries from the file *filename* in the fancy diary buffer (because the ordinary diary buffer is just the buffer associated with your diary file, you cannot use the include mechanism unless you use the fancy diary buffer). The include mechanism is recursive, by the way, so that included files can include other files, and so on; you must be careful not to have a cycle of inclusions, of course. To enable the include facility, add lines as follows to your '.emacs' file:

```
(add-hook 'list-diary-entries-hook 'include-other-diary-files)
(add-hook 'mark-diary-entries-hook 'mark-included-diary-files)
```

36.10 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.

The generality of sexp diary entries lets you specify any diary entry that you can describe algorithmically. Suppose you get paid on the 21st of the month if it is a weekday, and to the Friday before if the 21st is on a weekend. The diary entry

```
&%%(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
```

applies to just those dates. This example illustrates how the sexp can depend on the variable `date`; this variable is a list (*month day year*) that gives the Gregorian date for which the diary entries are being found. If the value of the expression is `t`, the entry applies to that date. If the expression evaluates to `nil`, the entry does *not* apply to that date.

The following sexp diary entries take advantage of the ability (in the fancy diary display) to concoct diary entries 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.)

There are a number of other available sexp diary entries that are important to those who follow the Hebrew calendar:

`%%(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*.)

36.11 Customizing Appointment Reminders

You can specify exactly how Emacs reminds you of an appointment and how far in advance it begins doing so. Here are the variables that you can set:

`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 `t` (the default), Emacs rings the terminal bell for appointment reminders.

`appt-visible`
: If this is `t` (the default), Emacs displays the appointment message in echo area.

`appt-display-mode-line`
: If this is `t` (the default), Emacs displays the number of minutes to the appointment on the mode line.

`appt-msg-window`
: If this is `t` (the default), Emacs displays the appointment message in another window.

`appt-display-duration`
The number of seconds an appointment message is displayed. The default is 5 seconds.

Appendix A Tips and Standards

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.

A.1 Writing Clean Lisp Programs

Here are some tips for avoiding common errors in writing 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 `cadr`. Believe it or not, there is more than one plausible way to define `cadr`. Play it safe; append your name prefix to produce a name like `foo-cadr` or `mylib-cadr` instead.

  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 one file *foo* uses a macro defined in another file *bar*, *foo* should contain `(require '`*bar*`)` before the first use of the macro. (And *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 167.
- If you define a major mode, make sure to run a hook variable using `run-hooks`, just as the existing major modes do. See Section 20.4 [Hooks], page 332.
- 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 we cannot do without them.

  Instead, define sequences consisting of `C-c` followed by a non-letter. These sequences are reserved for major modes.

Changing all the major modes in Emacs 18 so they would follow this convention was a lot of work. Abandoning this convention would waste that work and inconvenience the users.

- It is a bad idea to define aliases for the Emacs primitives. Use the standard names instead.
- Redefining an Emacs primitive is an even worse idea. 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.
- If a file requires certain standard library programs to be loaded beforehand, then the comments at the beginning of the file should say so.
- 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 27.2.4 [Text Lines], page 443.
- Don't use functions that set the mark in your Lisp code (unless you are writing a command to set the mark). 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 28.6 [The Mark], page 457.

  In particular, don't use 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.
- The recommended way to print a message in the echo area is with the `message` function, not `princ`. See Section 35.4 [The Echo Area], page 583.
- 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 122.

  Do not use `message`, `throw`, `sleep-for`, or `beep` to report errors.
- Avoid using recursive edits. Instead, do what the Rmail `w` 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 18.10 [Recursive Editing], page 285.

- 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 library. (In fact, in Emacs names of this form are conventionally used for program-generated buffers.) The users will find Emacs more coherent if all libraries use the same conventions.
- 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 them in one or two significant places.
- Please put a copyright notice on the file if you give copies to anyone. Use the same lines that appear at the top of the Lisp files in Emacs itself. If you have not signed papers to assign the copyright to the Foundation, then place your name in the copyright notice in place of the Foundation's name.

A.2 Tips for Making Compiled Code Fast

Here are ways of improving the execution speed of byte-compiled lisp programs.

- Use the '`profile`' library to profile your program. See the file '`profile.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`, `assq` or `assoc` is even faster than explicit iteration. It may be worth rearranging a data structure so that one of these primitive search functions can be used.
- Certain built-in functions are handled specially by the byte compiler avoiding the need for an ordinary function call. It is a good idea to use these functions rather than alternatives. To see whether a function is handled specially by the compiler, examine its `byte-compile` property. If the property is non-`nil`, then the function is handled specially.

  For example, the following input will show you that `aref` is compiled specially (see Section 6.3 [Array Functions], page 90) while `elt` is not (see Section 6.1 [Sequence Functions], page 87):

  ```
  (get 'aref 'byte-compile)
       ⇒ byte-compile-two-args
  ```

```
(get 'elt 'byte-compile)
     ⇒ nil
```

- Make small functions inline, so that calls to them in compiled code run faster. See Section 11.9 [Inline Functions], page 163.

