XEmacs Lisp Reference Manual

Version 3.4 (for XEmacs 21.1), May 1999

by Ben Wing

Based on the GNU Emacs Lisp Reference Manual
by Bil Lewis, Dan LaLiberte, Richard Stallman
and the GNU Manual Group

Version 3.3
Revised for XEmacs Versions 21.1,
April 1998.

Cover art by Etienne Suvasa.

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

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

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

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

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

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

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

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

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

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore,

by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

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

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

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

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

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

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

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

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

```
one line to give the program's name and 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 XEmacs text editor is written in the programming language called XEmacs Lisp. You can write new code in XEmacs Lisp and install it as an extension to the editor. However, XEmacs 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 XEmacs 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. XEmacs 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 XEmacs Lisp, presuming considerable familiarity with the use of XEmacs for editing. (See *The XEmacs Reference Manual*, for this basic information.) Generally speaking, the earlier chapters describe features of XEmacs Lisp that have counterparts in many programming languages, and later chapters describe features that are peculiar to XEmacs Lisp or relate specifically to editing.

This is edition 3.3.

1.1 Caveats

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

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

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

This manual was originally written for FSF Emacs 19 and was updated by Ben Wing (wing@666.com) for Lucid Emacs 19.10 and later for XEmacs 19.12, 19.13, 19.14, and 20.0. It was further updated by the XEmacs Development Team for 19.15, version 20 and 21. Please send comments and corrections relating to XEmacs-specific portions of this manual to

    xemacs@xemacs.org

or post to the newsgroup

`comp.emacs.xemacs`

–Ben Wing

1.2 Lisp History

Lisp (LISt Processing language) was first developed in the late 1950's 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 descendants of Maclisp came together and developed a standard for Lisp systems, called Common Lisp.

XEmacs 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 XEmacs. Sometimes the simplifications are so drastic that a Common Lisp user might be very confused. We will occasionally point out how XEmacs 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 Lisp objects, and vice versa. See Section 2.1 [Printed Representation], page 15, 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` has three separate meanings: it is a symbol with the name '`nil`'; it is the logical truth value *false*; and it is the empty list—the list of zero elements. When used as a variable, `nil` always has the value `nil`.

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

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

```
(cons 'foo ())                ; Emphasize the empty list
(not nil)                     ; Emphasize the truth value false
```

In contexts where a truth value is expected, any non-`nil` value is considered to be *true*. However, `t` is the preferred way to represent the truth value *true*. When you need to choose a value which represents *true*, and there is no other basis for choosing, use `t`. The symbol `t` always has value `t`.

In XEmacs 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 151.

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.

```
(news-cadr '(a b c))
     ↦ (car (cdr '(a b c)))
     ⇒ b
```

Sometimes to help describe one form we show another form that produces identical results. The exact equivalence of two forms is indicated with '≡'.

```
(cons 'a nil) ≡ (list 'a)
```

1.3.4 Printing Notation

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

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

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

1.3.5 Error Messages

Some examples signal errors. This normally displays an error message in the echo area. We show the error message on a line starting with '$\boxed{\text{error}}$'. 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. These examples show the contents of the buffer in question between two lines of dashes containing the buffer name. In addition, '⋆' indicates the location of point. (The symbol for point, of course, is not part of the text in the buffer; it indicates the place *between* two characters where point is 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 [Lisp Data Types], page 15, for a list of XEmacs 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 164, 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 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 based on the GNU Emacs Lisp Reference Manual, version 2.4, 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.

Ben Wing adapted this manual for XEmacs 19.14 and 20.0, and earlier for Lucid Emacs 19.10, XEmacs 19.12, and XEmacs 19.13. He is the sole author of many of the manual sections, in particular the XEmacs-specific sections: events, faces, extents, glyphs, specifiers, toolbar, menubars, scrollbars, dialog boxes, devices, consoles, hash tables, range tables, char tables, databases, and others. The section on annotations was originally written by Chuck Thompson. Corrections to v3.1 and later were done by Martin Buchholz, Steve Baur, and Hrvoje Niksic.

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

2 Lisp Data Types

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

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

A few fundamental object types are built into XEmacs. 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*, *character* (starting with XEmacs 20.0), *float*, *cons*, *symbol*, *string*, *vector*, *bit-vector*, *subr*, *compiled-function*, *hashtable*, *range-table*, *char-table*, *weak-list*, and several special types, such as *buffer*, that are related to editing. (See Section 2.5 [Editing Types], page 29.)

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

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

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

This chapter describes the purpose, printed representation, and read syntax of each of the standard types in Emacs Lisp. Details on how to use these types can be found in later chapters.

2.1 Printed Representation and Read Syntax

The *printed representation* of an object is the format of the output generated by the Lisp printer (the function `prin1`) for that object. The *read syntax* of an object is the format of the input accepted by the Lisp reader (the function `read`) for that object. 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 it encounters '`#<`'.

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

When you evaluate an expression interactively, the Lisp interpreter first reads the textual representation of it, producing a Lisp object, and then evaluates that object (see Chapter 8 [Evaluation], page 119). However, evaluation and reading are separate activities. Reading returns the Lisp object represented by the text that is read; the object may or may not be evaluated later. See Section 17.3 [Input Functions], page 253, for a description of `read`, the basic function for reading objects.

2.2 Comments

A *comment* is text that is written in a program only for the sake of humans that read the program, and that has no effect on the meaning of the program. In Lisp, a semicolon (‘;’) starts a comment if it is not within a string or character constant. The comment continues to the end of line. The Lisp reader discards comments; they do not become part of the Lisp objects which represent the program within the Lisp system.

The ‘#@*count*’ construct, which skips the next *count* characters, is useful for program-generated comments containing binary data. The XEmacs Lisp byte compiler uses this in its output files (see Chapter 15 [Byte Compilation], page 205). It isn't meant for source files, however.

See Section A.4 [Comment Tips], page 762, for conventions for formatting comments.

2.3 Primitive Types

For reference, here is a list of all the primitive types that may exist in XEmacs. Note that some of these types may not exist in some XEmacs executables; that depends on the options that XEmacs was configured with.

- bit-vector
- buffer
- char-table
- character
- charset
- coding-system
- cons
- color-instance
- compiled-function
- console
- database
- device
- event
- extent
- face

- float
- font-instance
- frame
- glyph
- hashtable
- image-instance
- integer
- keymap
- marker
- process
- range-table
- specifier
- string
- subr
- subwindow
- symbol
- toolbar-button
- tooltalk-message
- tooltalk-pattern
- vector
- weak-list
- window
- window-configuration
- x-resource

In addition, the following special types are created internally but will never be seen by Lisp code. You may encounter them, however, if you are debugging XEmacs. The printed representation of these objects begins '`#<INTERNAL EMACS BUG`', which indicates to the Lisp programmer that he has found an internal bug in XEmacs if he ever encounters any of these objects.

- char-table-entry
- command-builder
- extent-auxiliary
- extent-info
- lcrecord-list
- lstream
- opaque
- opaque-list
- popup-data
- symbol-value-buffer-local

- symbol-value-forward
- symbol-value-lisp-magic
- symbol-value-varalias
- toolbar-data

2.4 Programming Types

There are two general categories of types in XEmacs Lisp: those having to do with Lisp programming, and those having to do with editing. The former exist in many Lisp implementations, in one form or another. The latter are unique to XEmacs Lisp.

2.4.1 Integer Type

The range of values for integers in XEmacs Lisp is -134217728 to 134217727 (28 bits; i.e., -2^{27} to $2^{28}-1$) on most machines. (Some machines, in particular 64-bit machines such as the DEC Alpha, may provide a wider range.) It is important to note that the XEmacs Lisp arithmetic functions do not check for overflow. Thus `(1+ 134217727)` is -134217728 on most machines. (However, you *will* get an error if you attempt to read an out-of-range number using the Lisp reader.)

The read syntax for integers is a sequence of (base ten) digits with an optional sign at the beginning. (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.
268435457        ; Causes an error on a 28-bit implementation.
```

See Chapter 3 [Numbers], page 45, for more information.

2.4.2 Floating Point Type

XEmacs supports floating point numbers. 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 45, for more information.

2.4.3 Character Type

In XEmacs version 19, and in all versions of FSF GNU Emacs, a *character* in XEmacs Lisp is nothing more than an integer. This is yet another holdover from XEmacs Lisp's derivation from vintage-1980 Lisps; modern versions of Lisp consider this equivalence a bad idea, and have separate character types. In XEmacs version 20, the modern convention is followed, and characters are their own primitive types. (This change was necessary in order for MULE, i.e. Asian-language, support to be correctly implemented.)

Even in XEmacs version 20, remnants of the equivalence between characters and integers still exist; this is termed the *char-int confoundance disease*. In particular, many functions such as `eq`, `equal`, and `memq` have equivalent functions (`old-eq`, `old-equal`, `old-memq`, etc.) that pretend like characters are integers are the same. Byte code compiled under any version 19 Emacs will have all such functions mapped to their `old-` equivalents when the byte code is read into XEmacs 20. This is to preserve compatibility – Emacs 19 converts all constant characters to the equivalent integer during byte-compilation, and thus there is no other way to preserve byte-code compatibility even if the code has specifically been written with the distinction between characters and integers in mind.

Every character has an equivalent integer, called the *character code*. For example, the character *A* is represented as the integer 65, following the standard ASCII representation of characters. If XEmacs was not compiled with MULE support, the range of this integer will always be 0 to 255 – eight bits, or one byte. (Integers outside this range are accepted but silently truncated; however, you should most decidedly *not* rely on this, because it will not work under XEmacs with MULE support.) When MULE support is present, the range of character codes is much larger. (Currently, 19 bits are used.)

FSF GNU Emacs uses kludgy character codes above 255 to represent keyboard input of ASCII characters in combination with certain modifiers. XEmacs does not use this (a more general mechanism is used that does not distinguish between ASCII keys and other keys), so you will never find character codes above 255 in a non-MULE XEmacs.

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 2.4.8 [String Type], page 25.

The read syntax for characters begins with a question mark, followed by the character (if it's printable) or some symbolic representation of it. In XEmacs 20, where characters are their own type, this is also the print representation. In XEmacs 19, however, where 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 XEmacs Lisp provides for characters.

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:

```
;; Under XEmacs 20:
?Q ⇒ ?Q    ?q ⇒ ?q
(char-int ?Q) ⇒ 81
;; Under XEmacs 19:
?Q ⇒ 81     ?q ⇒ 113
```

You can use the same syntax for punctuation characters, but it is often a good idea to add a '`\`' so that the Emacs commands for editing Lisp code don't get confused. For example, '`?\ `' is the way to write the space character. If the character is '`\`', you *must* use a second '`\`' to quote it: '`?\\`'. XEmacs 20 always prints punctuation characters with a '`\`' in front of them, to avoid confusion.

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. Their character codes are 7, 8, 9, 10, 11, 12, 13, and 27 in decimal. Thus,

```
;; Under XEmacs 20:
?\a ⇒ ?\^G                ; C-g
(char-int ?\a) ⇒ 7
?\b ⇒ ?\^H                ; backspace, ⟨BS⟩, C-h
(char-int ?\b) ⇒ 8
?\t ⇒ ?\t                 ; tab, ⟨TAB⟩, C-i
(char-int ?\t) ⇒ 9
?\n ⇒ ?\n                 ; newline, ⟨LFD⟩, C-j
?\v ⇒ ?\^K                ; vertical tab, C-k
?\f ⇒ ?\^L                ; formfeed character, C-l
?\r ⇒ ?\r                 ; carriage return, ⟨RET⟩, C-m
?\e ⇒ ?\^[                ; escape character, ⟨ESC⟩, C-[
?\\ ⇒ ?\\                 ; backslash character, \
;; Under XEmacs 19:
?\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; this usage has nothing to do with the character ⟨ESC⟩.

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

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

```
;; Under XEmacs 20:
?\^I ⇒ ?\t   ?\C-I ⇒ ?\t
(char-int ?\^I) ⇒ 9
;; Under XEmacs 19:
?\^I ⇒ 9     ?\C-I ⇒ 9
```

There is also a character read syntax beginning with '\M-'. This sets the high bit of the character code (same as adding 128 to the character code). For example, '?\M-A' stands for the character with character code 193, or 128 plus 65. You should *not* use this syntax in your programs. It is a holdover of yet another confoundance disease from earlier Emacsen. (This was used to represent keyboard input with the ⟨META⟩ key set, thus the 'M'; however, it conflicts with the legitimate ISO-8859-1 interpretation of the character code. For example, character code 193 is a lowercase 'a' with an acute accent, in ISO-8859-1.)

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.

```
;; Under XEmacs 20:
?\012 ⇒ ?\n          ?\n ⇒ ?\n          ?\C-j ⇒ ?\n
?\101 ⇒ ?A           ?A ⇒ ?A
;; Under XEmacs 19:
?\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, '?\+' is equivalent to '?+'. There is no reason to add a backslash before most characters. However, you should add a backslash before any of the characters '()\|;'`"#.,' to avoid confusing the Emacs commands for editing Lisp code. Also add a backslash before whitespace characters such as space, tab, newline and formfeed. However, it is cleaner to use one of the easily readable escape sequences, such as '\t', instead of an actual whitespace character such as a tab.

2.4.4 Symbol Type

A *symbol* in XEmacs Lisp is an object with a name. The symbol name serves as the printed representation of the symbol. In ordinary use, the name is unique—no two symbols have the same name.

A symbol can serve as a variable, as a function name, or to hold a property list. Or it may serve only to be distinct from all other Lisp objects, so that its presence in a data structure may be recognized reliably. In a given context, usually only one of these uses is intended. But you can use one symbol in all of these ways, independently.

A symbol name can contain any characters whatever. Most symbol names are written with letters, digits, and the punctuation characters '-+=*/'. Such names require no special punctuation; the characters of the name suffice as long as the name does not look like a number. (If it does, write a '\' at the beginning of the name to force interpretation as a symbol.) The characters '_~!@$%^&:<>{}' are less often used but also require no special punctuation. Any other characters may be included in a symbol's name by escaping them with a backslash. In contrast to its use in strings, however, a backslash in the name of a symbol simply quotes the single character that follows the backslash. For example, in a string, '\t' represents a tab character; in the name of a symbol, however, '\t' merely quotes the letter *t*. To have a symbol with a tab character in its name, you must actually use a tab (preceded with a backslash). But it's rare to do such a thing.

> **Common Lisp note:** In Common Lisp, lower case letters are always "folded" to upper case, unless they are explicitly escaped. In Emacs Lisp, upper case and lower case letters are distinct.

Here are several examples of symbol names. Note that the '+' in the fifth example is escaped to prevent it from being read as a number. This is not necessary in the sixth example because the rest of the name makes it invalid as a number.

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

2.4.5 Sequence Types

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

Arrays are further subdivided into strings, vectors, and bit vectors. Vectors can hold elements of any type, but string elements must be characters, and bit vector elements must be either 0 or 1. However, the characters in a string can have extents (see Chapter 40 [Extents], page 585) and text properties (see Section 36.18 [Text Properties], page 539) like characters in a buffer; vectors do not support extents or text properties even when their elements happen to be characters.

Lists, strings, vectors, and bit 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 101.

It is impossible to read the same sequence twice, since sequences are always created anew upon reading. If you read the read syntax for a sequence twice, you get two sequences with equal contents. There is one exception: the empty list `()` always stands for the same object, `nil`.

2.4.6 Cons Cell and List Types

A *cons cell* is an object comprising two pointers named the CAR and the CDR. Each of them can point to any Lisp object.

A *list* is a series of cons cells, linked together so that the CDR of each cons cell points either to another cons cell or to the empty list. See Chapter 5 [Lists], page 77, for functions that work on lists. Because most cons cells are used as part of lists, the phrase *list structure* has come to refer to any structure made out of cons cells.

The names CAR and CDR 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, each object inside the parentheses becomes 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 of the list, 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 by both humans and computers, the box illustrations can be understood only by humans.) The following represents the three-element list `(rose violet buttercup)`:

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

In this 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
```

2.4.6.1 Dotted Pair Notation

Dotted pair notation is an alternative syntax for cons cells that represents the CAR and CDR explicitly. In this syntax, `(`*a* `.` *b*`)` stands for a cons cell whose CAR is the object *a*, and whose CDR is the object *b*. Dotted pair notation is therefore more general than list syntax. In the dotted pair notation, the list '`(1 2 3)`' is written as '`(1 . (2 . (3 . nil)))`'. For `nil`-terminated lists, 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.

Here's how box notation can illustrate dotted pairs. This example shows the pair `(rose . violet)`:

```
  --- ---
 |___|___|--> 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 why `(rose . violet . buttercup)` is invalid syntax; it would require a cons cell that has 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)))`.

2.4.6.2 Association List Type

An *association list* or *alist* is a specially-constructed list whose elements are cons cells. In each element, the CAR is considered a *key*, and the CDR is considered an *associated value*.

(In some cases, the associated value is stored in the CAR of the CDR.) Association lists are often used as stacks, since it is easy to add or remove associations at the front of the list.

For example,

```
(setq alist-of-colors
      '((rose . red) (lily . white)  (buttercup . yellow)))
```

sets the variable `alist-of-colors` to an alist of three elements. In the first element, `rose` is the key and `red` is the value.

See Section 5.8 [Association Lists], page 92, for a further explanation of alists and for functions that work on alists.

2.4.7 Array Type

An *array* is composed of an arbitrary number of slots for referring to other Lisp objects, arranged in a contiguous block of memory. Accessing any element of an array takes the same amount of time. In contrast, accessing an element of a list requires time proportional to the position of the element in the list. (Elements at the end of a list take longer to access than elements at the beginning of a list.)