A.3 Tips for Documentation Strings

Here are some tips for the writing of documentation strings.

- Every command, function or variable intended for users to know about should have a documentation string.
- An internal subroutine of a Lisp program need not have a documentation string, and you can save space by using a comment instead.
- The first line of the documentation string should consist of one or two complete sentences which stand on their own as a summary. In particular, start the line with a capital letter and end with a period.

  The documentation string can have additional lines which 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.
- 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!
- A variable's documentation string should start with '*' if the variable is one that users would want to set interactively often. If the value is a long list, or a function, or if the variable would only be set in init files, then don't start the documentation string with '*'. See Section 10.5 [Defining Variables], page 135.
- 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 both that the variable only has two meaningfully distinct values and which value means "yes".

- 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`'.

  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 be variable.
- 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.
- 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 '`\\[forward-char]`'. When the documentation string is printed, Emacs will substitute whatever key is currently bound to `forward-char`. This will usually be '`C-f`', but if the user has moved key bindings, it will be the correct key for that user. See Section 21.3 [Keys in Documentation], page 339.
- 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.
- Don't use the term "Elisp", since that is or was a trademark. Use the term "Emacs Lisp".

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 inserted.

  (The 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 are used to 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)))
```

These comments are also written before a function definition to explain what the function does and how to call it properly.

'`;;;`' Comments that start with three semicolons, '`;;;`', should start at the left margin. Such comments are not used within function definitions, but are used to make more general comments. For example:

```
;;; This Lisp code is run in Emacs
;;; when it is to operate as a server
;;; for other processes.
```

'`;;;;`' 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 the number of semicolons. See section "Manipulating Comments" in *The GNU Emacs Manual*.

If you wish to "comment out" a number of lines of code, use triple semicolons at the beginnings of the lines.

Any character may be included in a comment, but it is advisable to precede a character with syntactic significance in Lisp (such as '`\`' or unpaired '`(`' or '`)`') with a '`\`', to prevent it from confusing the Emacs commands for editing Lisp.

A.5 Conventional Headers for Emacs Libraries

Emacs 19 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.
copying conditions...
```

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 `;;<TAB>`, 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. This field is important; it's how people will find your package when they're looking for things by topic area.

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.

'`;;; 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

The first step in building Emacs is to compile the C sources. This produces a program called '`temacs`', also called a *bare impure Emacs*. It contains the Emacs Lisp interpreter and I/O routines, but not the editing commands.

Then, to create a working Emacs editor, issue the '`temacs -l loadup`' command. This directs '`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 which is still impure but no longer bare.

It takes a long 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`' which has these files preloaded. '`emacs`' starts more quickly because it does not need to load the files. This is the program 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.

When the '`emacs`' executable is started, it automatically loads the user's '`.emacs`' file, or the default initialization file '`default.el`' if the user has none. (See Section 34.1 [Starting Up], page 561.) With the '`.emacs`' file, you can produce a version of Emacs that suits you and is not the same as the version other people use. With '`default.el`', you can customize Emacs for all the users at your site who don't choose to customize it for themselves. (For further reflection: why is this different from the case of the barber who shaves every man who doesn't shave himself?)

On some systems, dumping does not work. Then, you must start Emacs with the '`temacs -l loadup`' command each time you use it. This takes a long time, but since you need to start Emacs once a day at most—and once a week or less frequently if you never log out—the extra time is not too severe a problem.

Before '`emacs`' is dumped, the documentation strings for primitive and preloaded functions (and variables) need to be found in the file where they are stored. This is done by calling `Snarf-documentation` (see Section 21.2 [Accessing Documentation], page 336). These strings were moved out of '`emacs`' to make it smaller. See Section 21.1 [Documentation Basics], page 335.

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 use this function in an Emacs that was already dumped, you must set `command-line-processed` to `nil` first for good results. See Section 34.1.4 [Command Line Arguments], page 564.

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 18.36.1 of Fri Feb 27 1987 on slug
     (berkeley-unix)"
```

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

emacs-build-time Variable

The value of this variable is the time at which Emacs was built at the local site.

```
emacs-build-time
     ⇒ "Fri Feb 27 14:55:57 1987"
```

emacs-version Variable

The value of this variable is the version of Emacs being run. It is a string, e.g. `"18.36.1"`.

B.2 Pure Storage

There are two types of storage in GNU Emacs Lisp for user-created Lisp objects: *normal storage* and *pure storage*. Normal storage is where all the new data which is created 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 which 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 will have to increase the compilation parameter `PURESIZE` in the file 'config.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 symbols, or any other objects, but just returns them unchanged. It signals an error if asked to copy markers.

This function is used only while Emacs is being built and dumped; it is called only in the file '`emacs/lisp/loaddefs.el`'.

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). It is set to `nil` when Emacs is saved out as '`emacs`'. The flag is set and reset in the C sources.

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), then 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, buffers and certain other editing types, which are fairly large, are allocated in individual blocks, one per object, while 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 scanning all the objects that have been allocated and marking those that are still accessible to Lisp programs. To begin with, all the symbols, their values and associated function definitions, and any data presently on the stack, are accessible. Any objects which can be reached indirectly through other accessible objects are also accessible.

When this is finished, all inaccessible objects 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 notice. That is what the garbage collector arranges to do.

Unused cons cells are chained together onto a *free list* for future allocation; likewise for symbols and markers. The accessible strings are compacted so they are contiguous in memory; then the rest of the space formerly occupied by strings is made available to the string creation functions. Vectors, buffers, windows and other large objects are individually allocated and freed using `malloc`.

> **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-markers . free-markers)
 used-string-chars
 used-vector-slots
 (used-floats . free-floats))

(garbage-collect)
     ⇒ ((3435 . 2332) (1688 . 0)
           (57 . 417) 24510 3839 (4 . 1))
```

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-markers
: The number of markers in use.

free-markers
: The number of markers 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.

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 request 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 new 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 100,000. 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 which 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.

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 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, then return that value.\n\
The remaining args are not evalled at all.\n\
If all args return NIL, return NIL.")
  (args)
     Lisp_Object args;
{
  register Lisp_Object val;
  Lisp_Object args_left;
  struct gcpro gcpro1;

  if (NULL(args))
    return Qnil;

  args_left = args;
  GCPRO1 (args_left);
```

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

  UNGCPRO;
  return val;
}
```

Let's start with a precise explanation of the arguments to the `DEFUN` macro. Here are the general names for them:

`DEFUN (`*lname*`, `*fname*`, `*sname*`, `*min*`, `*max*`, `*interactive*`, `*doc*`)`

lname This is the name of the Lisp symbol to define with this function; 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. For `or`, no arguments are required.

max This is the maximum number of arguments that the function accepts. Alternatively, it can be `UNEVALLED`, indicating a special form that receives unevaluated arguments. A function with the equivalent of an `&rest` argument would have `MANY` in this position. Both `UNEVALLED` and `MANY` are macros. This argument must be one of these macros or a number at least as large as *min*. It may not be greater than six.

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 an interactive function taking no arguments.

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 list of argument names that every C function must have, followed by ordinary C declarations for them. Normally, all the arguments must be declared as `Lisp_Object`. If the function has no upper limit on the number of arguments in Lisp, then in C it receives two arguments: the number of Lisp arguments, and the address of a block containing their values. These 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 know that one pointer to the object is protected; 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 a string is moved, any pointers to it that the garbage collector does not know about will not be properly relocated. Therefore, all pointers to strings must be protected 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. There are also `GCPRO3` and `GCPRO4`.

In addition to using these macros, you must declare the local variables such as `gcpro1` which they implicitly use. If you protect two variables, with `GCPRO2`, you must declare `gcpro1` and `gcpro2`, as it uses them both. Alas, we can't explain all the tricky details here.

Defining the C function is not enough; you must also create the Lisp symbol for the primitive and store a suitable subr object in its function cell.

This is done by adding code to an initialization routine. The code looks like this:

```
defsubr (&subr-structure-name);
```

subr-structure-name is the name you used as the third argument to `DEFUN`.

If you are adding a 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 that function call to it. 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.

This function `syms_of_`*filename* is also the place to define any C variables which are to be visible as Lisp variables. `DEFVAR_LISP` is used to make a C variable of type `Lisp_Object` visible in Lisp. `DEFVAR_INT` is used to make a C variable of type `int` visible in Lisp with a value that is an integer.

Here is another function, with more complicated arguments. This comes from the code for the X Window System, and it demonstrates the use of macros and functions to manipulate Lisp objects.

```
DEFUN ("coordinates-in-window-p", Fcoordinates_in_window_p,
  Scoordinates_in_window_p, 2, 2,
  "xSpecify coordinate pair: \nXExpression which evals to window: ",
  "Return non-nil if POSITIONS is in WINDOW.\n\
  \(POSITIONS is a list, (SCREEN-X SCREEN-Y)\)\n\
  Returned value is list of positions expressed\n\
  relative to window upper left corner.")
  (coordinate, window)
     register Lisp_Object coordinate, window;
{
  register Lisp_Object xcoord, ycoord;

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