XEmacs defines three types of array, strings, vectors, and bit vectors. A string is an array of characters, a vector is an array of arbitrary objects, and a bit vector is an array of 1's and 0's. All are one-dimensional. (Most other programming languages support multidimensional arrays, but they are not essential; you can get the same effect with an array of arrays.) Each type of array has its own read syntax; see Section 2.4.8 [String Type], page 25, Section 2.4.9 [Vector Type], page 26, and Section 2.4.10 [Bit Vector Type], page 26.

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 the string type, the vector type, and the bit vector type.

2.4.8 String Type

A *string* is an array of characters. Strings are used for many purposes in XEmacs, 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 or *C-a* using the convenient escape sequences, like this: `"\t, \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 2.4.3 [Character Type], page 18, for a description of the read syntax for characters.)

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

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

```
"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."
```

A string can hold extents and 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 extents and properties with no special effort. See Chapter 40 [Extents], page 585, See Section 36.18 [Text Properties], page 539.

Note that FSF GNU Emacs has a special read and print syntax for strings with text properties, but XEmacs does not currently implement this. It was judged better not to include this in XEmacs because it entails that `equal` return `nil` when passed a string with text properties and the equivalent string without text properties, which is often counter-intuitive.

See Chapter 4 [Strings and Characters], page 59, for functions that work on strings.

2.4.9 Vector Type

A *vector* is a one-dimensional array of elements of any type. It takes a constant amount of time to access any element of a vector. (In a list, the access time of an element is proportional to the distance of the element from the beginning of the list.)

The printed representation of a vector consists of a left square bracket, the elements, and a right square bracket. This is also the read syntax. Like numbers and strings, vectors are considered constants for evaluation.

```
[1 "two" (three)]      ; A vector of three elements.
     ⇒ [1 "two" (three)]
```

See Section 6.4 [Vectors], page 106, for functions that work with vectors.

2.4.10 Bit Vector Type

A *bit vector* is a one-dimensional array of 1's and 0's. It takes a constant amount of time to access any element of a bit vector, as for vectors. Bit vectors have an extremely compact internal representation (one machine bit per element), which makes them ideal for keeping track of unordered sets, large collections of boolean values, etc.

The printed representation of a bit vector consists of '`#*`' followed by the bits in the vector. This is also the read syntax. Like numbers, strings, and vectors, bit vectors are considered constants for evaluation.

```
#*00101000         ; A bit vector of eight elements.
     ⇒ #*00101000
```

See Section 6.6 [Bit Vectors], page 107, for functions that work with bit vectors.

2.4.11 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 164).

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 172). A named function in Lisp is actually a symbol with a valid function in its function cell (see Section 11.4 [Defining Functions], page 168).

Most of the time, functions are called when their names are written in Lisp expressions in Lisp programs. However, you can construct or obtain a function object at run time and then call it with the primitive functions `funcall` and `apply`. See Section 11.5 [Calling Functions], page 169.

2.4.12 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 XEmacs is concerned. See Chapter 12 [Macros], page 177, for an explanation of how to write a macro.

2.4.13 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 125).

It does not matter to the caller of a function whether the function is primitive. However, this does matter if you try 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. Calls to the redefined function from Lisp will use the new definition, but calls from C code may still use the built-in definition.

The term *function* refers to all Emacs functions, whether written in Lisp or C. See Section 2.4.11 [Function Type], page 27, 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.
```

2.4.14 Compiled-Function Type

The byte compiler produces *compiled-function objects*. The evaluator handles this data type specially when it appears as a function to be called. See Chapter 15 [Byte Compilation], page 205, for information about the byte compiler.

The printed representation for a compiled-function object is normally '`#<compiled-function...>`'. If `print-readably` is true, however, it is '`#[...]`'.

2.4.15 Autoload Type

An *autoload object* is a list whose first element is the symbol `autoload`. It is stored as the function definition of a symbol as a placeholder for the real definition; it says that the real definition is found in a file of Lisp code that should be loaded when necessary. The autoload object contains the name of the file, plus some other information about the real definition.

After the file has been loaded, the symbol should have a new function definition that is not an autoload object. The new definition is then called as if it had been there to begin with. From the user's point of view, the function call works as expected, using the function definition in the loaded file.

An autoload object is usually created with the function `autoload`, which stores the object in the function cell of a symbol. See Section 14.2 [Autoload], page 198, for more details.

2.4.16 Char Table Type

(not yet documented)

2.4.17 Hash Table Type

A *hash table* is a table providing an arbitrary mapping from one Lisp object to another, using an internal indexing method called *hashing*. Hash tables are very fast (much more efficient that using an association list, when there are a large number of elements in the table).

Hash tables have no read syntax. They print in hash notation (The "hash" in "hash notation" has nothing to do with the "hash" in "hash table"), giving the number of elements, total space allocated for elements, and a unique number assigned at the time the hash table was created. (Hash tables automatically resize as necessary so there is no danger of running out of space for elements.)

```
(make-hashtable 50)
     ⇒ #<hashtable 0/71 0x313a>
```

See Chapter 46 [Hash Tables], page 665, for information on how to create and work with hash tables.

2.4.18 Range Table Type

A *range table* is a table that maps from ranges of integers to arbitrary Lisp objects. Range tables automatically combine overlapping ranges that map to the same Lisp object, and operations are provided for mapping over all of the ranges in a range table.

Range tables have a special read syntax beginning with '`#s(range-table`' (this is an example of *structure* read syntax, which is also used for char tables and faces).

```
(setq x (make-range-table))
(put-range-table 20 50 'foo x)
(put-range-table 100 200 "bar" x)
x
     ⇒ #s(range-table data ((20 50) foo (100 200) "bar"))
```

See Chapter 47 [Range Tables], page 669, for information on how to create and work with range tables.

2.4.19 Weak List Type

(not yet documented)

2.5 Editing Types

The types in the previous section are common to many Lisp dialects. XEmacs Lisp provides several additional data types for purposes connected with editing.

2.5.1 Buffer Type

A *buffer* is an object that holds text that can be edited (see Chapter 30 [Buffers], page 431). Most buffers hold the contents of a disk file (see Chapter 28 [Files], page 391) 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 31 [Windows], page 445). But a buffer need not be displayed in any window.

The contents of a buffer are much like a string, but buffers are not used like strings in XEmacs Lisp, and the available operations are different. For example, insertion of text into a buffer is very efficient, whereas "inserting" text into a string requires concatenating substrings, and the result is an entirely new string object.

Each buffer has a designated position called *point* (see Chapter 34 [Positions], page 487). At any time, one buffer is the *current buffer*. Most editing commands act on the contents of the current buffer in the neighborhood of point. Many of the standard Emacs functions manipulate or test the characters in the current buffer; a whole chapter in this manual is devoted to describing these functions (see Chapter 36 [Text], page 511).

Several other data structures are associated with each buffer:

- a local syntax table (see Chapter 38 [Syntax Tables], page 569);
- a local keymap (see Chapter 20 [Keymaps], page 315);
- a local variable binding list (see Section 10.9 [Buffer-Local Variables], page 157);
- a list of extents (see Chapter 40 [Extents], page 585);
- and various other related properties.

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

A buffer may be *indirect*, which means it shares the text of another buffer. See Section 30.11 [Indirect Buffers], page 442.

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

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

2.5.2 Marker Type

A *marker* denotes a position in a specific buffer. Markers therefore have two components: one for the buffer, and one for the position. Changes in the buffer's text automatically relocate the position value as necessary to ensure that the marker always points between the same two characters in the buffer.

Markers have no read syntax. They print in hash notation, giving the current character position and the name of the buffer.

```
(point-marker)
     ⇒ #<marker at 50661 in objects.texi>
```

See Chapter 35 [Markers], page 499, for information on how to test, create, copy, and move markers.

2.5.3 Extent Type

An *extent* specifies temporary alteration of the display appearance of a part of a buffer (or string). It contains markers delimiting a range of the buffer, plus a property list (a list whose elements are alternating property names and values). Extents are used to present parts of the buffer temporarily in a different display style. They have no read syntax, and print in hash notation, giving the buffer name and range of positions.

Extents can exist over strings as well as buffers; the primary use of this is to preserve extent and text property information as text is copied from one buffer to another or between different parts of a buffer.

Extents have no read syntax. They print in hash notation, giving the range of text they cover, the name of the buffer or string they are in, the address in core, and a summary of some of the properties attached to the extent.

```
(extent-at (point))
     ⇒ #<extent [51742, 51748) font-lock text-prop 0x90121e0 in buffer objects.texi>
```

See Chapter 40 [Extents], page 585, for how to create and use extents.

Extents are used to implement text properties. See Section 36.18 [Text Properties], page 539.

2.5.4 Window Type

A *window* describes the portion of the frame that XEmacs uses to display a buffer. (In standard window-system usage, a *window* is what XEmacs calls a *frame*; XEmacs confusingly uses the term "window" to refer to what is called a *pane* in standard window-system usage.) Every window has one associated buffer, whose contents appear in the window. By contrast, a given buffer may appear in one window, no window, or several windows.

Though many windows may exist simultaneously, at any time one window is designated the *selected window*. This is the window where the cursor is (usually) displayed when XEmacs 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 2.5.5 [Frame Type], page 31.

Windows have no read syntax. They print in hash notation, giving the name of the buffer being displayed and a unique number assigned at the time the window was created. (This number can be useful because the buffer displayed in any given window can change frequently.)

```
(selected-window)
     ⇒ #<window on "objects.texi" 0x266c>
```

See Chapter 31 [Windows], page 445, for a description of the functions that work on windows.

2.5.5 Frame Type

A *frame* is a rectangle on the screen (a *window* in standard window-system terminology) that contains one or more non-overlapping Emacs windows (*panes* in standard window-system terminology). 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 type, name as used for resourcing, and a unique number assigned at the time the frame was created.

```
(selected-frame)
     ⇒ #<x-frame "emacs" 0x9db>
```

See Chapter 32 [Frames], page 471, for a description of the functions that work on frames.

2.5.6 Device Type

A *device* represents a single display on which frames exist. Normally, there is only one device object, but there may be more than one if XEmacs is being run on a multi-headed display (e.g. an X server with attached color and mono screens) or if XEmacs is simultaneously driving frames attached to different consoles, e.g. an X display and a TTY connection.

Devices do not have a read syntax. They print in hash notation, giving the device's type, connection name, and a unique number assigned at the time the device was created.

```
(selected-device)
     ⇒ #<x-device on ":0.0" 0x5b9>
```

See Chapter 33 [Consoles and Devices], page 483, for a description of several functions related to devices.

2.5.7 Console Type

A *console* represents a single keyboard to which devices (i.e. displays on which frames exist) are connected. Normally, there is only one console object, but there may be more than one if XEmacs is simultaneously driving frames attached to different X servers and/or TTY connections. (XEmacs is capable of driving multiple X and TTY connections at the same time, and provides a robust mechanism for handling the differing display capabilities of such heterogeneous environments. A buffer with embedded glyphs and multiple fonts and colors, for example, will display reasonably if it simultaneously appears on a frame on a color X display, a frame on a mono X display, and a frame on a TTY connection.)

Consoles do not have a read syntax. They print in hash notation, giving the console's type, connection name, and a unique number assigned at the time the console was created.

```
(selected-console)
     ⇒ #<x-console on "localhost:0" 0x5b7>
```

See Chapter 33 [Consoles and Devices], page 483, for a description of several functions related to consoles.

2.5.8 Window Configuration Type

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

Window configurations do not have a read syntax. They print in hash notation, giving a unique number assigned at the time the window configuration was created.

```
(current-window-configuration)
     ⇒ #<window-configuration 0x2db4>
```

See Section 31.16 [Window Configurations], page 468, for a description of several functions related to window configurations.

2.5.9 Event Type

(not yet documented)

2.5.10 Process Type

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

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

Process objects have no read syntax. They print in hash notation, giving the name of the process, its associated process ID, and the current state of the process:

```
(process-list)
     ⇒ (#<process "shell" pid 2909 state:run>)
```

See Chapter 49 [Processes], page 673, for information about functions that create, delete, return information about, send input or signals to, and receive output from processes.

2.5.11 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 18 [Minibuffers], page 261) or output in the echo area (see Section 45.3 [The Echo Area], page 648).

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

See Chapter 17 [Read and Print], page 251, for a description of functions related to streams, including parsing and printing functions.

2.5.12 Keymap Type

A *keymap* maps keys typed by the user to commands. This mapping controls how the user's command input is executed.

NOTE: In XEmacs, a keymap is a separate primitive type. In FSF GNU Emacs, a keymap is actually a list whose CAR is the symbol `keymap`.

See Chapter 20 [Keymaps], page 315, for information about creating keymaps, handling prefix keys, local as well as global keymaps, and changing key bindings.

2.5.13 Syntax Table Type

Under XEmacs 20, a *syntax table* is a particular type of char table. Under XEmacs 19, a syntax table a vector of 256 integers. In both cases, each element defines how one character is interpreted when it appears in a buffer. For example, in C mode (see Section 26.1 [Major Modes], page 361), the '`+`' character is punctuation, but in Lisp mode it is a valid character in a symbol. These modes specify different interpretations by changing the syntax table entry for '`+`'.

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

See Chapter 38 [Syntax Tables], page 569, for details about syntax classes and how to make and modify syntax tables.

2.5.14 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 256, although in XEmacs 20 this may change to be a particular type of char table. See Section 45.11 [Display Tables], page 659.

2.5.15 Database Type

(not yet documented)

2.5.16 Charset Type

(not yet documented)

2.5.17 Coding System Type

(not yet documented)

2.5.18 ToolTalk Message Type

(not yet documented)

2.5.19 ToolTalk Pattern Type

(not yet documented)

2.6 Window-System Types

XEmacs also has some types that represent objects such as faces (collections of display characters), fonts, and pixmaps that are commonly found in windowing systems.

2.6.1 Face Type

(not yet documented)

2.6.2 Glyph Type

(not yet documented)

2.6.3 Specifier Type

(not yet documented)

2.6.4 Font Instance Type

(not yet documented)

2.6.5 Color Instance Type

(not yet documented)

2.6.6 Image Instance Type

(not yet documented)

2.6.7 Toolbar Button Type

(not yet documented)

2.6.8 Subwindow Type

(not yet documented)

2.6.9 X Resource Type

(not yet documented)

2.7 Type Predicates

The XEmacs Lisp interpreter itself does not perform type checking on the actual arguments passed to functions when they are called. It could not do so, since function arguments in Lisp do not have declared data types, as they do in other programming languages. It is therefore up to the individual function to test whether each actual argument belongs to a type that the function can use.

All built-in functions do check the types of their actual arguments when appropriate, and signal a `wrong-type-argument` error if an argument is of the wrong type. For example, here is what happens if you pass an argument to `+` that it cannot handle:

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

If you want your program to handle different types differently, you must do explicit type checking. The most common way to check the type of an object is to call a *type predicate* function. Emacs has a type predicate for each type, as well as some predicates for combinations of types.

A type predicate function takes one argument; it returns `t` if the argument belongs to the appropriate type, and `nil` otherwise. Following a general Lisp convention for predicate functions, most type predicates' names end with '`p`'.

Here is an example which uses the predicates `listp` to check for a list and `symbolp` to check for a symbol.

```
(defun add-on (x)
  (cond ((symbolp x)
         ;; If X is a symbol, put it on LIST.
         (setq list (cons x list)))
        ((listp x)
```

```
           ;; If X is a list, add its elements to LIST.
           (setq list (append x list)))