```
    if (XINT (ycoord) == (screen_height - 1))
      return Qnil;
    if ((XINT (ycoord) < XINT (XWINDOW (window)->top))
        || (XINT (ycoord) >= (XINT (XWINDOW (window)->top)
                              + XINT (XWINDOW (window)->height)) - 1))
      {
        return Qnil;
      }
    XFASTINT (ycoord) -= XFASTINT (XWINDOW (window)->top);
    return (Fcons (xcoord, Fcons (ycoord, Qnil)));
  }
```

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

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

B.5 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-four to twenty-six bits are used to address the object, and the remaining six to eight bits are used for a tag that identifies the object's type.

Because all access to data is through tagged pointers, it is always possible to determine the type of any object. This allows variables to be untyped, and the values assigned to them to be changed without regard to type. Function arguments also can be of any type; if you want a function to accept only a certain type of argument, you must check the type explicitly using a suitable predicate (see Section 2.5 [Type Predicates], page 34).

B.5.1 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 which names the buffer. It is guaranteed to be unique. See Section 24.2 [Buffer Names], page 383.

`save_modified`
: This field contains the time when the buffer was last saved, as an integer. See Section 24.4 [Buffer Modification], page 387.

`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 24.4 [Buffer Modification], page 387.

`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.

`undodata`
: This field points to the buffer's undo stack. See Section 29.9 [Undo], page 476.

`syntax_table_v`
: This field contains the syntax table for the buffer. See Chapter 31 [Syntax Tables], page 523.

`downcase_table`
: This field contains the conversion table for converting text to lower case. See Section 4.8 [Case Table], page 63.

`upcase_table`
: This field contains the conversion table for converting text to upper case. See Section 4.8 [Case Table], page 63.

`case_canon_table`
: This field contains the conversion table for canonicalizing text for case-folding search. See Section 4.8 [Case Table], page 63.

`case_eqv_table`
: This field contains the equivalence table for case-folding search. See Section 4.8 [Case Table], page 63.

`display_table`
: This field contains the buffer's display table, or `nil` if it doesn't have one. See Section 35.13 [Display Tables], page 596.

`markers`
: This field contains the chain of all markers that point into the buffer. At each deletion or motion of the buffer gap, all of these markers must be checked and perhaps updated. See Chapter 28 [Markers], page 453.

`backed_up`
: This field is a flag which 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 28.6 [The Mark], page 457.

`local_var_alist`
: This field contains the association list containing all of the variables local in this buffer, and their values. The function `buffer-local-variables` returns a copy of this list. See Section 10.9 [Buffer-Local Variables], page 143.

`mode_line_format`
: This field contains a Lisp object which controls how to display the mode line for this buffer. See Section 20.3 [Mode Line Format], page 326.

B.5.2 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.

`height` The height of the window, measured in lines.

`width` The width of the window, measured in columns.

`buffer` The buffer which 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.

`start` The position in the buffer which is the first character to be displayed in the window.

`pointm` This is the value of point in the current buffer when this window is selected; when it is not selected, it retains its previous value.

`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.)

`next` This is the window that is the next in the chain of siblings.

`prev` This is the window that is the previous in the chain of siblings.

`force_start`
: This is a flag which, if non-`nil`, says that the window has been scrolled explicitly by the Lisp program. At the next redisplay, if

point is off the screen, instead of scrolling the window to show the text around point, point will be moved to a location that is on the screen.

`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.

B.5.3 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.

`flags` A symbol indicating the state of the process. Possible values include `run`, `stop`, `closed`, etc.

`reason` An integer, the Unix signal number that the process received that caused the process to terminate or stop. If the process has exited, then this is the exit code it specified.

`mark` A marker indicating the position of end of last output from this process inserted into the buffer. This is usually the end of the buffer.

`kill_without_query`
A flag, non-`nil` meaning this process should not cause confirmation to be needed if Emacs is killed.

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 which 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`.

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 122, for an explanation of how errors are generated and handled.

symbol *string*; *reference.*

`error` `"error"`
See Section 9.5.3 [Errors], page 122.

`quit` `"Quit"`
See Section 18.8 [Quitting], page 280.

`args-out-of-range`
`"Args out of range"`
See Chapter 6 [Sequences Arrays Vectors], page 87.

`arith-error`
`"Arithmetic error"`
See `/` and `%` in Chapter 3 [Numbers], page 39.

`beginning-of-buffer`
`"Beginning of buffer"`
See Section 27.2 [Motion], page 440.

`buffer-read-only`
`"Buffer is read-only"`
See Section 24.6 [Read Only Buffers], page 389.

`end-of-buffer`
`"End of buffer"`
See Section 27.2 [Motion], page 440.

`end-of-file`
`"End of file during parsing"`
This is not a `file-error`.
See Section 16.3 [Input Functions], page 224.

`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 22 [Files], page 345.

`file-locked`
: This is a `file-error`.
See Section 22.5 [File Locks], page 352.

`file-already-exists`
: This is a `file-error`.
See Section 22.4 [Writing to Files], page 351.

`file-supersession`
: This is a `file-error`.
See Section 24.4 [Buffer Modification], page 387.

`invalid-function`
: `"Invalid function"`
See Section 8.2.3 [Classifying Lists], page 107.

`invalid-read-syntax`
: `"Invalid read syntax"`
See Section 16.3 [Input Functions], page 224.

`invalid-regexp`
: `"Invalid regexp"`
See Section 30.2 [Regular Expressions], page 507.

`no-catch` `"No catch for tag"`
See Section 9.5.1 [Catch and Throw], page 119.

`search-failed`
: `"Search failed"`
See Chapter 30 [Searching and Matching], page 505.

`setting-constant`
: `"Attempt to set a constant symbol"`
The values of the symbols `nil` and `t` may not be changed.
See Section 10.2 [Variables that Never Change], page 132.

`void-function`
: `"Symbol's function definition is void"`
See Section 11.8 [Function Cells], page 161.

`void-variable`
: `"Symbol's value as variable is void"`
See Section 10.6 [Accessing Variables], page 138.

`wrong-number-of-arguments`
: `"Wrong number of arguments"`
  See Section 8.2.3 [Classifying Lists], page 107.

`wrong-type-argument`
: `"Wrong type argument"`
  See Section 2.5 [Type Predicates], page 34.

Appendix D Buffer-Local Variables

The table below shows all of the variables that are automatically local (when set) in each buffer in Emacs Version 18 with the common packages loaded.

`abbrev-mode`
: see Chapter 32 [Abbrevs], page 535

`auto-fill-function`
: see Section 29.12 [Auto Filling], page 481

`buffer-auto-save-file-name`
: see Section 23.2 [Auto-Saving], page 378

`buffer-backed-up`
: see Section 23.1 [Backup Files], page 373

`buffer-display-table`
: see Section 35.13 [Display Tables], page 596

`buffer-file-name`
: see Section 24.3 [Buffer File Name], page 385

`buffer-file-number`
: see Section 24.3 [Buffer File Name], page 385

`buffer-file-truename`
: see Section 24.3 [Buffer File Name], page 385

`buffer-offer-save`
: see Section 22.2 [Saving Buffers], page 348

`buffer-read-only`
: see Section 24.6 [Read Only Buffers], page 389

`buffer-saved-size`
: see Section 27.1 [Point], page 439

`buffer-undo-list`
: see Section 29.9 [Undo], page 476

`case-fold-search`
: see Section 30.7 [Searching and Case], page 522

`ctl-arrow`
: see Section 35.12 [Usual Display], page 595

`default-directory`
: see Section 34.3 [System Environment], page 567

`fill-column`
: see Section 29.12 [Auto Filling], page 481

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 only loaded when their respective mode is used. This is not an exhaustive list.

Almost all of these maps are used as local maps. Indeed, of the modes that presently exist, only Vip mode and Terminal 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 in C mode as a local map.

`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-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.

`doctor-mode-map`
: A sparse keymap used by Doctor mode.

`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 in Emacs Lisp mode.

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

`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.

`lisp-interaction-mode-map`
: A sparse keymap used in Lisp mode.

`lisp-mode-map`
: A sparse keymap used in Lisp mode.

`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 in 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 which let you provide functions to be called from within Emacs on suitable occasions.

Most of these variables have names ending with '`-hook`' are *normal hooks*, that are run with `run-hooks`. The value of such a hook is a list of functions. The recommended way to put a new function on such a hook is to call `add-hook`. See Section 20.4 [Hooks], page 332, for more information about using hooks.

The variables whose names end in '`-function`' have single functions as their values. Usually there is a specific reason why the variable is not a normal hook, such as, the need to pass an argument to the function. (In older Emacs versions, some of these variables had names ending in '`-hook`' even though they were not normal hooks.)

The variables whose names end in '`-hooks`' have lists of functions as their values, but these functions are called in a special way (they are passed arguments, or else their values are used).

`activate-mark-hook`

`after-change-function`

`after-init-hook`

`auto-fill-function`

`auto-save-hook`

`before-change-function`

`before-init-hook`

`blink-paren-function`

`c-mode-hook`

`command-history-hook`

`comment-indent-function`

`deactivate-mark-hook`

`dired-mode-hook`

`disabled-command-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
LaTeX-mode-hook
ledit-mode-hook
lisp-indent-function
lisp-interaction-mode-hook
lisp-mode-hook
m2-mode-hook
mail-mode-hook
mail-setup-hook
medit-mode-hook
mh-compose-letter-hook
mh-folder-mode-hook
mh-letter-mode-hook
mim-mode-hook
news-mode-hook
news-reply-mode-hook
news-setup-hook
nroff-mode-hook
outline-mode-hook
plain-TeX-mode-hook
pre-abbrev-expand-hook
pre-command-hook
post-command-hook
prolog-mode-hook
protect-innocence-hook
rmail-edit-mode-hook
rmail-mode-hook
rmail-summary-mode-hook
scheme-indent-hook
scheme-mode-hook
scribe-mode-hook
```

```
shell-mode-hook
shell-set-directory-error-hook
suspend-hook
suspend-resume-hook
temp-buffer-show-function
term-setup-hook
terminal-mode-hook
terminal-mode-break-hook
TeX-mode-hook
text-mode-hook
vi-mode-hook
view-hook
window-setup-hook
write-contents-hooks
write-file-hooks
```