          (t
           ;; We only handle symbols and lists.
           (error "Invalid argument %s in add-on" x))))
```

Here is a table of predefined type predicates, in alphabetical order, with references to further information.

`annotationp`
: See Section 44.2 [Annotation Primitives], page 642.

`arrayp`
: See Section 6.3 [Array Functions], page 104.

`atom`
: See Section 5.3 [List-related Predicates], page 78.

`bit-vector-p`
: See Section 6.7 [Bit Vector Functions], page 108.

`bitp`
: See Section 6.7 [Bit Vector Functions], page 108.

`boolean-specifier-p`
: See Section 41.4 [Specifier Types], page 602.

`buffer-glyph-p`
: See Section 43.3 [Glyph Types], page 638.

`buffer-live-p`
: See Section 30.10 [Killing Buffers], page 441.

`bufferp`
: See Section 30.1 [Buffer Basics], page 431.

`button-event-p`
: See Section 19.5.3 [Event Predicates], page 294.

`button-press-event-p`
: See Section 19.5.3 [Event Predicates], page 294.

`button-release-event-p`
: See Section 19.5.3 [Event Predicates], page 294.

`case-table-p`
: See Section 4.12 [Case Tables], page 71.

`char-int-p`
: See Section 4.5 [Character Codes], page 62.

`char-or-char-int-p`
: See Section 4.5 [Character Codes], page 62.

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

`char-table-p`
: See Section 4.13 [Char Tables], page 73.

`characterp`
: See Section 4.4 [Predicates for Characters], page 62.

`color-instance-p`
: See Section 42.3 [Colors], page 623.

`color-pixmap-image-instance-p`
: See Section 43.2.3.1 [Image Instance Types], page 635.

`color-specifier-p`
: See Section 41.4 [Specifier Types], page 602.

`commandp` See Section 19.3 [Interactive Call], page 286.

`compiled-function-p`
: See Section 2.4.14 [Compiled-Function Type], page 28.

`console-live-p`
: See Section 33.4 [Connecting to a Console or Device], page 485.

`consolep` See Chapter 33 [Consoles and Devices], page 483.

`consp` See Section 5.3 [List-related Predicates], page 78.

`database-live-p`
: See Section 48.1 [Connecting to a Database], page 671.

`databasep`
: See Chapter 48 [Databases], page 671.

`device-live-p`
: See Section 33.4 [Connecting to a Console or Device], page 485.

`device-or-frame-p`
: See Section 33.2 [Basic Device Functions], page 484.

`devicep` See Chapter 33 [Consoles and Devices], page 483.

`eval-event-p`
: See Section 19.5.3 [Event Predicates], page 294.

`event-live-p`
: See Section 19.5.3 [Event Predicates], page 294.

`eventp` See Section 19.5 [Events], page 290.

`extent-live-p`
: See Section 40.2 [Creating and Modifying Extents], page 586.

`extentp` See Chapter 40 [Extents], page 585.

`face-boolean-specifier-p`
: See Section 41.4 [Specifier Types], page 602.

`facep` See Section 42.1.2 [Basic Face Functions], page 616.

`floatp` See Section 3.3 [Predicates on Numbers], page 46.

`font-instance-p`
See Section 42.2 [Fonts], page 621.

`font-specifier-p`
See Section 41.4 [Specifier Types], page 602.

`frame-live-p`
See Section 32.4 [Deleting Frames], page 476.

`framep` See Chapter 32 [Frames], page 471.

`functionp`
(not yet documented)

`generic-specifier-p`
See Section 41.4 [Specifier Types], page 602.

`glyphp` See Chapter 43 [Glyphs], page 625.

`hashtablep`
See Chapter 46 [Hash Tables], page 665.

`icon-glyph-p`
See Section 43.3 [Glyph Types], page 638.

`image-instance-p`
See Section 43.2 [Images], page 630.

`image-specifier-p`
See Section 41.4 [Specifier Types], page 602.

`integer-char-or-marker-p`
See Section 35.2 [Predicates on Markers], page 500.

`integer-or-char-p`
See Section 4.4 [Predicates for Characters], page 62.

`integer-or-marker-p`
See Section 35.2 [Predicates on Markers], page 500.

`integer-specifier-p`
See Section 41.4 [Specifier Types], page 602.

`integerp` See Section 3.3 [Predicates on Numbers], page 46.

`itimerp` (not yet documented)

`key-press-event-p`
See Section 19.5.3 [Event Predicates], page 294.

`keymapp` See Section 20.3 [Creating Keymaps], page 316.

`keywordp` (not yet documented)

`listp` See Section 5.3 [List-related Predicates], page 78.

`markerp` See Section 35.2 [Predicates on Markers], page 500.

`misc-user-event-p`
: See Section 19.5.3 [Event Predicates], page 294.

`mono-pixmap-image-instance-p`
: See Section 43.2.3.1 [Image Instance Types], page 635.

`motion-event-p`
: See Section 19.5.3 [Event Predicates], page 294.

`mouse-event-p`
: See Section 19.5.3 [Event Predicates], page 294.

`natnum-specifier-p`
: See Section 41.4 [Specifier Types], page 602.

`natnump`
: See Section 3.3 [Predicates on Numbers], page 46.

`nlistp`
: See Section 5.3 [List-related Predicates], page 78.

`nothing-image-instance-p`
: See Section 43.2.3.1 [Image Instance Types], page 635.

`number-char-or-marker-p`
: See Section 35.2 [Predicates on Markers], page 500.

`number-or-marker-p`
: See Section 35.2 [Predicates on Markers], page 500.

`numberp`
: See Section 3.3 [Predicates on Numbers], page 46.

`pointer-glyph-p`
: See Section 43.3 [Glyph Types], page 638.

`pointer-image-instance-p`
: See Section 43.2.3.1 [Image Instance Types], page 635.

`process-event-p`
: See Section 19.5.3 [Event Predicates], page 294.

`processp`
: See Chapter 49 [Processes], page 673.

`range-table-p`
: See Chapter 47 [Range Tables], page 669.

`ringp`
: (not yet documented)

`sequencep`
: See Section 6.1 [Sequence Functions], page 101.

`specifierp`
: See Chapter 41 [Specifiers], page 599.

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

`subrp`
: See Section 11.8 [Function Cells], page 173.

`subwindow-image-instance-p`
: See Section 43.2.3.1 [Image Instance Types], page 635.

`subwindowp`
: See Section 43.6 [Subwindows], page 640.

`symbolp`
: See Chapter 7 [Symbols], page 111.

`syntax-table-p`
: See Chapter 38 [Syntax Tables], page 569.

`text-image-instance-p`
: See Section 43.2.3.1 [Image Instance Types], page 635.

`timeout-event-p`
: See Section 19.5.3 [Event Predicates], page 294.

`toolbar-button-p`
: See Chapter 23 [Toolbar], page 351.

`toolbar-specifier-p`
: See Chapter 23 [Toolbar], page 351.

`user-variable-p`
: See Section 10.5 [Defining Variables], page 149.

`vectorp`
: See Section 6.4 [Vectors], page 106.

`weak-list-p`
: See Section 5.10 [Weak Lists], page 98.

`window-configuration-p`
: See Section 31.16 [Window Configurations], page 468.

`window-live-p`
: See Section 31.3 [Deleting Windows], page 449.

`windowp`
: See Section 31.1 [Basic Windows], page 445.

The most general way to check the type of an object is to call the function `type-of`. Recall that each object belongs to one and only one primitive type; `type-of` tells you which one (see Chapter 2 [Lisp Data Types], page 15). But `type-of` knows nothing about non-primitive types. In most cases, it is more convenient to use type predicates than `type-of`.

type-of *object* — Function

This function returns a symbol naming the primitive type of *object*. The value is one of `bit-vector`, `buffer`, `char-table`, `character`, `charset`, `coding-system`, `cons`, `color-instance`, `compiled-function`, `console`, `database`, `device`, `event`, `extent`, `face`, `float`, `font-instance`, `frame`, `glyph`, `hashtable`, `image-instance`, `integer`, `keymap`, `marker`, `process`, `range-table`, `specifier`, `string`, `subr`, `subwindow`, `symbol`, `toolbar-button`, `tooltalk-message`, `tooltalk-pattern`, `vector`, `weak-list`, `window`, `window-configuration`, or `x-resource`.

```
(type-of 1)
     ⇒ integer
(type-of 'nil)
     ⇒ symbol
```

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

2.8 Equality Predicates

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

eq *object1 object2* Function

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

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

(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 113.)

NOTE: Under XEmacs 19, characters are really just integers, and thus characters and integers are `eq`. Under XEmacs 20, it was necessary to preserve remnants of this in function such as `old-eq` in order to maintain byte-code compatibility. Byte code compiled under any Emacs 19 will automatically have calls to `eq` mapped to `old-eq` when executed under XEmacs 20.

```
(eq 'foo 'foo)
     ⇒ t

(eq 456 456)
     ⇒ t

(eq "asdf" "asdf")
     ⇒ nil

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

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

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

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

old-eq *obj1 obj2* Function

This function exists under XEmacs 20 and is exactly like `eq` except that it suffers from the char-int confoundance disease. In other words, it returns `t` if given a character and the equivalent integer, even though the objects are of different types! You should *not* ever call this function explicitly in your code. However, be aware that all calls to `eq` in byte code compiled under version 19 map to `old-eq` in XEmacs 20. (Likewise for `old-equal`, `old-memq`, `old-member`, `old-assq` and `old-assoc`.)

```
;; Remember, this does not apply under XEmacs 19.
?A
     ⇒ ?A
(char-int ?A)
     ⇒ 65
(old-eq ?A 65)
     ⇒ t               ; Eek, we've been infected.
(eq ?A 65)
     ⇒ nil             ; We are still healthy.
```

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.

Note that in FSF GNU Emacs, comparison of strings takes into account their text properties, and you have to use `string-equal` if you want only the strings themselves compared. This difference does not exist in XEmacs; `equal` and `string-equal` always return the same value on the same strings.

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

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

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

3 Numbers

XEmacs supports two numeric data types: *integers* and *floating point numbers*. Integers are whole numbers such as −3, 0, #b0111, #xFEED, #o744. Their values are exact. The number prefixes '#b', '#o', and '#x' are supported to represent numbers in binary, octal, and hexadecimal notation (or radix). Floating point numbers are numbers with fractional parts, such as −4.5, 0.0, or 2.71828. They can also be expressed in exponential notation: 1.5e2 equals 150; in this example, '`e2`' stands for ten to the second power, and is multiplied by 1.5. Floating point values are not exact; they have a fixed, limited amount of precision.

3.1 Integer Basics

The range of values for an integer depends on the machine. The minimum range is −134217728 to 134217727 (28 bits; i.e., -2^{27} to $2^{27}-1$), but some machines may provide a wider range. Many examples in this chapter assume an integer has 28 bits.

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

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

To understand how various functions work on integers, especially the bitwise operators (see Section 3.8 [Bitwise Operations], page 53), it is often helpful to view the numbers in their binary form.

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

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

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

The integer −1 looks like this:

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

−1 is represented as 28 ones. (This is called *two's complement* notation.)

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

```
1111  1111 1111  1111 1111  1111 1011
```

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

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

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

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

Many of the following functions accept markers for arguments as well as integers. (See Chapter 35 [Markers], page 499.) More precisely, the actual arguments to such functions may be either integers or markers, which is why we often give these arguments the name *int-or-marker*. When the argument value is a marker, its position value is used and its buffer is ignored.

3.2 Floating Point Basics

XEmacs supports floating point numbers. 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`'.

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

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`, `integer-char-or-marker-p`, `number-or-marker-p` and `number-char-or-marker-p`, in Section 35.2 [Predicates on Markers], page 500.

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 nonnegative integer, and returns `t` if so, `nil` otherwise. 0 is considered non-negative.

zerop *number* Function

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

These two forms are equivalent: `(zerop x)` ≡ `(= x 0)`.

3.4 Comparison of Numbers

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

At present, each integer value has a unique Lisp object in XEmacs Lisp. Therefore, `eq` is equivalent to `=` where integers are concerned. It is sometimes convenient to use `eq` for comparing an unknown value with an integer, because `eq` does not report an error if the unknown value is not a number—it accepts arguments of any type. By contrast, `=` signals an error if the arguments are not numbers or markers. However, it is a good idea to use `=` if you can, even for comparing integers, just in case we change the representation of integers in a future XEmacs 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:

```
(defconst fuzz-factor 1.0e-6)
(defun approx-equal (x y)
  (or (and (= x 0) (= y 0))
      (< (/ (abs (- x y))
            (max (abs x) (abs y)))
         fuzz-factor)))
```

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

In addition to numbers, all of the following functions also accept characters and markers as arguments, and treat them as their number equivalents.

`=` *number* &rest *more-numbers* — Function

This function returns `t` if all of its arguments are numerically equal, `nil` otherwise.

```
(= 5)
     ⇒ t
(= 5 6)
     ⇒ nil
(= 5 5.0)
     ⇒ t
(= 5 5 6)
     ⇒ nil
```

`/=` *number* &rest *more-numbers* — Function

This function returns `t` if no two arguments are numerically equal, `nil` otherwise.

```
(/= 5 6)
     ⇒ t
(/= 5 5 6)
     ⇒ nil
(/= 5 6 1)
     ⇒ t
```

`<` *number* &rest *more-numbers* — Function

This function returns `t` if the sequence of its arguments is monotonically increasing, `nil` otherwise.

```
(< 5 6)
     ⇒ t
(< 5 6 6)
     ⇒ nil
(< 5 6 7)
     ⇒ t
```

`<=` *number* &rest *more-numbers* — Function

This function returns `t` if the sequence of its arguments is monotonically nondecreasing, `nil` otherwise.

```
(<= 5 6)
     ⇒ t
(<= 5 6 6)
     ⇒ t
(<= 5 6 5)
     ⇒ nil
```

`>` *number* &rest *more-numbers* — Function

This function returns `t` if the sequence of its arguments is monotonically decreasing, `nil` otherwise.

`>=` *number* &rest *more-numbers* — Function

This function returns `t` if the sequence of its arguments is monotonically nonincreasing, `nil` otherwise.

max *number* &rest *more-numbers* Function

This function returns the largest of its arguments.

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

min *number* &rest *more-numbers* Function

This function returns the smallest of its arguments.

```
(min -4 1)
     ⇒ -4
```

3.5 Numeric Conversions

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

float *number* Function

This returns *number* converted to floating point. If *number* is already a floating point number, `float` returns it unchanged.

There are four functions to convert floating point numbers to integers; they differ in how they round. These functions accept integer arguments also, and return such arguments unchanged.

truncate *number* Function

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

floor *number* &optional *divisor* Function

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

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

ceiling *number* Function

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

round *number* Function

This returns *number*, converted to an integer by rounding towards the nearest integer. Rounding a value equidistant between two integers may choose the integer closer to zero, or it may prefer an even integer, depending on your machine.

3.6 Arithmetic Operations

XEmacs Lisp provides the traditional four arithmetic operations: addition, subtraction, multiplication, and division. Remainder and modulus functions supplement the division functions. The functions to add or subtract 1 are provided because they are traditional in Lisp and commonly used.

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

It is important to note that in XEmacs Lisp, arithmetic functions do not check for overflow. Thus `(1+ 134217727)` may evaluate to -134217728, depending on your hardware.

1+ *number-or-marker* — Function

This function returns *number-or-marker* plus 1. For example,

```
(setq foo 4)
     ⇒ 4
(1+ foo)
     ⇒ 5
```

This function is not analogous to the C operator `++`—it does not increment a variable. It just computes a sum. Thus, if we continue,

```
foo
     ⇒ 4
```

If you want to increment the variable, you must use `setq`, like this:

```
(setq foo (1+ foo))
     ⇒ 5
```

Now that the `cl` package is always available from lisp code, a more convenient and natural way to increment a variable is `(incf foo)`.

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.

```
(+)
     ⇒ 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, `-` subtracts each of the *other-numbers-or-markers* from *number-or-marker*, cumulatively. If there are no arguments, the result is 0.

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

***** &rest *numbers-or-markers* — Function

This function multiplies its arguments together, and returns the product. When given no arguments, `*` returns 1.

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

/ *dividend divisor* &rest *divisors* — Function

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

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

If you divide by 0, an `arith-error` error is signaled. (See Section 9.5.3 [Errors], page 136.)

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

The result of `(/ -17 6)` could in principle be -3 on some machines.

% *dividend divisor* — Function

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

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

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

```
(% 9 4)
     ⇒ 1
```

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

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

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

always equals *dividend*.

mod *dividend divisor* Function

This function returns the value of *dividend* modulo *divisor*; in other words, the remainder after division of *dividend* by *divisor*, but with the same sign as *divisor*. The arguments must be numbers or markers.

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

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

```
(mod 9 4)
     ⇒ 1
(mod -9 4)
     ⇒ 3
(mod 9 -4)
     ⇒ -3
(mod -9 -4)
     ⇒ -1
(mod 5.5 2.5)
     ⇒ .5
```

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

```
(+ (mod dividend divisor)
   (* (floor dividend divisor) divisor))
```

always equals *dividend*, subject to rounding error if either argument is floating point. For `floor`, see Section 3.5 [Numeric Conversions], page 49.

3.7 Rounding Operations

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

ffloor *float* Function

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

fceiling *float* Function

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

ftruncate *float* Function

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

fround *float* Function

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

3.8 Bitwise Operations on Integers

In a computer, an integer is represented as a binary number, a sequence of *bits* (digits which are either zero or one). A bitwise operation acts on the individual bits of such a sequence. For example, *shifting* moves the whole sequence left or right one or more places, reproducing the same pattern "moved over".

The bitwise operations in XEmacs Lisp apply only to integers.

lsh *integer1 count* Function

`lsh`, which is an abbreviation for *logical shift*, shifts the bits in *integer1* to the left *count* places, or to the right if *count* is negative, bringing zeros into the vacated bits. If *count* is negative, `lsh` shifts zeros into the leftmost (most-significant) bit, producing a positive result even if *integer1* is negative. Contrast this with `ash`, below.

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

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

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

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

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

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

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

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

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

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

The function `lsh`, like all XEmacs Lisp arithmetic functions, does not check for overflow, so shifting left can discard significant bits and change the sign of the number. For example, left shifting 134,217,727 produces −2 on a 28-bit machine:

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

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

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

which becomes the following when left shifted:

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

ash *integer1 count* Function

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

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

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

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

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

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

Here are other examples:

```
                    ;              28-bit binary values

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

logand &rest *ints-or-markers* Function

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

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

Therefore,

```
(logand 13 12)
     ⇒ 12
```

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

```
                    ;              28-bit binary values

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

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

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

logior &rest *ints-or-markers* Function

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

```
                       ;        28-bit binary values

(logior 12 5)          ; 12  =  0000  0000 0000  0000 0000  0000 1100
                       ;  5  =  0000  0000 0000  0000 0000  0000 0101
     ⇒ 13              ; 13  =  0000  0000 0000  0000 0000  0000 1101
(logior 12 5 7)        ; 12  =  0000  0000 0000  0000 0000  0000 1100
                       ;  5  =  0000  0000 0000  0000 0000  0000 0101
                       ;  7  =  0000  0000 0000  0000 0000  0000 0111
     ⇒ 15              ; 15  =  0000  0000 0000  0000 0000  0000 1111
```

logxor &rest *ints-or-markers* Function

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