Appendix G Emacs 18 Antinews

For those users who live backwards in time, here is information about downgrading to Emacs version 18. We hope you will enjoy the greater simplicity that results from the absence of many Emacs 19 features.

G.1 Old Features in the Lisp Language

The following functions are missing or different in Emacs version 18.

- The functions `delete`, `member`, `indirect-function`, `map-y-or-n-p`, and `invocation-name` have been removed.
- The function `read` now skips a terminator character that terminates a symbol when reading from a buffer. Thus, if you use `read` on a buffer containing '`foo(bar)`' following point, it returns `foo` and leaves point after the open-parenthesis. This means there's no way you can properly read the list '`(bar)`', but that's the way the cookie crumbles.

  Because of this simplification, it's no longer necessary for an input stream function to accept an optional argument. In Emacs 18, an input stream is always called with no arguments, and should always return the next character of input.
- The function `documentation` takes just one argument; `documentation-property` takes just two.
- `random` no longer has the optional argument *n*.
- You can no longer arrange to run a hook if a particular Lisp library is loaded. The variable `after-load-alist` and the function `eval-after-load` have been removed.
- The function `autoload` no longer supports autoloading a keymap.
- "Magic" comments of the form '`;;;###autoload`' are now just comments. They don't do anything in particular except look pretty. If you want a function to be autoloaded by default, edit '`loaddefs.h`' by hand. What do you think editors are for?
- We took out the '`%S`' from the `format` function, and the optional argument *noescap* from `prin1-to-string`. We removed the `print-level` variable.

G.2 Compilation Features

- Inline functions are nonexistent in Emacs 18. We find they make the calling function unnecessarily large. (Small size is one of the features of Emacs 18.)

- We eliminated the two special forms, `eval-when-compile` and `eval-and-compile`, as well as the `compile-defun` command.
- When you load a Lisp file or library, you will no longer receive a warning if the directory contains both a '`.elc`' file and a new '`.el`' file that is newer. So be on your toes.
- We removed the special data type for byte-code functions. Compiled functions now work by means of an interpreted function which calls the function `bytecode`. That function runs the byte code interpreter.

G.3 Floating Point Numbers

Emacs 18 doesn't have or need floating point arithmetic built in. It has a handy Lisp program that allows you to emulate floating point. You'll have to write programs specially to use it, though.

As a result, certain macros, functions, and predicates no longer handle specifications for floating point numbers.

The function `string-to-number`, the predicate `floatp`, and the variable `float-output-format` have all been eliminated.

The functions `float`, `truncate`, `floor`, `ceil`, `round`, and `logb` do not exist; neither do the functions `abs`, `cos`, `sin`, `tan`, `acos`, `asin`, `atan`, `exp`, `expt`, `log10`, `log`, or `sqrt`.

The `format` function no longer handles the specifications '`%e`', '`%f`' and '`%g`' for printing floating point numbers; likewise for `message`.

G.4 Changes in Basic Editing Functions

- `kill-new` and `kill-append`, the primitives for putting text in the kill ring, have been eliminated.
- The variables `interprogram-paste-function` and `interprogram-cut-function` have been removed in Emacs 18.

  In addition, there's no need for `mark-active` and `deactivate-mark` because there is no Transient Mark mode. We also removed the hooks `activate-mark-hook` and `deactivate-mark-hook`.
- The `kill-region` function can no longer be used in read-only buffers. The `compare-buffer-substrings` and `current-kill` functions have been removed.
- The variable `overwrite-mode-binary` has been removed.
- The function `move-to-column` allows just one argument, *column*.
- The search functions now just return `t` when successful. This affects the functions `search-forward`, `search-backward`, `word-search-`

forward, word-search-backward, re-search-forward, and re-search-backward.

- When you do regular expression searching or matching, there is a fixed limit of ten '\(...\)' pairs that you can get information about with match-beginning and match-end. Moreover, save-match-data does not exist; you must use an explicit unwind-protect to save the match data.
- translate-region is gone.
- The variables before-change-function, after-change-function, and first-change-hook have been eliminated.
- The second argument to insert-abbrev-table-description is no longer optional.

G.5 Text Properties

We eliminated text properties.

G.6 Features for Files

Many file-related functions have been eliminated or simplified. Here is a basic listing of these functions.

The functions file-accessible-directory-p, file-truename, make-directory, delete-directory, set-visited-file-modtime, directory-abbrev-alist, abbreviate-file-name, write-region, write-contents-hooks, after-save-hook, set-default-file-modes, default-file-modes, and unix-sync have been eliminated.

We got rid of the "initial file name" argument to read-file-name.

Additionally, we removed the 12th element from the list returned by file-attributes.

directory-files always sorts the list of files. It's not user friendly to process the files in any haphazard order.

We eliminated the variables write-contents-hooks and local-write-file-hooks.

G.7 Making Certain File Names "Magic"

There are no more magic filenames. Sorry, but all the mana has been used up.

G.8 Frames

There is only one frame in Emacs 18, so all of the frame functions have been eliminated.

G.9 X Window System Features

We have simplified the way emacs and X interact by removing a great deal of creeping featurism.

- The functions `mouse-position` and `set-mouse-position`, and the special form `track-mouse` have been eliminated.
- Likewise, the functions `x-set-selection`, `x-set-cut-buffer`, `x-close-current-connection`, and `x-open-connection` have all been removed from Emacs Lisp 18.
- We removed a series of functions that gave information about the X server and the screen you were using; after all, the whole point of X is that all servers are equivalent. The names of the removed functions are: `x-display-screens`, `x-server-version`, `x-server-vendor`, `x-display-pixel-height`, `x-display-mm-height`, `x-display-pixel-width`, `x-display-mm-width`, `x-display-backing-store`, `x-display-save-under`, `x-display-planes`, `x-display-visual-class`, `x-display-color-p`, and `x-display-color-cells`.

  Additionally, we removed the variable `x-no-window-manager` and the functions `x-synchronize` and `x-get-resource`.

  We didn't abolish `x-display-color-p`, but we renamed it to `x-color-display-p`. We did abolish `x-color-defined-p`.
- `x-popup-menu` no longer accepts a keymap for its first argument.
- We removed both the function `x-rebind-key` and the related function `x-rebind-keys`.
- We abolished `x-parse-geometry`.

G.10 Window Actions that Were No Longer Useful

Various behaviors of windows in Emacs 19 were obsolete by the time Emacs 18 was due to come out. We have removed them. These changes are listed below.

- We removed the functions `window-at`, `window-minibuffer-p`, `set-window-dedicated-p`, `coordinates-in-window-p`, `walk-windows`, `window-dedicated-p`, and `window-end`.
- We removed the variables `pop-up-frames`, `pop-up-frame-function`, `display-buffer-function`, and `other-window-scroll-buffer`.

- The function `minibuffer-window` no longer accepts a frame as argument, since frames as objects do not exist in Emacs version 18. It returns the window used for minibuffers.
- The functions `next-window` and `previous-window` no longer accept the *all-frames* argument since there is just one frame.
- The functions `get-lru-window`, `get-largest-window`, `get-buffer-window`, and `get-buffer-window` also no longer take the optional argument *all-frames* because there is just one frame to search.

G.11 Display Features

- There are no overlays, and no faces.
- We eliminated the mode line spec '`%l`' that in later versions used to display the current line number. We removed the variables `line-number-mode` and `line-number-display-limit`.
- `baud-rate` is now a function rather than a variable.
- You can no longer call `message` with `nil` as the only argument; therefore, you can not reliably make the contents of the minibuffer visible.
- The variable `temp-buffer-show-function` has been renamed `temp-buffer-show-hook`.
- We removed the function `force-mode-line-update`. Use the following idiom instead:

  ```
  (set-buffer-modified-p (buffer-modified-p))
  ```
- Display tables no longer exist. We know what the ASCII characters should look like, and we made them look that way.

G.12 Working with Input Events

The big news about input events is that we got rid of function key and mouse events. Now the only input events are characters. What's more, these characters now have to be in the range of 0 to 127, optionally with a meta bit. This makes for big simplifications.

- Functions like `define-key`, `global-set-key`, `read-key-sequence`, and `local-set-key` used to accept strings or vectors in Emacs 19; now they only accept strings.
- The documentation functions (`single-key-description`, `key-description`, etc.) also no longer accept vectors, but they do accept strings.
- We removed the `read-event`, `event-start`, `posn-window`, `posn-point`, `posn-col-row`, `posn-timestamp`, `scroll-bar-scale`, and `event-end` functions, since they were only useful for non-character events.

- We removed the `unread-command-events` and `last-event-frame` variables.
- The functions `this-command-keys` and `recent-keys` now always return a string. Likewise, a keyboard macro's definition can only be a string, not a vector.
- We eliminated '`e`' as an interactive specification since it was useful only with non-character events.
- In Emacs 18, we represent Meta characters as character objects with the same encoding used in strings: 128 plus the corresponding non-Meta ASCII character.

G.13 Menus

You can no longer define menus as keymaps; good system design requires crafting a special-purpose interface for each facility, so it can precisely fit the requirements of that facility. We decided that unifying keymaps and menus was simply too much of a strain.

In Emacs 18, you can only activate menus with the mouse. Using them with a keyboard was too confusing for too many users.

Emacs 18 has no menu bars. All functions and variables related to the menu bar have been eliminated.

G.14 Changes in Minibuffer Features

The minibuffer history feature has been eliminated. Thus, we removed the optional argument *hist* from the minibuffer input functions `read-from-minibuffer` and `completing-read`.