```
                       ;        28-bit binary values

(logxor 12 5)          ; 12  =  0000  0000 0000  0000 0000  0000 1100
                       ;  5  =  0000  0000 0000  0000 0000  0000 0101
     ⇒ 9               ;  9  =  0000  0000 0000  0000 0000  0000 1001
(logxor 12 5 7)        ; 12  =  0000  0000 0000  0000 0000  0000 1100
                       ;  5  =  0000  0000 0000  0000 0000  0000 0101
                       ;  7  =  0000  0000 0000  0000 0000  0000 0111
     ⇒ 14              ; 14  =  0000  0000 0000  0000 0000  0000 1110
```

lognot *integer* Function

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

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

3.9 Standard Mathematical Functions

These mathematical functions are available if floating point is supported (which is the normal state of affairs). 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*.

sinh *arg* Function
cosh *arg* Function
tanh *arg* Function

These are the ordinary hyperbolic trigonometric functions.

asinh *arg* Function
acosh *arg* Function
atanh *arg* Function

These are the inverse hyperbolic trigonometric functions.

exp *arg* Function

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

log *arg* &optional *base* Function

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

log10 *arg* Function

This function returns the logarithm of *arg*, with base 10. If *arg* is negative, the result is a NaN. `(log10` *x*`)` $\equiv$ `(log` *x* `10)`, at least approximately.

expt *x y* Function

This function returns *x* raised to power *y*. If both arguments are integers and *y* is positive, the result is an integer; in this case, it is truncated to fit the range of possible integer values.

sqrt *arg* Function

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

cube-root *arg* Function

This returns the cube root of *arg*.

3.10 Random Numbers

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

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

If you want truly unpredictable random numbers, execute `(random t)`. This chooses a new seed based on the current time of day and on XEmacs's process ID number.

random &optional *limit* Function

This function returns a pseudo-random integer. Repeated calls return a series of pseudo-random integers.

If *limit* is a positive integer, the value is chosen to be nonnegative and less than *limit*.

If *limit* is `t`, it means to choose a new seed based on the current time of day and on XEmacs'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 XEmacs 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, XEmacs Lisp has many functions expressly for manipulating them. XEmacs Lisp programs use strings more often than individual characters.

4.1 String and Character Basics

Strings in XEmacs Lisp are arrays that contain an ordered sequence of characters. Characters are their own primitive object type in XEmacs 20. However, in XEmacs 19, characters are represented in XEmacs Lisp as integers; whether an integer was intended as a character or not is determined only by how it is used. See Section 2.4.3 [Character Type], page 18.

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 101.) 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 104).

Strings use an efficient representation for storing the characters in them, and thus take up much less memory than a vector of the same length.

Sometimes you will see strings used to hold key sequences. This exists for backward compatibility with Emacs 18, but should *not* be used in new code, since many key chords can't be represented at all and others (in particular meta key chords) are confused with accented characters.

Strings are useful for holding regular expressions. You can also match regular expressions against strings (see Section 37.3 [Regexp Search], page 557). The functions `match-string` (see Section 37.6.1 [Simple Match Data], page 562) and `replace-match` (see Section 37.6.2 [Replacing Match], page 563) are useful for decomposing and modifying strings based on regular expression matching.

Like a buffer, a string can contain extents in it. These extents are created when a function such as `buffer-substring` is called on a region with duplicable extents in it. When the string is inserted into a buffer, the extents are inserted along with it. See Section 40.9 [Duplicable Extents], page 596.

See Chapter 36 [Text], page 511, for information about functions that display strings or copy them into buffers. See Section 2.4.3 [Character Type], page 18, and Section 2.4.8 [String Type], page 25, 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 101, and Section 6.2 [Arrays], page 103.

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, `nil` otherwise.

In XEmacs addition, this function also returns `t` if *object* is an integer that can be represented as a character. This is because of compatibility with previous XEmacs and should not be depended on.

4.3 Creating Strings

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

string &rest *characters* Function

This function returns a new string made up of *characters*.

```
(string ?X ?E ?m ?a ?c ?s)
     ⇒ "XEmacs"
(string)
     ⇒ ""
```

Analogous functions operating on other data types include `list`, `cons` (see Section 5.5 [Building Lists], page 82), `vector` (see Section 6.4 [Vectors], page 106) and `bit-vector` (see Section 6.6 [Bit Vectors], page 107). This function has not been available in XEmacs prior to 21.0 and FSF Emacs prior to 20.3.

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 4.7 [String Conversion], page 65), `make-vector` (see Section 6.4 [Vectors], page 106), and `make-list` (see Section 5.5 [Building Lists], page 82).

substring *string start* &optional *end* Function

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

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

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

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

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

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

When `nil` is used as an index, it stands for the length of the string. Thus,

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

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

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

But we recommend `copy-sequence` for this purpose (see Section 6.1 [Sequence Functions], page 101).

If the characters copied from *string* have duplicable extents or text properties, those are copied into the new string also. See Section 40.9 [Duplicable Extents], page 596.

A `wrong-type-argument` error is signaled if either *start* or *end* is not an integer or `nil`. An `args-out-of-range` error is signaled if *start* indicates a character following *end*, or if either integer is out of range for *string*.

Contrast this function with `buffer-substring` (see Section 36.2 [Buffer Contents], page 512), which returns a string containing a portion of the text in the current buffer. The beginning of a string is at index 0, but the beginning of a buffer is at index 1.

concat &rest *sequences* Function

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

```
(concat "abc" "-def")
     ⇒ "abc-def"
(concat "abc" (list 120 (+ 256 121)) [122])
     ⇒ "abcxyz"
;; nil is an empty sequence.
(concat "abc" nil "-def")
     ⇒ "abc-def"
(concat "The " "quick brown " "fox.")
     ⇒ "The quick brown fox."
(concat)
```

```
    ⇒ ""
```

The 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. **Don't use this feature; we plan to eliminate it. If you already use this feature, change your programs now!** The proper way to convert an integer to a decimal number in this way is with `format` (see Section 4.10 [Formatting Strings], page 67) or `number-to-string` (see Section 4.7 [String Conversion], page 65).

```
(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 171, `vconcat` in Section 6.4 [Vectors], page 106, `bvconcat` in Section 6.6 [Bit Vectors], page 107, and `append` in Section 5.5 [Building Lists], page 82.

4.4 The Predicates for Characters

characterp *object* Function

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

Some functions that work on integers (e.g. the comparison functions `<`, `<=`, `=`, `/=`, etc. and the arithmetic functions `+`, `-`, `*`, etc.) accept characters and implicitly convert them into integers. In general, functions that work on characters also accept char-ints and implicitly convert them into characters. WARNING: Neither of these behaviors is very desirable, and they are maintained for backward compatibility with old E-Lisp programs that confounded characters and integers willy-nilly. These behaviors may change in the future; therefore, do not rely on them. Instead, convert the characters explicitly using `char-int`.

integer-or-char-p *object* Function

This function returns `t` if *object* is an integer or character.

4.5 Character Codes

char-int *ch* Function

This function converts a character into an equivalent integer. The resulting integer will always be non-negative. The integers in the range 0 - 255 map to characters as follows:

0 - 31 Control set 0

32 - 127 ASCII

128 - 159	Control set 1
160 - 255	Right half of ISO-8859-1

If support for MULE does not exist, these are the only valid character values. When MULE support exists, the values assigned to other characters may vary depending on the particular version of XEmacs, the order in which character sets were loaded, etc., and you should not depend on them.

int-char *integer* Function

This function converts an integer into the equivalent character. Not all integers correspond to valid characters; use `char-int-p` to determine whether this is the case. If the integer cannot be converted, `nil` is returned.

char-int-p *object* Function

This function returns `t` if *object* is an integer that can be converted into a character.

char-or-char-int-p *object* Function

This function returns `t` if *object* is a character or an integer that can be converted into one.

4.6 Comparison of Characters and Strings

char-equal *character1 character2* Function

This function returns `t` if the arguments represent the same character, `nil` otherwise. This function ignores differences in case if `case-fold-search` is non-`nil`.

```
(char-equal ?x ?x)
     ⇒ t
(let ((case-fold-search t))
  (char-equal ?x ?X))
     ⇒ t
(let ((case-fold-search nil))
  (char-equal ?x ?X))
     ⇒ nil
```

char= *character1 character2* Function

This function returns `t` if the arguments represent the same character, `nil` otherwise. Case is significant.

```
(char= ?x ?x)
     ⇒ t
(char= ?x ?X)
     ⇒ nil
(let ((case-fold-search t))
  (char-equal ?x ?X))
     ⇒ nil
(let ((case-fold-search nil))
  (char-equal ?x ?X))
     ⇒ nil
```

string= *string1 string2* Function

This function returns `t` if the characters of the two strings match exactly; case is significant.

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

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 of no characters is less than any other string.

```
(string< "" "abc")
     ⇒ t
(string< "ab" "abc")
     ⇒ t
(string< "abc" "")
     ⇒ nil
(string< "abc" "ab")
     ⇒ nil
(string< "" "")
     ⇒ nil
```

string-lessp *string1 string2* Function

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

See also `compare-buffer-substrings` in Section 36.3 [Comparing Text], page 513, for a way to compare text in buffers. The function `string-match`, which matches a regular expression against a string, can be used for a kind of string comparison; see Section 37.3 [Regexp Search], page 557.

4.7 Conversion of Characters and Strings

This section describes functions for conversions between characters, strings and integers. `format` and `prin1-to-string` (see Section 17.5 [Output Functions], page 256) can also convert Lisp objects into strings. `read-from-string` (see Section 17.3 [Input Functions], page 253) can "convert" a string representation of a Lisp object into an object.

See Chapter 27 [Documentation], page 381, for functions that produce textual descriptions of text characters and general input events (`single-key-description` and `text-char-description`). These functions are used primarily for making help messages.

char-to-string *character* Function

This function returns a new string 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 60.) This conversion can also be done with `format` using the '`%c`' format specification. (See Section 4.10 [Formatting Strings], page 67.)

```
(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. (Under XEmacs 19, the value is also 0 when the first character of *string* is the null character, ASCII code 0.)

```
(string-to-char "ABC")
     ⇒ ?A   ;; Under XEmacs 20.
     ⇒ 65   ;; Under XEmacs 19.
(string-to-char "xyz")
     ⇒ ?x   ;; Under XEmacs 20.
     ⇒ 120  ;; Under XEmacs 19.
(string-to-char "")
     ⇒ 0
(string-to-char "\000")
     ⇒ ?\^ ;; Under XEmacs 20.
     ⇒ 0    ;; Under XEmacs 20.
```

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

number-to-string *number* Function

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

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

`int-to-string` is a semi-obsolete alias for this function.

See also the function `format` in Section 4.10 [Formatting Strings], page 67.

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

This function returns the numeric value of the characters in *string*, read in *base*. It skips spaces and tabs at the beginning of *string*, then reads as much of *string* as it can interpret as a number. (On some systems it ignores other whitespace at the beginning, not just spaces and tabs.) If the first character after the ignored whitespace is not a digit or a minus sign, this function returns 0.

If *base* is not specified, it defaults to ten. With *base* other than ten, only integers can be read.

```
(string-to-number "256")
     ⇒ 256
(string-to-number "25 is a perfect square.")
     ⇒ 25
(string-to-number "X256")
     ⇒ 0
(string-to-number "-4.5")
     ⇒ -4.5
(string-to-number "ffff" 16)
     ⇒ 65535
```

`string-to-int` is an obsolete alias for this function.

4.8 Modifying Strings

You can modify a string using the general array-modifying primitives. See Section 6.2 [Arrays], page 103. The function `aset` modifies a single character; the function `fillarray` sets all characters in the string to a specified character.

Each string has a tick counter that starts out at zero (when the string is created) and is incremented each time a change is made to that string.

string-modified-tick *string* Function

This function returns the tick counter for '`string`'.

4.9 String Properties

Similar to symbols, extents, faces, and glyphs, you can attach additional information to strings in the form of *string properties*. These differ from text properties, which are logically attached to particular characters in the string.

To attach a property to a string, use `put`. To retrieve a property from a string, use `get`. You can also use `remprop` to remove a property from a string and `object-props` to retrieve a list of all the properties in a string.

4.10 Formatting Strings

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

Formatting is often useful for computing messages to be displayed. In fact, the functions `message` and `error` provide the same formatting feature described here; they differ from `format` only in how they use the result of formatting.

format *string* &rest *objects* Function

This function returns a new string that is made by copying *string* and then replacing any format specification in the copy with encodings of the corresponding *objects*. The arguments *objects* are the computed values to be formatted.

A format specification is a sequence of characters beginning with a '`%`'. Thus, if there is a '`%d`' in *string*, the `format` function replaces it with the printed representation of one of the values to be formatted (one of the arguments *objects*). For example:

```
(format "The value of fill-column is %d." fill-column)
     ⇒ "The value of fill-column is 72."
```

If *string* contains more than one format specification, the format specifications correspond with successive values from *objects*. Thus, the first format specification in *string* uses the first such value, the second format specification uses the second such value, and so on. Any extra format specifications (those for which there are no corresponding values) cause unpredictable behavior. Any extra values to be formatted are ignored.

Certain format specifications require values of particular types. 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 valid format specifications:

'`%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. This is equivalent to printing the object with `princ`.

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. This is equivalent to printing the object with `prin1`.

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

'%o' Replace the specification with the base-eight representation of an integer.

'%d'
'%i' Replace the specification with the base-ten representation of an integer.

'%x' Replace the specification with the base-sixteen representation of an integer, using lowercase letters.

'%X' Replace the specification with the base-sixteen representation of an integer, using uppercase letters.

'%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 (e.g. '7.85200e+03').

'%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 a "pretty format". Either exponential notation or decimal-point notation will be used (usually whichever is shorter), and trailing zeroes are removed from the fractional part.

'%%' 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 %d is %o,
         and the hex value is %x." 18 18 18)
     ⇒ "The octal value of 18 is 22,
         and the hex value is 12."
```

There are many additional flags and specifications that can occur between the '%' and the format character, in the following order:

1. An optional repositioning specification, which is a positive integer followed by a '$'.
2. Zero or more of the optional flag characters '-', '+', ' ', '0', and '#'.
3. An asterisk ('*', meaning that the field width is now assumed to have been specified as an argument.
4. An optional minimum field width.

5. An optional precision, preceded by a '.' character.

A *repositioning* specification changes which argument to `format` is used by the current and all following format specifications. Normally the first specification uses the first argument, the second specification uses the second argument, etc. Using a repositioning specification, you can change this. By placing a number *N* followed by a '$' between the '%' and the format character, you cause the specification to use the *N*th argument. The next specification will use the *N*+1'th argument, etc.

For example:

```
(format "Can't find file `%s' in directory `%s'."
        "ignatius.c" "loyola/")
     ⇒ "Can't find file `ignatius.c' in directory `loyola/'."

(format "In directory `%2$s', the file `%1$s' was not found."
        "ignatius.c" "loyola/")
     ⇒ "In directory `loyola/', the file `ignatius.c' was not found."

(format
    "The numbers %d and %d are %1$x and %x in hex and %1$o and %o in octal."
    37 12)
⇒ "The numbers 37 and 12 are 25 and c in hex and 45 and 14 in octal."
```

As you can see, this lets you reprocess arguments more than once or reword a format specification (thereby moving the arguments around) without having to actually reorder the arguments. This is especially useful in translating messages from one language to another: Different languages use different word orders, and this sometimes entails changing the order of the arguments. By using repositioning specifications, this can be accomplished without having to embed knowledge of particular languages into the location in the program's code where the message is displayed.

All the specification characters allow an optional numeric prefix between the '%' and the character, and following any repositioning specification or flag. 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 normally on the left, but will be on the right if the '-' flag character is given. The padding character is normally a space, but if the '0' flag character is given, zeros are used for padding.

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

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

`format` never truncates an object's printed representation, no matter what width you specify. Thus, you can use a numeric prefix to specify a minimum spacing between columns with no risk of losing information.

In the following three examples, '%7s' specifies a minimum width of 7. In the first case, the string inserted in place of '%7s' has only 3 letters, so 4 blank spaces are inserted for padding. In the second case, the string `"specification"` is 13 letters wide but is not truncated. In the third case, the padding is on the right.

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

After any minimum field width, a precision may be specified by preceding it with a '.' character. The precision specifies the minimum number of digits to appear in '%d', '%i', '%o', '%x', and '%X' conversions (the number is padded on the left with zeroes as necessary); the number of digits printed after the decimal point for '%f', '%e', and '%E' conversions; the number of significant digits printed in '%g' and '%G' conversions; and the maximum number of non-padding characters printed in '%s' and '%S' conversions. The default precision for floating-point conversions is six.

The other flag characters have the following meanings:

- The ' ' flag means prefix non-negative numbers with a space.
- The '+' flag means prefix non-negative numbers with a plus sign.
- The '#' flag means print numbers in an alternate, more verbose format: octal numbers begin with zero; hex numbers begin with a '0x' or '0X'; a decimal point is printed in '%f', '%e', and '%E' conversions even if no numbers are printed after it; and trailing zeroes are not omitted in '%g' and '%G' conversions.

4.11 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 actually an integer under XEmacs 19.) 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)
```

```
     ⇒ ?x    ;; Under XEmacs 20.
     ⇒ 120   ;; Under XEmacs 19.
```

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 actually an integer under XEmacs 19.) 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)
     ⇒ ?X    ;; Under XEmacs 20.
     ⇒ 88    ;; Under XEmacs 19.
```

capitalize *string-or-char* Function

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

The definition of a word is any sequence of consecutive characters that are assigned to the word constituent syntax class in the current syntax table (see Section 38.2.1 [Syntax Class Table], page 570).

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)
     ⇒ ?X    ;; Under XEmacs 20.
     ⇒ 88    ;; Under XEmacs 19.
```

4.12 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 36.17 [Case Changes], page 538). You need a case table if you are using a language which has letters other than 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 XEmacs 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.)

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

Each buffer has a case table. XEmacs 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 the standard syntax table. See Chapter 38 [Syntax Tables], page 569.

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 standard syntax table and define a case table for the 8-bit ISO Latin 1 character set.

4.13 The Char Table

A char table is a table that maps characters (or ranges of characters) to values. Char tables are specialized for characters, only allowing particular sorts of ranges to be assigned values. Although this loses in generality, it makes for extremely fast (constant-time) lookups, and thus is feasible for applications that do an extremely large number of lookups (e.g. scanning a buffer for a character in a particular syntax, where a lookup in the syntax table must occur once per character).

Note that char tables as a primitive type, and all of the functions in this section, exist only in XEmacs 20. In XEmacs 19, char tables are generally implemented using a vector of 256 elements.

When MULE support exists, the types of ranges that can be assigned values are

- all characters
- an entire charset
- a single row in a two-octet charset
- a single character

When MULE support is not present, the types of ranges that can be assigned values are

- all characters
- a single character

char-table-p *object* Function

This function returns non-`nil` if *object* is a char table.

4.13.1 Char Table Types

Each char table type is used for a different purpose and allows different sorts of values. The different char table types are

`category` Used for category tables, which specify the regexp categories that a character is in. The valid values are `nil` or a bit vector of 95 elements. Higher-level Lisp functions are provided for working with category tables. Currently categories and category tables only exist when MULE support is present.

`char` A generalized char table, for mapping from one character to another. Used for case tables, syntax matching tables, `keyboard-translate-table`, etc. The valid values are characters.

`generic` An even more generalized char table, for mapping from a character to anything.

`display` Used for display tables, which specify how a particular character is to appear when displayed. #### Not yet implemented.

`syntax` Used for syntax tables, which specify the syntax of a particular character. Higher-level Lisp functions are provided for working with syntax tables. The valid values are integers.

char-table-type *table* Function

This function returns the type of char table *table*.

char-table-type-list Function

This function returns a list of the recognized char table types.

valid-char-table-type-p *type* Function

This function returns `t` if *type* if a recognized char table type.

4.13.2 Working With Char Tables

make-char-table *type* Function

This function makes a new, empty char table of type *type*. *type* should be a symbol, one of `char`, `category`, `display`, `generic`, or `syntax`.

put-char-table *range val table* Function

This function sets the value for chars in *range* to be *val* in *table*.

range specifies one or more characters to be affected and should be one of the following:

- `t` (all characters are affected)
- A charset (only allowed when MULE support is present)
- A vector of two elements: a two-octet charset and a row number (only allowed when MULE support is present)
- A single character

val must be a value appropriate for the type of *table*.

get-char-table *ch table* Function

This function finds the value for char *ch* in *table*.

get-range-char-table *range table* &optional *multi* Function

This function finds the value for a range in *table*. If there is more than one value, *multi* is returned (defaults to `nil`).

reset-char-table *table* Function

This function resets a char table to its default state.

map-char-table *function table* &optional *range* Function

This function maps *function* over entries in *table*, calling it with two args, each key and value in the table.

range specifies a subrange to map over and is in the same format as the *range* argument to `put-range-table`. If omitted or `t`, it defaults to the entire table.

valid-char-table-value-p *value char-table-type* Function

This function returns non-`nil` if *value* is a valid value for *char-table-type*.

check-valid-char-table-value *value char-table-type* Function

This function signals an error if *value* is not a valid value for *char-table-type*.

5 Lists

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

5.1 Lists and Cons Cells

Lists in Lisp are not a primitive data type; they are built up from *cons cells*. A cons cell is a data object that represents an ordered pair. It records two Lisp objects, one labeled as the CAR, and the other labeled as the CDR. These names are traditional; see Section 2.4.6 [Cons Cell Type], page 22. CDR is pronounced "could-er."

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

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

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

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

5.2 Lists as Linked Pairs of Boxes

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

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

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

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

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

See Section 2.4.6 [Cons Cell Type], page 22, for the read and print syntax of cons cells and lists, and for more "box and arrow" illustrations of lists.

5.3 Predicates on Lists

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

consp *object* Function

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

atom *object* Function

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

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

listp *object* Function

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

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

nlistp *object* Function

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

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

null *object* Function

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

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

5.4 Accessing Elements of Lists

car *cons-cell* Function

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

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

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

cdr *cons-cell* Function

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

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

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

car-safe *object* Function

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

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

cdr-safe *object* Function

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

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

nth *n list* Function

This function returns the *n*th element of *list*. Elements are numbered starting with zero, so the CAR of *list* is element number zero. If the length of *list* is *n* or less, the value is `nil`.

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

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

(nth n x) ≡ (car (nthcdr n x))
```

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 zero or negative, `nthcdr` returns all of *list*. If the length of *list* is *n* or less, `nthcdr` returns `nil`.

```
(nthcdr 1 '(1 2 3 4))
     ⇒ (2 3 4)
(nthcdr 10 '(1 2 3 4))
     ⇒ nil
(nthcdr -3 '(1 2 3 4))
     ⇒ (1 2 3 4)
```

Many convenience functions are provided to make it easier for you to access particular elements in a nested list. All of these can be rewritten in terms of the functions just described.

caar *cons-cell* Function
cadr *cons-cell* Function
cdar *cons-cell* Function
cddr *cons-cell* Function
caaar *cons-cell* Function
caadr *cons-cell* Function
cadar *cons-cell* Function
caddr *cons-cell* Function
cdaar *cons-cell* Function
cdadr *cons-cell* Function
cddar *cons-cell* Function
cdddr *cons-cell* Function
caaaar *cons-cell* Function
caaadr *cons-cell* Function
caadar *cons-cell* Function
caaddr *cons-cell* Function
cadaar *cons-cell* Function
cadadr *cons-cell* Function
caddar *cons-cell* Function
cadddr *cons-cell* Function
cdaaar *cons-cell* Function
cdaadr *cons-cell* Function
cdadar *cons-cell* Function
cdaddr *cons-cell* Function
cddaar *cons-cell* Function
cddadr *cons-cell* Function
cdddar *cons-cell* Function
cddddr *cons-cell* Function

Each of these functions is equivalent to one or more applications of `car` and/or `cdr`. For example,

```
(cadr x)
```

is equivalent to

```
(car (cdr x))
```

and

```
(cdaddr x)
```

is equivalent to

```
(cdr (car (cdr (cdr x))))
```

That is to say, read the a's and d's from right to left and apply a `car` or `cdr` for each a or d found, respectively.

first *list* Function

This is equivalent to `(nth 0 list)`, i.e. the first element of *list*. (Note that this is also equivalent to `car`.)

second *list* Function

This is equivalent to `(nth 1 list)`, i.e. the second element of *list*.

third *list* Function
fourth *list* Function
fifth *list* Function
sixth *list* Function
seventh *list* Function
eighth *list* Function
ninth *list* Function
tenth *list* Function

These are equivalent to `(nth 2 list)` through `(nth 9 list)` respectively, i.e. the third through tenth elements of *list*.

5.5 Building Cons Cells and Lists

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

cons *object1 object2* Function

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

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

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

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

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

list &rest *objects* Function

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

```
(list 1 2 3 4 5)
     ⇒ (1 2 3 4 5)
```

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

make-list *length object* Function

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

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

append &rest *sequences* Function

This function returns a list containing all the elements of *sequences*. The *sequences* may be lists, vectors, or strings, but the last one should be a list. All arguments except the last one are copied, so none of them are altered.

More generally, the final argument to `append` may be any Lisp object. The final argument is not copied or converted; it becomes the CDR of the last cons cell in the new list. If the final argument is itself a list, then its elements become in effect elements of the result list. If the final element is not a list, the result is a "dotted list" since its final CDR is not `nil` as required in a true list.

See `nconc` in Section 5.6.3 [Rearrangement], page 87, for a way to join lists with no copying.

Here is an example of using `append`:

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

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

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

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

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

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

This once was the usual way to copy a list, before the function `copy-sequence` was invented. See Chapter 6 [Sequences Arrays Vectors], page 101.

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
```

Here are some examples where the final argument is not a list:

```
(append '(x y) 'z)
     ⇒ (x y . z)
(append '(x y) [z])
     ⇒ (x y . [z])
```

The second example shows that when the final argument is a sequence but not a list, the sequence's elements do not become elements of the resulting list. Instead, the sequence becomes the final CDR, like any other non-list final argument.

The `append` function also allows integers as arguments. It converts them to strings of digits, making up the decimal print representation of the integer, and then uses the strings instead of the original integers. **Don't use this feature; we plan to eliminate it. If you already use this feature, change your programs now!** The proper way to convert an integer to a decimal number in this way is with `format` (see Section 4.10 [Formatting Strings], page 67) or `number-to-string` (see Section 4.7 [String Conversion], page 65).

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`. When used on a list, `setcar` replaces one element of a list with a different element.

setcar *cons object* Function

This function stores *object* as the new CAR of *cons*, replacing its previous CAR. It returns the value *object*. For example:

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

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

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

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

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

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

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

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

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

5.6.2 Altering the CDR of a List

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

setcdr *cons object* Function

This function stores *object* as the new CDR of *cons*, replacing its previous CDR. It returns the value *object*.

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

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

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

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

Here is the result in box notation:

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

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

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

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

Here is this result in box notation:

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

5.6.3 Functions that Rearrange Lists

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

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

nconc &rest *lists* Function

This function returns a list containing all the elements of *lists*. Unlike `append` (see Section 5.5 [Building Lists], page 82), the *lists* are *not* copied. Instead, the last CDR of each of the *lists* is changed to refer to the following list. The last of the *lists* is not altered. For example:

```
(setq x '(1 2 3))
     ⇒ (1 2 3)
(nconc x '(4 5))
     ⇒ (1 2 3 4 5)
x
     ⇒ (1 2 3 4 5)
```

Since the last argument of `nconc` is not itself modified, it is reasonable to use a constant list, such as `'(4 5)`, as in the above example. For the same reason, the last argument need not be a list:

```
(setq x '(1 2 3))
     ⇒ (1 2 3)
(nconc x 'z)
     ⇒ (1 2 3 . z)
x
     ⇒ (1 2 3 . z)
```

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

```
(defun add-foo (x)            ; We want this function to add
  (nconc '(foo) x))           ;   foo to the front of its arg.

(symbol-function 'add-foo)
     ⇒ (lambda (x) (nconc (quote (foo)) x))

(setq xx (add-foo '(1 2)))    ; It seems to work.
     ⇒ (foo 1 2)
(setq xy (add-foo '(3 4)))    ; What happened?
     ⇒ (foo 1 2 3 4)
(eq xx xy)
     ⇒ t

(symbol-function 'add-foo)
     ⇒ (lambda (x) (nconc (quote (foo 1 2 3 4) x)))
```

nreverse *list* Function

This function reverses the order of the elements of *list*. Unlike `reverse`, `nreverse` alters its argument by reversing the CDRs in the cons cells forming the list. The cons cell that used to be the last one in *list* becomes the first 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 make a sorted copy without destroying the original, copy it first with `copy-sequence` and then sort.

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

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

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 36.14 [Sorting], page 529, for more functions that perform sorting. See `documentation` in Section 27.2 [Accessing Documentation], page 382, for a useful example of `sort`.

5.7 Using Lists as Sets

A list can represent an unordered mathematical set—simply consider a value an element of a set if it appears in the list, and ignore the order of the list. To form the union of two sets, use `append` (as long as you don't mind having duplicate elements). Other useful functions for sets include `memq` and `delq`, and their `equal` versions, `member` and `delete`.

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

memq *object list* Function

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

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

delq *object list* Function

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

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

```
(delq 'a '(a b c)) ≡ (cdr '(a b c))
```

When an element to be deleted appears in the middle of the list, removing it involves changing the CDRs (see Section 5.6.2 [Setcdr], page 86).

```
(setq sample-list '(a b c (4)))
     ⇒ (a b c (4))
(delq 'a sample-list)
     ⇒ (b c (4))
sample-list
     ⇒ (a b c (4))
(delq 'c sample-list)
     ⇒ (a b (4))
sample-list
     ⇒ (a b (4))
```

Note that `(delq 'c sample-list)` modifies `sample-list` to splice out the third element, but `(delq 'a sample-list)` does not splice anything—it just returns a shorter list. Don't assume that a variable which formerly held the argument *list* now has fewer elements, or that it still holds the original list! Instead, save the result of `delq` and use that. Most often we store the result back into the variable that held the original list:

```
(setq flowers (delq 'rose flowers))
```

In the following example, the `(4)` that `delq` attempts to match and the `(4)` in the `sample-list` are not `eq`:

```
(delq '(4) sample-list)
     ⇒ (a c (4))
```

The following two functions are like `memq` and `delq` but use `equal` rather than `eq` to compare elements. 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, `member` returns a list starting with its first occurrence in *list*. Otherwise, it returns `nil`.

Compare this with `memq`:

```
(member '(2) '((1) (2)))  ; (2) and (2) are equal.
     ⇒ ((2))
(memq '(2) '((1) (2)))    ; (2) and (2) are not eq.
     ⇒ nil
;; Two strings with the same contents are equal.
(member "foo" '("foo" "bar"))
     ⇒ ("foo" "bar")
```

delete *object list* Function

This function destructively removes all elements `equal` to *object* from *list*. It is to `delq` as `member` is to `memq`: it uses `equal` to compare elements with *object*, like `member`; when it finds an element that matches, it removes the element just as `delq` would. For example:

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

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

See also the function `add-to-list`, in Section 10.7 [Setting Variables], page 152, for another way to add an element to a list stored in a variable.

5.8 Association Lists

An *association list*, or *alist* for short, records a mapping from keys to values. It is a list of cons cells called *associations*: the CAR of each cell is the *key*, and the CDR is the *associated value*.[1]

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

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

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

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

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

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

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 XEmacs 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 5.9 [Property Lists], page 96, 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 2.8 [Equality Predicates], page 41). It returns `nil` if no association in *alist* has a CAR `equal` to *key*. For example:

[1] This usage of "key" is not related to the term "key sequence"; it means a value used to look up an item in a table. In this case, the table is the alist, and the alist associations are the items.

```
(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")
```

rassoc *value alist* Function

This function returns the first association with value *value* in *alist*. It returns `nil` if no association in *alist* has a CDR `equal` to *value*.

`rassoc` is like `assoc` except that it compares the CDR of each *alist* association instead of the CAR. You can think of this as "reverse `assoc`", finding the key for a given value.

assq *key alist* Function

This function is like `assoc` in that it returns the first association for *key* in *alist*, but it makes the comparison using `eq` instead of `equal`. `assq` returns `nil` if no association in *alist* has a CAR `eq` to *key*. This function is used more often than `assoc`, since `eq` is faster than `equal` and most alists use symbols as keys. See Section 2.8 [Equality Predicates], page 41.

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

On the other hand, `assq` is not usually useful in alists where the keys may not be symbols:

```
(setq leaves
      '(("simple leaves" . oak)
        ("compound leaves" . horsechestnut)))

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

rassq *value alist* Function

This function returns the first association with value *value* in *alist*. It returns `nil` if no association in *alist* has a CDR `eq` to *value*.

`rassq` is like `assq` except that it compares the CDR of each *alist* association instead of the CAR. You can think of this as "reverse `assq`", finding the key for a given value.

For example:

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

(rassq 'acorns trees)
     ⇒ (oak . acorns)
(rassq 'spores trees)
     ⇒ nil
```

Note that `rassq` cannot search for a value stored in the CAR of the CDR of an element:

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

(rassq 'white colors)
     ⇒ nil
```

In this case, the CDR of the association `(lily white)` is not the symbol `white`, but rather the list `(white)`. This becomes clearer if the association is written in dotted pair notation:

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

remassoc *key alist* Function

This function deletes by side effect any associations with key *key* in *alist* – i.e. it removes any elements from *alist* whose `car` is `equal` to *key*. The modified *alist* is returned.

If the first member of *alist* has a `car` that is `equal` to *key*, there is no way to remove it by side effect; therefore, write `(setq foo (remassoc key foo))` to be sure of changing the value of `foo`.

remassq *key alist* Function

This function deletes by side effect any associations with key *key* in *alist* – i.e. it removes any elements from *alist* whose `car` is `eq` to *key*. The modified *alist* is returned.

This function is exactly like `remassoc`, but comparisons between *key* and keys in *alist* are done using `eq` instead of `equal`.

remrassoc *value alist* Function

This function deletes by side effect any associations with value *value* in *alist* – i.e. it removes any elements from *alist* whose `cdr` is `equal` to *value*. The modified *alist* is returned.

If the first member of *alist* has a `car` that is `equal` to *value*, there is no way to remove it by side effect; therefore, write `(setq foo (remassoc value foo))` to be sure of changing the value of `foo`.

`remrassoc` is like `remassoc` except that it compares the CDR of each *alist* association instead of the CAR. You can think of this as "reverse `remassoc`", removing an association based on its value instead of its key.

remrassq *value alist* Function

This function deletes by side effect any associations with value *value* in *alist* – i.e. it removes any elements from *alist* whose `cdr` is `eq` to *value*. The modified *alist* is returned.

This function is exactly like `remrassoc`, but comparisons between *value* and values in *alist* are done using `eq` instead of `equal`.

copy-alist *alist* Function

This function returns a two-level deep copy of *alist*: it creates a new copy of each association, so that you can alter the associations of the new alist without changing the old one.