The *initial* argument to `read-from-minibuffer` and other minibuffer input functions can no longer be a cons cell `(`*string* `.` *position*`)`.

In the function `read-no-blanks-input`, the *initial* argument is no longer optional.

G.15 New Features for Defining Commands

- The special meaning of '`@`' in an interactive specification has been eliminated.
- Emacs 18 does not support use of format-style '`%`'-sequences in the prompt strings in interactive specifications.
- The property `enable-recursive-minibuffers` no longer has any special meaning.

G.16 Removed Features for Reading Input

We removed the third argument (*meta*) from the function `set-input-mode`. Consequently, we added the variable `meta-flag`; set it to `t` to enable use of a Meta key, and to `nil` to disable it. (Those are the only two alternatives.)

We also removed the variable `extra-keyboard-modifiers`.

We removed the function `keyboard-translate` and the variables `num-input-keys` and `function-key-map`.

G.17 Removed Syntax Table Features

- We eliminated the functions `skip-syntax-forward`, `skip-syntax-backward`, `forward-comment`.
- We removed the syntax flag for "prefix syntax" and the flag for the alternate comment style. Emacs 18 supports only one style of comment in any given syntax table.
- We abolished the variable `words-include-escapes`.

G.18 The Case Table

Case tables do not exist in Emacs 18. Due to this change, we have removed the associated functions `set-standard-case-table`, `standard-case-table`, `current-case-table`, `set-case-table`, and `set-case-syntax-pair`.

G.19 Features for Dealing with Buffers

- We eliminated several functions for dealing with buffers: `buffer-modified-tick` and `generate-new-buffer-name`.
- We renamed `buffer-disable-undo` to `buffer-flush-undo`—a more picturesque name, you will agree.
- The function `other-buffer` takes just one argument in Emacs 18.
- The function `rename-buffer` now requires you to specify precisely the new name you want.
- We removed the local variable `list-buffers-directory`.
- We got rid of the hook `kill-buffer-hook`.

G.20 Local Variables Features

- The function `kill-all-local-variables` always eliminates all buffer-local variables of the current buffer. No more exceptions.
- Making a variable buffer-local when it is void now sets it to `nil`.
- We eliminated the functions `default-boundp`, because it is no longer possible for the default binding of a variable to be void.
- The special forms `defconst` and `defvar` now set the variable's local value rather than its default value when the variable is local in the current buffer.

G.21 Features for Subprocesses

`call-process` and `call-process-region` no longer indicate the termination status of the subprocess. We call on users to have faith that the subprocess executed properly.

The standard asynchronous subprocess features do not work on VMS; instead, special VMS asynchronous subprocess functions have been added. Since they are only for VMS, we can't be bothered documenting them; sorry. Use the source, Luke!

The function `signal-process` has been removed.

We eliminated the transaction queue feature, and the associated functions `tq-create`, `tq-enqueue`, and `tq-close`.

G.22 Dealing with Times And Time Delays

- We removed the functions `current-time`, `current-time-zone`, `run-at-time`, and `cancel-timer`.
- The function `current-time-string` no longer accepts any optional arguments.
- The functions `sit-for` and `sleep-for` no longer allow an optional argument to let you specify the time period in milliseconds; just in seconds. Additionally, we took out the optional third argument *nodisp* from `sit-for`.
- We removed the optional second and third arguments from the `accept-process-output` function. It accepts just one argument, the process.

G.23 Features not Available for Lisp Debuggers

- In Emacs 18, you can no longer specify to invoke the Lisp debugger only upon encountering certain types of errors. Any non-`nil` value for the variable `debug-on-error` says to invoke the debugger for any error whatever.
- We removed the variable `command-debug-status` and the function `backtrace-frame`.

G.24 Memory Allocation Changes

We removed the function `memory-limit`.

The list returned by `garbage-collect` no longer contains an element to describe floating point numbers, since there aren't any floating point numbers in Emacs 18.

G.25 Hook Changes

- We removed the hooks `pre-abbrev-expand-hook`, `pre-command-hook`, `post-command-hook`, and `auto-save-hook`.
- We removed the variable `revert-buffer-insert-file-contents-function`.
- We also removed the new function `add-hook`; you will have to set your hooks by hand. If you want to get really into the swing of things, set your hook variables the archaic way: store just one function rather than a list of functions. But that is optional.
- The variable `lisp-indent-hook` has been renamed to `lisp-indent-function`.
- The variable `auto-fill-function` has been renamed to `auto-fill-hook`.
- The `blink-paren-function` has been renamed to `blink-paren-hook`.
- The variable `temp-buffer-show-function` has been renamed to `temp-buffer-show-hook`.

Index

All variables, functions, keys, programs, files, and concepts are in this one index.

All names and concepts are permuted, so they appear several times, one for each permutation of the parts of the name. For example, `function-name` would appear as **function-name** and **name, function-**.

B

E

F

G

J

K

N

O

P

Q

R

S

T

U

V

W

Short Contents

Table of Contents