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

(setq copy (copy-alist needles-per-cluster))
⇒
((2 "Austrian Pine" "Red Pine")
 (3 "Pitch Pine")
 (5 "White Pine"))

(eq needles-per-cluster copy)
     ⇒ nil
(equal needles-per-cluster copy)
     ⇒ t
(eq (car needles-per-cluster) (car copy))
     ⇒ nil
(cdr (car (cdr needles-per-cluster)))
     ⇒ ("Pitch Pine")
(eq (cdr (car (cdr needles-per-cluster)))
    (cdr (car (cdr copy))))
     ⇒ t
```

This example shows how `copy-alist` makes it possible to change the associations of one copy without affecting the other:

```
(setcdr (assq 3 copy) '("Martian Vacuum Pine"))
(cdr (assq 3 needles-per-cluster))
     ⇒ ("Pitch Pine")
```

5.9 Property Lists

A *property list* (or *plist*) is another way of representing a mapping from keys to values. Instead of the list consisting of conses of a key and a value, the keys and values alternate as successive entries in the list. Thus, the association list

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

has the equivalent property list form

```
(a 1 b 2 c 3)
```

Property lists are used to represent the properties associated with various sorts of objects, such as symbols, strings, frames, etc. The convention is that property lists can be modified in-place, while association lists generally are not.

Plists come in two varieties: *normal* plists, whose keys are compared with `eq`, and *lax* plists, whose keys are compared with `equal`,

valid-plist-p *plist* Function

Given a plist, this function returns non-`nil` if its format is correct. If it returns `nil`, `check-valid-plist` will signal an error when given the plist; that means it's a malformed or circular plist or has non-symbols as keywords.

check-valid-plist *plist* Function

Given a plist, this function signals an error if there is anything wrong with it. This means that it's a malformed or circular plist.

5.9.1 Working With Normal Plists

plist-get *plist prop* &optional *default* Function

This function extracts a value from a property list. The function returns the value corresponding to the given *prop*, or *default* if *prop* is not one of the properties on the list.

plist-put *plist prop val* Function

This function changes the value in *plist* of *prop* to *val*. If *prop* is already a property on the list, its value is set to *val*, otherwise the new *prop val* pair is added. The new plist is returned; use `(setq x (plist-put x prop val))` to be sure to use the new value. The *plist* is modified by side effects.

plist-remprop *plist prop* Function

This function removes from *plist* the property *prop* and its value. The new plist is returned; use `(setq x (plist-remprop x prop val))` to be sure to use the new value. The *plist* is modified by side effects.

plist-member *plist prop* Function

This function returns `t` if *prop* has a value specified in *plist*.

In the following functions, if optional arg *nil-means-not-present* is non-`nil`, then a property with a `nil` value is ignored or removed. This feature is a virus that has infected old Lisp implementations (and thus E-Lisp, due to RMS's enamorment with old Lisps), but should not be used except for backward compatibility.

plists-eq *a b* &optional *nil-means-not-present* Function

This function returns non-`nil` if property lists A and B are `eq` (i.e. their values are `eq`).

plists-equal *a b* &optional *nil-means-not-present* Function

This function returns non-`nil` if property lists A and B are `equal` (i.e. their values are `equal`; their keys are still compared using `eq`).

canonicalize-plist *plist* &optional *nil-means-not-present* Function

This function destructively removes any duplicate entries from a plist. In such cases, the first entry applies.

The new plist is returned. If *nil-means-not-present* is given, the return value may not be `eq` to the passed-in value, so make sure to `setq` the value back into where it came from.

5.9.2 Working With Lax Plists

Recall that a *lax plist* is a property list whose keys are compared using `equal` instead of `eq`.

lax-plist-get *lax-plist prop* &optional *default* Function

This function extracts a value from a lax property list. The function returns the value corresponding to the given *prop*, or *default* if *prop* is not one of the properties on the list.

lax-plist-put *lax-plist prop val* Function

This function changes the value in *lax-plist* of *prop* to *val*.

lax-plist-remprop *lax-plist prop* Function

This function removes from *lax-plist* the property *prop* and its value. The new plist is returned; use `(setq x (lax-plist-remprop x prop val))` to be sure to use the new value. The *lax-plist* is modified by side effects.

lax-plist-member *lax-plist prop* Function

This function returns `t` if *prop* has a value specified in *lax-plist*.

In the following functions, if optional arg *nil-means-not-present* is non-`nil`, then a property with a `nil` value is ignored or removed. This feature is a virus that has infected old Lisp implementations (and thus E-Lisp, due to RMS's enamorment with old Lisps), but should not be used except for backward compatibility.

lax-plists-eq *a b* &optional *nil-means-not-present* Function

This function returns non-`nil` if lax property lists A and B are `eq` (i.e. their values are `eq`; their keys are still compared using `equal`).

lax-plists-equal *a b* &optional *nil-means-not-present* Function

This function returns non-`nil` if lax property lists A and B are `equal` (i.e. their values are `equal`).

canonicalize-lax-plist *lax-plist* &optional *nil-means-not-present* Function

This function destructively removes any duplicate entries from a lax plist. In such cases, the first entry applies.

The new plist is returned. If *nil-means-not-present* is given, the return value may not be `eq` to the passed-in value, so make sure to `setq` the value back into where it came from.

5.9.3 Converting Plists To/From Alists

alist-to-plist *alist* Function

This function converts association list *alist* into the equivalent property-list form. The plist is returned. This converts from

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

into

```
(a 1 b 2 c 3)
```

The original alist is not modified.

plist-to-alist *plist* Function

This function converts property list *plist* into the equivalent association-list form. The alist is returned. This converts from

```
(a 1 b 2 c 3)
```

into

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

The original plist is not modified.

The following two functions are equivalent to the preceding two except that they destructively modify their arguments, using cons cells from the original list to form the new list rather than allocating new cons cells.

destructive-alist-to-plist *alist* Function

This function destructively converts association list *alist* into the equivalent property-list form. The plist is returned.

destructive-plist-to-alist *plist* Function

This function destructively converts property list *plist* into the equivalent association-list form. The alist is returned.

5.10 Weak Lists

A *weak list* is a special sort of list whose members are not counted as references for the purpose of garbage collection. This means that, for any object in the list, if there are no references to the object anywhere outside of the list (or other weak list or weak hash table), that object will disappear the next time a garbage collection happens. Weak lists can be useful for keeping track of things such as unobtrusive lists of another function's buffers or markers. When that function is done with the elements, they will automatically disappear from the list.

Weak lists are used internally, for example, to manage the list holding the children of an extent – an extent that is unused but has a parent will still be reclaimed, and will automatically be removed from its parent's list of children.

Weak lists are similar to weak hash tables (see Section 46.3 [Weak Hash Tables], page 666).

weak-list-p *object* Function

This function returns non-`nil` if *object* is a weak list.

Weak lists come in one of four types:

`simple` Objects in the list disappear if not referenced outside of the list.

`assoc` Objects in the list disappear if they are conses and either the car or the cdr of the cons is not referenced outside of the list.

`key-assoc`
Objects in the list disappear if they are conses and the car is not referenced outside of the list.

`value-assoc`
Objects in the list disappear if they are conses and the cdr is not referenced outside of the list.

make-weak-list &optional *type* Function

This function creates a new weak list of type *type*. *type* is a symbol (one of `simple`, `assoc`, `key-assoc`, or `value-assoc`, as described above) and defaults to `simple`.

weak-list-type *weak* Function

This function returns the type of the given weak-list object.

weak-list-list *weak* Function

This function returns the list contained in a weak-list object.

set-weak-list-list *weak new-list* Function

This function changes the list contained in a weak-list object.

6 Sequences, Arrays, and Vectors

Recall that the *sequence* type is the union of four other Lisp types: lists, vectors, bit vectors, and strings. In other words, any list is a sequence, any vector is a sequence, any bit 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 that has a slot for each elements. All the elements are accessible in constant time, but the length of an existing array cannot be changed. Strings, vectors, and bit vectors are the three types of arrays.

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

The following diagram shows the relationship between these types:

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

The elements of vectors and lists may be any Lisp objects. The elements of strings are all characters. The elements of bit vectors are the numbers 0 and 1.

6.1 Sequences

In XEmacs Lisp, a *sequence* is either a list, a vector, a bit 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, bit 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 extents or text properties, the extents and text properties in the copy are also copied, not shared with the original. (This means that modifying the extents or text properties of the original will not affect the copy.) However, the actual values of the properties are shared. See Chapter 40 [Extents], page 585, See Section 36.18 [Text Properties], page 539.

See also `append` in Section 5.5 [Building Lists], page 82, `concat` in Section 4.3 [Creating Strings], page 60, `vconcat` in Section 6.4 [Vectors], page 106, and `bvconcat` in Section 6.6 [Bit Vectors], page 107, for other ways to copy sequences.

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

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

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

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

;; Creating a bit vector.
(bit-vector 1 0 1 1 0 1 0 0)
     ⇒ #*10110100
```

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
(length #*01101)
    ⇒ 5
```

elt *sequence index* Function

This function returns the element of *sequence* indexed by *index*. Legitimate values of *index* are integers ranging from 0 up to one less than the length of *sequence*. If *sequence* is a list, then out-of-range values of *index* return `nil`; otherwise, they trigger an `args-out-of-range` error.

```
(elt [1 2 3 4] 2)
     ⇒ 3
(elt '(1 2 3 4) 2)
     ⇒ 3
(char-to-string (elt "1234" 2))
     ⇒ "3"
(elt #*00010000 3)
     ⇒ 1
(elt [1 2 3 4] 4)
     error Args out of range: [1 2 3 4], 4
(elt [1 2 3 4] -1)
     error Args out of range: [1 2 3 4], -1
```

This function generalizes `aref` (see Section 6.3 [Array Functions], page 104) and `nth` (see Section 5.4 [List Elements], page 79).

6.2 Arrays

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

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.

XEmacs defines three types of array, all of which are one-dimensional: *strings*, *vectors*, and *bit 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. A bit vector is another specialized array; its elements must be bits (an integer, either 0 or 1). Each type of array has its own read syntax. See Section 2.4.8 [String Type], page 25, Section 2.4.9 [Vector Type], page 26, and Section 2.4.10 [Bit Vector Type], page 26.

All kinds of array share these characteristics:

- The first element of an array has index zero, the second element has index 1, and so on. This is called *zero-origin* indexing. For example, an array of four elements has indices 0, 1, 2, and 3.

* The elements of an array may be referenced or changed with the functions `aref` and `aset`, respectively (see Section 6.3 [Array Functions], page 104).

In principle, if you wish to have an array of text characters, you could use either a string or a vector. In practice, we always choose strings for such applications, for four reasons:

* They usually occupy one-fourth the space of a vector of the same elements. (This is one-eighth the space for 64-bit machines such as the DEC Alpha, and may also be different when MULE support is compiled into XEmacs.)
* Strings are printed in a way that shows the contents more clearly as characters.
* Strings can hold extent and text properties. See Chapter 40 [Extents], page 585, See Section 36.18 [Text Properties], page 539.
* Many of the specialized editing and I/O facilities of XEmacs accept only strings. For example, you cannot insert a vector of characters into a buffer the way you can insert a string. See Chapter 4 [Strings and Characters], page 59.

By contrast, for an array of keyboard input characters (such as a key sequence), a vector may be necessary, because many keyboard input characters are non-printable and are represented with symbols rather than with characters. See Section 19.6.1 [Key Sequence Input], page 301.

Similarly, when representing an array of bits, a bit vector has the following advantages over a regular vector:

* They occupy 1/32nd the space of a vector of the same elements. (1/64th on 64-bit machines such as the DEC Alpha.)
* Bit vectors are printed in a way that shows the contents more clearly as bits.

6.3 Functions that Operate on Arrays

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

arrayp *object* Function

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

```
(arrayp "asdf")
⇒ t
(arrayp [a])
⇒ t
(arrayp #*101)
⇒ 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)
     ⇒ ?b
(aref #*1101 2)
     ⇒ 0
```

See also the function `elt`, in Section 6.1 [Sequence Functions], page 101.

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)
     ⇒ ?Z
x
     ⇒ "asdZasfd"

(setq bv #*1111)
     ⇒ #*1111
(aset bv 2 0)
     ⇒ 0
bv
     ⇒ #*1101
```

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 *object*, so that each element of *array* is *object*. It returns *array*.

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

(setq bv #*1101)
     ⇒ #*1101
(fillarray bv 0)
     ⇒ #*0000
```

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 101.

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 in XEmacs Lisp is the *string*, whose elements must be characters.) Vectors in XEmacs serve as obarrays (vectors of symbols), although this is a shortcoming that should be fixed. They are also used internally as part of the representation of a byte-compiled function; if you print such a function, you will see a vector in it.

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

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

A vector, like a string or a number, is considered a constant for evaluation: the result of evaluating it is the same vector. This does not evaluate or even examine the elements of the vector. See Section 8.2.1 [Self-Evaluating Forms], page 122.

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
```

6.5 Functions That Operate on Vectors

Here are some functions that relate to vectors:

vectorp *object* Function

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

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

vector &rest *objects* Function

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

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

make-vector *length object* Function

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

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

vconcat &rest *sequences* Function

This function returns a new vector containing all the elements of the *sequences*. The arguments *sequences* may be lists, vectors, or strings. If no *sequences* are given, an empty vector is returned.

The value is a newly constructed vector that is not `eq` to any existing vector.

```
(setq a (vconcat '(A B C) '(D E F)))
     ⇒ [A B C D E F]
(eq a (vconcat a))
     ⇒ nil
(vconcat)
     ⇒ []
(vconcat [A B C] "aa" '(foo (6 7)))
     ⇒ [A B C 97 97 foo (6 7)]
```

The `vconcat` function also allows integers as arguments. It converts them to strings of digits, making up the decimal print representation of the integer, and then uses the strings instead of the original integers. **Don't use this feature; we plan to eliminate it. If you already use this feature, change your programs now!** The proper way to convert an integer to a decimal number in this way is with `format` (see Section 4.10 [Formatting Strings], page 67) or `number-to-string` (see Section 4.7 [String Conversion], page 65).

For other concatenation functions, see `mapconcat` in Section 11.6 [Mapping Functions], page 171, `concat` in Section 4.3 [Creating Strings], page 60, `append` in Section 5.5 [Building Lists], page 82, and `bvconcat` in Section 6.7 [Bit Vector Functions], page 108.

The `append` function provides a way to convert a vector into a list with the same elements (see Section 5.5 [Building Lists], page 82):

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

6.6 Bit Vectors

Bit vectors are specialized vectors that can only represent arrays of 1's and 0's. Bit vectors have a very efficient representation and are useful for representing sets of boolean (true or false) values.

There is no limit on the size of a bit vector. You could, for example, create a bit vector with 100,000 elements if you really wanted to.

Bit vectors have a special printed representation consisting of '#*' followed by the bits of the vector. For example, a bit vector whose elements are 0, 1, 1, 0, and 1, respectively, is printed as

```
#*01101
```

Bit vectors are considered constants for evaluation, like vectors, strings, and numbers. See Section 8.2.1 [Self-Evaluating Forms], page 122.

6.7 Functions That Operate on Bit Vectors

Here are some functions that relate to bit vectors:

bit-vector-p *object* Function

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

```
(bit-vector-p #*01)
     ⇒ t
(bit-vector-p [0 1])
     ⇒ nil
(bit-vector-p "01")
     ⇒ nil
```

bitp *object* Function

This function returns `t` if *object* is either 0 or 1.

bit-vector &rest *objects* Function

This function creates and returns a bit vector whose elements are the arguments *objects*. The elements must be either of the two integers 0 or 1.

```
(bit-vector 0 0 0 1 0 0 0 0 1 0)
     ⇒ #*0001000010
(bit-vector)
     ⇒ #*
```

make-bit-vector *length object* Function

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

```
(setq picket-fence (make-bit-vector 9 1))
     ⇒ #*111111111
```

bvconcat &rest *sequences* Function

This function returns a new bit vector containing all the elements of the *sequences*. The arguments *sequences* may be lists, vectors, or bit vectors, all of whose elements are the integers 0 or 1. If no *sequences* are given, an empty bit vector is returned.

The value is a newly constructed bit vector that is not `eq` to any existing bit vector.

```
(setq a (bvconcat '(1 1 0) '(0 0 1)))
     ⇒ #*110001
(eq a (bvconcat a))
     ⇒ nil
```

```
(bvconcat)
     ⇒ #*
(bvconcat [1 0 0 0 0] #*111 '(0 0 0 0 1))
     ⇒ #*1000011100001
```

For other concatenation functions, see `mapconcat` in Section 11.6 [Mapping Functions], page 171, `concat` in Section 4.3 [Creating Strings], page 60, `vconcat` in Section 6.5 [Vector Functions], page 106, and `append` in Section 5.5 [Building Lists], page 82.

The `append` function provides a way to convert a bit vector into a list with the same elements (see Section 5.5 [Building Lists], page 82):

```
(setq bv #*00001110)
     ⇒ #*00001110
(append bv nil)
     ⇒ (0 0 0 0 1 1 1 0)
```

7 Symbols

A *symbol* is an object with a unique name. This chapter describes symbols, their components, their property lists, and how they are created and interned. Separate chapters describe the use of symbols as variables and as function names; see Chapter 10 [Variables], page 145, and Chapter 11 [Functions], page 163. For the precise read syntax for symbols, see Section 2.4.4 [Symbol Type], page 21.

You can test whether an arbitrary Lisp object is a symbol with `symbolp`:

symbolp *object* Function

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

7.1 Symbol Components

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

Print name
: The *print name cell* holds a string that names the symbol for reading and printing. See `symbol-name` in Section 7.3 [Creating Symbols], page 113.

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 151.

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 173.

Property list
: The *property list cell* holds the property list of the symbol. See `symbol-plist` in Section 7.4 [Symbol Properties], page 116.

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 XEmacs Lisp, this lookup uses a hashing algorithm and an obarray; see Section 7.3 [Creating Symbols], page 113.)

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 119). Keyboard macros (see Section 19.13 [Keyboard Macros], page 313), keymaps (see Chapter 20 [Keymaps], page 315) and autoload objects (see Section 8.2.8 [Autoloading], page 126) 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.

The property list cell normally should hold a correctly formatted property list (see Section 5.9 [Property Lists], page 96), as a number of functions expect to see a property list there.

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

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

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

Because this symbol is the variable which holds the name of the file being visited in the current buffer, the value cell contents we see are the name of the source file of this chapter of the XEmacs Lisp Manual. The property list cell contains the list `(variable-documentation 29529)` which tells the documentation functions where to find the documentation string for the variable `buffer-file-name` in the '`DOC`' file. (29529 is the offset from the beginning of the '`DOC`' file to where that documentation string 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 2.4.13 [Primitive Function Type], page 27). 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 XEmacs 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 149.

`defun` defines a symbol as a function, creating a lambda expression and storing it in the function cell of the symbol. This lambda expression thus becomes the function definition of the symbol. (The term "function definition", meaning the contents of the function cell, is derived from the idea that `defun` gives the symbol its definition as a function.) `defsubst`, `define-function` and `defalias` are other ways of defining a function. See Chapter 11 [Functions], page 163.

`defmacro` defines a symbol as a macro. It creates a macro object and stores it in the function cell of the symbol. Note that a given symbol can be a macro or a function, but not both at once, because both macro and function definitions are kept in the function cell, and that cell can hold only one Lisp object at any given time. See Chapter 12 [Macros], page 177.

In XEmacs Lisp, a definition is not required in order to use a symbol as a variable or function. Thus, you can make a symbol a global variable with `setq`, whether you define it first or not. The real purpose of definitions is to guide programmers and programming tools. They inform programmers who read the code that certain symbols are *intended* to be used as variables, or as functions. In addition, utilities such as '`etags`' and '`make-docfile`' recognize definitions, and add appropriate information to tag tables and the '`DOC`' file. See Section 27.2 [Accessing Documentation], page 382.

7.3 Creating and Interning Symbols

To understand how symbols are created in XEmacs Lisp, you must know how Lisp reads them. Lisp must ensure that it finds the same symbol every time it reads the same set of characters. Failure to do so would cause complete confusion.

When the Lisp reader encounters a symbol, it reads all the characters of the name. Then it "hashes" those characters to find an index in a table called an *obarray*. Hashing is an efficient method of looking something up. For example, instead of searching a telephone book cover to cover when looking up Jan Jones, you start with the J's and go from there. That is a simple version of hashing. Each element of the obarray is a *bucket* which holds all the symbols with a given hash code; to look for a given name, it is sufficient to look through all the symbols in the bucket for that name's hash code.

If a symbol with the desired name is found, the reader uses that symbol. If the obarray does not contain a symbol with that name, the reader makes a new symbol and adds it to the obarray. Finding or adding a symbol with a certain name is called *interning* it, and the symbol is then called an *interned symbol*.

Interning ensures that each obarray has just one symbol with any particular name. Other like-named symbols may exist, but not in the same obarray. Thus, the reader gets the same symbols for the same names, as long as you keep reading with the same obarray.

No obarray contains all symbols; in fact, some symbols are not in any obarray. They are called *uninterned symbols*. An uninterned symbol has the same four cells as other symbols; however, the only way to gain access to it is by finding it in some other object or as the value of a variable.

In XEmacs Lisp, an obarray is actually a vector. Each element of the vector is a bucket; its value is either an interned symbol whose name hashes to that bucket, or 0 if the bucket is empty. Each interned symbol has an internal link (invisible to the user) to the next symbol in the bucket. Because these links are invisible, there is no way to find all the symbols in an obarray except using `mapatoms` (below). The order of symbols in a bucket is not significant.

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

Do not try to put symbols in an obarray yourself. This does not work—only `intern` can enter a symbol in an obarray properly. **Do not try to intern one symbol in two obarrays.** This would garble both obarrays, because a symbol has just one slot to hold the following symbol in the obarray bucket. The results would be unpredictable.

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 the abbrev mechanism (see Chapter 39 [Abbrevs], page 579).

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

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

symbol-name *symbol* — Function

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

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

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 *obarray*, `intern` creates a new one, adds it to the obarray, and returns it. If *obarray* is omitted, the value of the global variable `obarray` is used.

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

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

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

This function returns the symbol in *obarray* whose name is *name*, or `nil` if *obarray* has no symbol with that name. Therefore, you can use `intern-soft` to test whether a symbol with a given name is already interned. If *obarray* is omitted, the value of the global variable `obarray` is used.

```
(intern-soft "frazzle")        ; No such symbol exists.
     ⇒ nil
(make-symbol "frazzle")        ; Create an uninterned one.
     ⇒ frazzle
(intern-soft "frazzle")        ; That one cannot be found.
     ⇒ nil
(setq sym (intern "frazzle"))  ; Create an interned one.
     ⇒ frazzle
(intern-soft "frazzle")        ; That one can be found!
     ⇒ frazzle
(eq sym 'frazzle)              ; And it is the same one.
     ⇒ t
```

obarray Variable

This variable is the standard obarray for use by `intern` and `read`.

mapatoms *function* &optional *obarray* Function

This function calls *function* for each symbol in the obarray *obarray*. It returns `nil`. If *obarray* is omitted, it defaults to the value of `obarray`, the standard obarray for ordinary symbols.

```
(setq count 0)
     ⇒ 0
(defun count-syms (s)
  (setq count (1+ count)))
     ⇒ count-syms
(mapatoms 'count-syms)
     ⇒ nil
count
     ⇒ 1871
```

See `documentation` in Section 27.2 [Accessing Documentation], page 382, for another example using `mapatoms`.

unintern *symbol* &optional *obarray* Function

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

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

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

7.4 Symbol Properties

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

Many objects other than symbols can have property lists associated with them, and XEmacs provides a full complement of functions for working with property lists. See Section 5.9 [Property Lists], page 96.

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.

7.4.1 Property Lists and Association Lists

Association lists (see Section 5.8 [Association Lists], page 92) are very similar to property lists. In contrast to association lists, the order of the pairs in the property list is not significant since the property names must be distinct.

Property lists are better than association lists for attaching information to various Lisp function names or variables. If all the associations 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. This is why the documentation for a variable is recorded in a property named `variable-documentation`. The byte compiler likewise uses properties to record those functions needing special treatment.

However, association lists have their own advantages. Depending on your application, it may be faster to add an association to the front of an association list than to update a property. All properties for a symbol are stored in the same property list, so there is a possibility of a conflict between different uses of a property name. (For this reason, it is a good idea to choose property names that are probably unique, such as by 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.

7.4.2 Property List Functions for Symbols

symbol-plist *symbol* Function

This function returns the property list of *symbol*.

setplist *symbol plist* Function

This function sets *symbol*'s property list to *plist*. Normally, *plist* should be a well-formed property list, but this is not enforced.

```
(setplist 'foo '(a 1 b (2 3) c nil))
     ⇒ (a 1 b (2 3) c nil)
(symbol-plist 'foo)
     ⇒ (a 1 b (2 3) c nil)
```

For symbols in special obarrays, which are not used for ordinary purposes, it may make sense to use the property list cell in a nonstandard fashion; in fact, the abbrev mechanism does so (see Chapter 39 [Abbrevs], page 579).

get *symbol property* Function

This function finds the value of the property named *property* in *symbol*'s property list. If there is no such property, `nil` is returned. Thus, there is no distinction between a value of `nil` and the absence of the property.

The name *property* is compared with the existing property names using `eq`, so any object is a legitimate property.

See `put` for an example.

put *symbol property value* Function

This function puts *value* onto *symbol*'s property list under the property name *property*, replacing any previous property value. The `put` function returns *value*.

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

7.4.3 Property Lists Outside Symbols

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

getf *plist property* &optional *default* Function

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

```
(getf '(foo 4) 'foo)
     ⇒ 4
```

putf *plist property value* Function

This stores *value* as the value of the *property* property in the property list *plist*. It may modify *plist* destructively, or it may construct a new list structure without altering the old. The function returns the modified property list, so you can store that back in the place where you got *plist*. For example,

```
(setq my-plist '(bar t foo 4))
     ⇒ (bar t foo 4)
(setq my-plist (putf my-plist 'foo 69))
     ⇒ (bar t foo 69)
(setq my-plist (putf my-plist 'quux '(a)))
     ⇒ (quux (a) bar t foo 5)
```

plists-eq *a b* Function

This function returns non-`nil` if property lists *a* and *b* are `eq`. This means that the property lists have the same values for all the same properties, where comparison between values is done using `eq`.

plists-equal *a b* Function

This function returns non-`nil` if property lists *a* and *b* are `equal`.

Both of the above functions do order-insensitive comparisons.

```
(plists-eq '(a 1 b 2 c nil) '(b 2 a 1))
     ⇒ t
(plists-eq '(foo "hello" bar "goodbye") '(bar "goodbye" foo "hello"))
     ⇒ nil
(plists-equal '(foo "hello" bar "goodbye") '(bar "goodbye" foo "hello"))
     ⇒ t
```

8 Evaluation

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

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

It is very common to read a Lisp expression and then evaluate the expression, but reading and evaluation are separate activities, and either can be performed alone. Reading per se does not evaluate anything; it converts the printed representation of a Lisp object to the object itself. It is up to the caller of `read` whether this object is a form to be evaluated, or serves some entirely different purpose. See Section 17.3 [Input Functions], page 253.

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 19 [Command Loop], page 281.

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

Evaluation of a function call ultimately calls the function specified in it. See Chapter 11 [Functions], page 163. The execution of the function may itself work by evaluating the function definition; or the function may be a Lisp primitive implemented in C, or it may be a byte-compiled function (see Chapter 15 [Byte Compilation], page 205).

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 145.

Evaluation of a form may create new environments for recursive evaluation by binding variables (see Section 10.3 [Local Variables], page 146). These environments are temporary and vanish by the time evaluation of the form is complete. The form may also make changes that persist; these changes are called *side effects*. An example of a form that produces side effects is `(setq foo 1)`.

The details of what evaluation means for each kind of form are described below (see Section 8.2 [Forms], page 121).

---

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

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.

Please note: it is generally cleaner and more flexible to call functions that are stored in data structures, rather than to evaluate expressions stored in data structures. Using functions provides the ability to pass information to them as arguments.

The functions and variables described in this section evaluate forms, specify limits to the evaluation process, or record recently returned values. Loading a file also does evaluation (see Chapter 14 [Loading], page 195).

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 121).

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
(eval 'foo)
     ⇒ bar
```

The number of currently active calls to `eval` is limited to `max-lisp-eval-depth` (see below).

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 during the evaluation.

You can use the variable `load-read-function` to specify a function for `eval-region` to use instead of `read` for reading expressions. See Section 14.1 [How Programs Do Loading], page 195.

`eval-region` always returns `nil`.

eval-buffer *buffer* &optional *stream* Command

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

max-lisp-eval-depth Variable

This variable defines the maximum depth allowed in calls to `eval`, `apply`, and `funcall` before an error is signaled (with error message `"Lisp nesting exceeds max-lisp-eval-depth"`). This counts internal uses of those functions, such as for calling the functions mentioned in Lisp expressions, and recursive evaluation of function call arguments and function body forms.

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 500. 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 146.

values Variable

The value of this variable is a list of the values returned by all the expressions that were read from buffers (including the minibuffer), evaluated, and printed. The elements are ordered most recent first.

```
(setq x 1)
     ⇒ 1
(list 'A (1+ 2) auto-save-default)
     ⇒ (A 3 t)
values
     ⇒ ((A 3 t) 1 ...)
```

This variable is useful for referring back to values of forms recently evaluated. It is generally a bad idea to print the value of `values` itself, since this may be very long. Instead, examine particular elements, like this:

```
;; Refer to the most recent evaluation result.
(nth 0 values)
     ⇒ (A 3 t)
;; That put a new element on,
;;   so all elements move back one.
(nth 1 values)
     ⇒ (A 3 t)
;; This gets the element that was next-to-most-recent
;;   before this example.
(nth 3 values)
     ⇒ 1
```

8.2 Kinds of Forms

A Lisp object that is intended to be evaluated is called a *form*. How XEmacs evaluates a form depends on its data type. XEmacs has three different kinds of form that are evaluated differently: symbols, lists, and "all other types". This section describes all three kinds, 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 there's no way to write them textually. It is possible to construct Lisp expressions containing these types by means of a Lisp program. Here is an example:

```
;; Build an expression containing a buffer object.
(setq 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 145.

In the following example, we set the value of a symbol with `setq`. Then we evaluate the symbol, and get back the value that `setq` stored.

```
(setq a 123)
     ⇒ 123
(eval 'a)
     ⇒ 123
a
     ⇒ 123
```

The symbols `nil` and `t` are treated specially, so that the value of `nil` is always `nil`, and the value of `t` is always `t`; you cannot set or bind them to any other values. Thus, these two symbols act like self-evaluating forms, even though `eval` treats them like any other symbol.

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 remaining list elements constitute the *arguments* for the function, macro, or special form.

The first step in evaluating a nonempty list is to examine its first element. This element alone determines what kind of form the list is and how the rest of the list is to be processed. The first element is *not* evaluated, as it would be in some Lisp dialects such as Scheme.

8.2.4 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 it obtains a non-symbol. See Section 11.3 [Function Names], page 167, for more information about using a symbol as a name for a function stored in the function cell of the symbol.

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

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

The following example illustrates the symbol indirection process. We use `fset` to set the function cell of a symbol and `symbol-function` to get the function cell contents (see Section 11.8 [Function Cells], page 173). Specifically, we store the symbol `car` into the function cell of `first`, and the symbol `first` into the function cell of `erste`.

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

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

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

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

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

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

indirect-function *function* Function

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

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

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

8.2.5 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)
```

The first step in evaluating a function call is to evaluate the remaining elements of the list from left to right. The results are the actual argument values, one value for each list element. The next step is to call the function with this list of arguments, effectively using the function `apply` (see Section 11.5 [Calling Functions], page 169). 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 164); then the forms in the function body are evaluated in order, and the value of the last body form becomes the value of the function call.

8.2.6 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, to be evaluated in place of the original form. The expansion may be any sort of form: a self-evaluating constant, a symbol, or a list. If the expansion is itself a macro call, this process of expansion repeats until some other sort of form results.

Ordinary evaluation of a macro call finishes by evaluating the expansion. However, the macro expansion is not necessarily evaluated right away, or at all, because other programs also expand macro calls, and they may or may not evaluate the expansions.

Normally, the argument expressions are not evaluated as part of computing the macro expansion, but instead appear as part of the expansion, so they are computed when the expansion is computed.

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 177, for a complete description of XEmacs Lisp macros.

8.2.7 Special Forms

A *special form* is a primitive function specially marked so that its arguments are not all evaluated. Most special forms define control structures or perform variable bindings—things which functions cannot do.

Each special form has its own rules for which arguments are evaluated and which are used without evaluation. Whether a particular argument is evaluated may depend on the results of evaluating other arguments.

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

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

`catch` see Section 9.5.1 [Catch and Throw], page 134

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

`condition-case`
see Section 9.5.3.3 [Handling Errors], page 138

`defconst` see Section 10.5 [Defining Variables], page 149

`defmacro` see Section 12.4 [Defining Macros], page 179

`defun` see Section 11.4 [Defining Functions], page 168

`defvar` see Section 10.5 [Defining Variables], page 149

`function` see Section 11.7 [Anonymous Functions], page 172

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

`interactive`
see Section 19.3 [Interactive Call], page 286

`let`
`let*` see Section 10.3 [Local Variables], page 146

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

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

`quote` see Section 8.3 [Quoting], page 126

`save-current-buffer`
see Section 34.3 [Excursions], page 495

`save-excursion`
see Section 34.3 [Excursions], page 495

`save-restriction`
see Section 34.4 [Narrowing], page 496

`save-selected-window`
see Section 34.3 [Excursions], page 495

`save-window-excursion`
see Section 31.16 [Window Configurations], page 468

`setq` see Section 10.7 [Setting Variables], page 152

`setq-default`
see Section 10.9.2 [Creating Buffer-Local], page 157

`unwind-protect`
see Section 9.5 [Nonlocal Exits], page 134

`while` see Section 9.4 [Iteration], page 133

`with-output-to-temp-buffer`
see Section 45.8 [Temporary Displays], page 656

> **Common Lisp note:** here are some comparisons of special forms in XEmacs Lisp and Common Lisp. `setq`, `if`, and `catch` are special forms in both XEmacs Lisp and Common Lisp. `defun` is a special form in XEmacs Lisp, but a macro in Common Lisp. `save-excursion` is a special form in XEmacs 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 XEmacs 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 XEmacs. It specifies which file contains the definition. When an autoload object appears as a symbol's function definition, calling that symbol as a function automatically loads the specified file; then it calls the real definition loaded from that file. See Section 14.2 [Autoload], page 198.

8.3 Quoting

The special form `quote` returns its single argument, as written, without evaluating it. This provides a way to include constant symbols and lists, which are not self-evaluating objects, in a program. (It is not necessary to quote self-evaluating objects such as numbers, strings, and vectors.)

quote *object* Special Form

This special form returns *object*, without evaluating it.

Because `quote` is used so often in programs, Lisp provides a convenient read syntax for it. An apostrophe character (‘'’) followed by a Lisp object (in read syntax) expands to a list whose first element is `quote`, and whose second element is the object. Thus, the read syntax `'x` is an abbreviation for `(quote x)`.

Here are some examples of expressions that use `quote`:

```
(quote (+ 1 2))
     ⇒ (+ 1 2)
(quote foo)
     ⇒ foo
'foo
     ⇒ foo
''foo
     ⇒ (quote foo)
'(quote foo)
     ⇒ (quote foo)
['foo]
     ⇒ [(quote foo)]
```

Other quoting constructs include `function` (see Section 11.7 [Anonymous Functions], page 172), which causes an anonymous lambda expression written in Lisp to be compiled, and ‘`’ (see Section 12.5 [Backquote], page 179), 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 121). We control the order of execution of the forms by enclosing them in *control structures*. Control structures are special forms which control when, whether, or how many times to execute the forms they contain.

The simplest order of execution is sequential execution: first form *a*, then form *b*, and so on. This is what happens when you write several forms in succession in the body of a function, or at top level in a file of Lisp code—the forms are executed in the order written. We call this *textual order*. For example, if a function body consists of two forms *a* and *b*, evaluation of the function evaluates first *a* and then *b*, and the function's value is the value of *b*.

Explicit control structures make possible an order of execution other than sequential.

XEmacs Lisp provides several kinds of control structure, including other varieties of sequencing, conditionals, iteration, and (controlled) jumps—all discussed below. The built-in control structures are special forms since their subforms are not necessarily evaluated or not evaluated sequentially. You can use macros to define your own control structure constructs (see Chapter 12 [Macros], page 177).

9.1 Sequencing

Evaluating forms in the order they appear is the most common way control passes from one form to another. In some contexts, such as in a function body, this happens automatically. Elsewhere you must use a control structure construct to do this: `progn`, the simplest control construct of Lisp.

A `progn` special form looks like this:

```
(progn a b c ...)
```

and it says to execute the forms *a*, *b*, *c* and so on, in that order. These forms are called the body of the `progn` form. The value of the last form in the body becomes the value of the entire `progn`.

In the early days of Lisp, `progn` was the only way to execute two or more forms in succession and use the value of the last of them. But programmers found they often needed to use a `progn` in the body of a function, where (at that time) only one form was allowed. So the body of a function was made into an "implicit `progn`": several forms are allowed just as in the body of an actual `progn`. Many other control structures likewise contain an implicit `progn`. As a result, `progn` is not used as often as it used to be. It is needed now most often inside an `unwind-protect`, `and`, `or`, or in the *then*-part of an `if`.

progn *forms...* Special Form

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

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

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

prog1 *form1 forms. . .* Special Form

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

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

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

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

prog2 *form1 form2 forms. . .* Special Form

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

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

9.2 Conditionals

Conditional control structures choose among alternatives. XEmacs 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 129.)

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

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

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

cond *clause*... Special Form

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

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

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

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

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

A clause may also look like this:

(*condition*)

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

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

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

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

For example,

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

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

Any conditional construct can be expressed with `cond` or with `if`. Therefore, the choice between them is a matter of style. For example:

```
(if a b c)
≡
(cond (a b) (t c))
```

9.3 Constructs for Combining Conditions

This section describes three constructs that are often used together with `if` and `cond` to express complicated conditions. The constructs `and` and `or` can also be used individually as kinds of multiple conditional constructs.

not *condition* Function

This function tests for the falsehood of *condition*. It returns `t` if *condition* is `nil`, and `nil` otherwise. The function `not` is identical to `null`, and we recommend using the name `null` if you are testing for an empty list.

and *conditions...* Special Form

The `and` special form tests whether all the *conditions* are true. It works by evaluating the *conditions* one by one in the order written.

If any of the *conditions* evaluates to `nil`, then the result of the `and` must be `nil` regardless of the remaining *conditions*; so `and` returns right away, ignoring the remaining *conditions*.

If all the *conditions* turn out non-`nil`, then the value of the last of them becomes the value of the `and` form.

Here is an example. The first condition returns the integer 1, which is not `nil`. Similarly, the second condition returns the integer 2, which is not `nil`. The third condition is `nil`, so the remaining condition is never evaluated.

```
(and (print 1) (print 2) nil (print 3))
     ⊣ 1
     ⊣ 2
⇒ nil
```

Here is a more realistic example of using `and`:

```
(if (and (consp foo) (eq (car foo) 'x))
    (message "foo is a list starting with x"))
```

Note that `(car foo)` is not executed if `(consp foo)` returns `nil`, thus avoiding an error.

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

```
(and arg1 arg2 arg3)
≡
(if arg1 (if arg2 arg3))
≡
(cond (arg1 (cond (arg2 arg3))))
```

or *conditions. . .* Special Form

The `or` special form tests whether at least one of the *conditions* is true. It works by evaluating all the *conditions* one by one in the order written.

If any of the *conditions* evaluates to a non-`nil` value, then the result of the `or` must be non-`nil`; so `or` returns right away, ignoring the remaining *conditions*. The value it returns is the non-`nil` value of the condition just evaluated.

If all the *conditions* turn out `nil`, then the `or` expression returns `nil`.

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

```
(or (eq x nil) (eq x 0))
```

Like the `and` construct, `or` can be written in terms of `cond`. For example:

```
(or arg1 arg2 arg3)
≡
(cond (arg1)
      (arg2)
      (arg3))
```

You could almost write `or` in terms of `if`, but not quite:

```
(if arg1 arg1
  (if arg2 arg2
    arg3))
```

This is not completely equivalent because it can evaluate *arg1* or *arg2* twice. By contrast, `(or` *arg1 arg2 arg3*`)` never evaluates any argument more than once.

9.4 Iteration

Iteration means executing part of a program repetitively. For example, you might want to repeat some computation once for each element of a list, or once for each integer from 0 to *n*. You can do this in XEmacs 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 134).

The value of a `while` form is always `nil`.

```
(setq num 0)
     ⇒ 0
(while (< num 4)
  (princ (format "Iteration %d." num))
  (setq num (1+ num)))
     ⊣ Iteration 0.
     ⊣ Iteration 1.
     ⊣ Iteration 2.
     ⊣ Iteration 3.
     ⇒ nil
```

If you would like to execute something on each iteration before the end-test, put it together with the end-test in a `progn` as the first argument of `while`, as shown here:

```
(while (progn
         (forward-line 1)
         (not (looking-at "^$"))))
```

This moves forward one line and continues moving by lines until it reaches an empty. It is unusual in that the `while` has no body, just the end test (which also does the real work of moving point).

9.5 Nonlocal Exits

A *nonlocal exit* is a transfer of control from one point in a program to another remote point. Nonlocal exits can occur in XEmacs Lisp as a result of errors; you can also use them under explicit control. Nonlocal exits unbind all variable bindings made by the constructs being exited.

9.5.1 Explicit Nonlocal Exits: `catch` and `throw`

Most control constructs affect only the flow of control within the construct itself. The function `throw` is the exception to this rule of normal program execution: it performs a nonlocal exit on request. (There are other exceptions, but they are for error handling only.) `throw` is used inside a `catch`, and jumps back to that `catch`. For example:

```
(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.

Executing `throw` exits all Lisp constructs up to the matching `catch`, including function calls. When binding constructs such as `let` or function calls are exited in this way, the bindings are unbound, just as they are when these constructs exit normally (see Section 10.3 [Local Variables], page 146). Likewise, `throw` restores the buffer and position saved by `save-excursion` (see Section 34.3 [Excursions], page 495), and the narrowing status saved by `save-restriction` and the window selection saved by `save-window-excursion` (see Section 31.16 [Window Configurations], page 468). It also runs any cleanups established with the `unwind-protect` special form when it exits that form (see Section 9.5.4 [Cleanups], page 142).

The `throw` need not appear lexically within the `catch` that it jumps to. It can equally well be called from another function called within the `catch`. As long as the `throw` takes place chronologically after entry to the `catch`, and chronologically before exit from it, it has access to that `catch`. This is why `throw` can be used in commands such as `exit-recursive-edit` that throw back to the editor command loop (see Section 19.10 [Recursive Editing], page 310).

> **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. XEmacs 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, `catch` evaluates the forms of the *body* in textual order. If the forms execute normally, without error or nonlocal exit, the value of the last body form is returned from the `catch`.

If a `throw` is done within *body* specifying the same value *tag*, the `catch` exits immediately; the value it returns is whatever was specified as the second argument of `throw`.

throw *tag value* Function

The purpose of `throw` is to return from a return point previously established with `catch`. The argument *tag* is used to choose among the various existing return points; it must be `eq` to the value specified in the `catch`. If multiple return points match *tag*, the innermost one is used.

The argument *value* is used as the value to return from that `catch`.

If no return point is in effect with tag *tag*, then a `no-catch` error is signaled with data `(`*tag value*`)`.

9.5.2 Examples of `catch` and `throw`

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

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

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

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

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

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

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

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

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

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

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

9.5.3 Errors

When XEmacs Lisp attempts to evaluate a form that, for some reason, cannot be evaluated, it *signals* an *error*.

When an error is signaled, XEmacs's default reaction is to print an error message and terminate execution of the current command. This is the right thing to do in most cases, such as if you type `C-f` at the end of the buffer.

In complicated programs, simple termination may not be what you want. For example, the program may have made temporary changes in data structures, or created temporary buffers that should be deleted before the program is finished. In such cases, you would use `unwind-protect` to establish *cleanup expressions* to be evaluated in case of error. (See Section 9.5.4 [Cleanups], page 142.) Occasionally, you may wish the program to continue execution despite an error in a subroutine. In these cases, you would use `condition-case` to establish *error handlers* to recover control in case of error.

Resist the temptation to use error handling to transfer control from one part of the program to another; use `catch` and `throw` instead. See Section 9.5.1 [Catch and Throw], page 134.

9.5.3.1 How to Signal an Error

Most errors are signaled "automatically" within Lisp primitives which you call for other purposes, such as if you try to take the CAR of an integer or move forward a character at the end of the buffer; you can also signal errors explicitly with the functions `error` and `signal`.

Quitting, which happens when the user types *C-g*, is not considered an error, but it is handled almost like an error. See Section 19.8 [Quitting], page 306.

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

This function signals an error with an error message constructed by applying `format` (see Section 4.7 [String Conversion], page 65) to *format-string* and *args*.

These examples show typical uses of `error`:

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

signal *error-symbol data* Function

This function signals an error named by *error-symbol*. The argument *data* is a list of additional Lisp objects relevant to the circumstances of the error.

The argument *error-symbol* must be an *error symbol*—a symbol bearing a property `error-conditions` whose value is a list of condition names. This is how XEmacs Lisp classifies different sorts of errors.

The number and significance of the objects in *data* depends on *error-symbol*. For example, with a `wrong-type-arg` error, there are two objects in the list: a predicate that describes the type that was expected, and the object that failed to fit that type. See Section 9.5.3.4 [Error Symbols], page 140, for a description of error symbols.

Both *error-symbol* and *data* are available to any error handlers that handle the error: `condition-case` binds a local variable to a list of the form `(`*error-symbol* `.` *data*`)` (see Section 9.5.3.3 [Handling Errors], page 138). If the error is not handled, these two values are used in printing the error message.

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

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

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

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

9.5.3.2 How XEmacs Processes Errors

When an error is signaled, `signal` searches for an active *handler* for the error. A handler is a sequence of Lisp expressions designated to be executed if an error happens in part of the Lisp program. If the error has an applicable handler, the handler is executed, and control resumes following the handler. The handler executes in the environment of the `condition-case` that established it; all functions called within that `condition-case` have already been exited, and the handler cannot return to them.

If there is no applicable handler for the error, the current command is terminated and control returns to the editor command loop, because the command loop has an implicit handler for all kinds of errors. The command loop's handler uses the error symbol and associated data to print an error message.

An error that has no explicit handler may call the Lisp debugger. The debugger is enabled if the variable `debug-on-error` (see Section 16.1.1 [Error Debugging], page 217) 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 XEmacs editor command loop. You can arrange to trap errors occurring in a part of your program by establishing an error handler, with the special form `condition-case`. A simple example looks like this:

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

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

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

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

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

When an error is handled, control returns to the handler. Before this happens, XEmacs 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 that can be handled).

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

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

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

```
(error nil)

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

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

Each error that occurs has an *error symbol* that describes what kind of error it is. The `error-conditions` property of this symbol is a list of condition names (see Section 9.5.3.4 [Error Symbols], page 140). Emacs searches all the active `condition-case` forms for a handler that specifies one or more of these condition names; the innermost matching `condition-case` handles the error. Within this `condition-case`, the first applicable handler handles the error.

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

The argument *var* is a variable. `condition-case` does not bind this variable when executing the *protected-form*, only when it handles an error. At that time, it binds *var* locally to 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 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 XEmacs Lisp language.

These narrow classifications are grouped into a hierarchy of wider classes called *error conditions*, identified by *condition names*. The narrowest such classes belong to the error symbols themselves: each error symbol is also a condition name. There are also condition names for more extensive classes, up to the condition name `error` which takes in all kinds of errors. Thus, each error has one or more condition names: `error`, the error symbol if that is distinct from `error`, and perhaps some intermediate classifications.

In order for a symbol to be an error symbol, it must have an `error-conditions` property which gives a list of condition names. This list defines the conditions that this kind of error belongs to. (The error symbol itself, and the symbol `error`, should always be members of this list.) Thus, the hierarchy of condition names is defined by the `error-conditions` properties of the error symbols.

In addition to the `error-conditions` list, the error symbol should have an `error-message` property whose value is a string to be printed when that error is signaled but not handled. If the `error-message` property exists, but is not a string, the error message '`peculiar error`' is used.

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

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

This error has three condition names: `new-error`, the narrowest classification; `my-own-errors`, which we imagine is a wider classification; and `error`, which is the widest of all.

The error string should start with a capital letter but it should not end with a period. This is for consistency with the rest of Emacs.

Naturally, XEmacs will never signal `new-error` on its own; only an explicit call to `signal` (see Section 9.5.3.1 [Signaling Errors], page 137) in your code can do this:

```
(signal 'new-error '(x y))
     error  A new error: x, y
```

This error can be handled through any of the three condition names. This example handles `new-error` and any other errors in the class `my-own-errors`:

```
(condition-case foo
    (bar nil t)
  (my-own-errors nil))
```

The significant way that errors are classified is by their condition names—the names used to match errors with handlers. An error symbol serves only as a convenient way to specify the intended error message and list of condition names. It would be cumbersome to give `signal` a list of condition names rather than one error symbol.

By contrast, using only error symbols without condition names would seriously decrease the power of `condition-case`. Condition names make it possible to categorize errors at various levels of generality when you write an error handler. Using error symbols alone would eliminate all but the narrowest level of classification.

See Appendix C [Standard Errors], page 773, 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.

If the *body* forms finish normally, `unwind-protect` returns the value of the last *body* form, after it evaluates the *cleanup-forms*. If the *body* forms do not finish, `unwind-protect` does not return any value in the normal sense.

Only the *body* is actually protected by the `unwind-protect`. If any of the *cleanup-forms* themselves exits nonlocally (e.g., via a `throw` or an error), `unwind-protect` is *not* guaranteed to evaluate the rest of them. If the failure of one of the *cleanup-forms* has the potential to cause trouble, then protect it with another `unwind-protect` around that form.

The number of currently active `unwind-protect` forms counts, together with the number of local variable bindings, against the limit `max-specpdl-size` (see Section 10.3 [Local Variables], page 146).

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 49 [Processes], page 673) to try to establish a connection to a remote machine. As the function `ftp-login` is highly susceptible to numerous problems that the writer of the function cannot anticipate, it is protected with a form that guarantees deletion of the process in the event of failure. Otherwise, XEmacs 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.

Here is another example which uses `unwind-protect` to make sure to kill a temporary buffer. In this example, the value returned by `unwind-protect` is used.

```
(defun shell-command-string (cmd)
  "Return the output of the shell command CMD, as a string."
  (save-excursion
    (set-buffer (generate-new-buffer " OS*cmd"))
    (shell-command cmd t)
    (unwind-protect
        (buffer-string)
      (kill-buffer (current-buffer)))))
```

10 Variables

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

In Lisp, unlike most programming languages, programs are represented primarily as Lisp objects and only secondarily as text. The Lisp objects used for variables are symbols: the symbol name is the variable name, and the variable's value is stored in the value cell of the symbol. The use of a symbol as a variable is independent of its use as a function name. See Section 7.1 [Symbol Components], page 111.

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

10.1 Global Variables

The simplest way to use a variable is *globally*. This means that the variable has just one value at a time, and this value is in effect (at least for the moment) throughout the Lisp system. The value remains in effect until you specify a new one. When a new value replaces the old one, no trace of the old value remains in the variable.

You specify a value for a symbol with `setq`. For example,

```
(setq x '(a b))
```

gives the variable `x` the value `(a b)`. Note that `setq` does not evaluate its first argument, the name of the variable, but it does evaluate the second argument, the new value.

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

```
x ⇒ (a b)
```

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

If you do another `setq`, the new value replaces the old one:

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

10.2 Variables That Never Change

In XEmacs Lisp, some symbols always evaluate to themselves: the two special symbols `nil` and `t`, as well as *keyword symbols*, that is, symbols whose name begins with the character '`:`'. 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 have 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 that last until the function exits. The `let` special form explicitly establishes new local values for specified variables; these last until exit from the `let` form.

Establishing a local value saves away the previous value (or lack of one) of the variable. When the life span of the local value is over, the previous value is restored. In the mean time, we say that the previous value is *shadowed* and *not visible*. Both global and local values may be shadowed (see Section 10.8.1 [Scope], page 154).

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

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

All of the *value-forms* in *bindings* are evaluated in the order they appear and *before* 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`, but it binds each variable right after computing its local value, before computing the local value for the next variable. Therefore, an expression in *bindings* can reasonably refer to the preceding symbols bound in this `let*` form. Compare the following example with the example above for `let`.

```
(setq Y 2)
     ⇒ 2
(let* ((Y 1)
       (Z Y))     ; Use the just-established value of Y.
  (list Y Z))
     ⇒ (1 1)
```

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

- Function calls (see Chapter 11 [Functions], page 163).
- Macro calls (see Chapter 12 [Macros], page 177).
- `condition-case` (see Section 9.5.3 [Errors], page 136).

Variables can also have buffer-local bindings (see Section 10.9 [Buffer-Local Variables], page 157). These kinds of bindings work somewhat like ordinary local bindings, but they are localized depending on "where" you are in Emacs, rather than localized in time.

max-specpdl-size Variable

This variable defines the limit on the total number of local variable bindings and `unwind-protect` cleanups (see Section 9.5 [Nonlocal Exits], page 134) 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 120.

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. Subsequent attempts to use this symbol's value as a variable will 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 *variable definition*: a special form, either `defconst` or `defvar`.

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

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

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 user options in their init files, and override the default values given in the definitions. For this reason, user options must be defined with `defvar`.

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

This special form defines *symbol* as a value and initializes it. The definition informs a person reading your code that *symbol* is used as a variable that programs are likely to set or change. It is also used for all user option variables except in the preloaded parts of XEmacs. 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, `defvar` evaluates it and sets *symbol* to the result. (If *value* is omitted, the value of *symbol* is not changed in any case.)

When you evaluate a top-level `defvar` form with `C-M-x` in Emacs Lisp mode (`eval-defun`), a special feature of `eval-defun` evaluates it as a `defconst`. The purpose of this is to make sure the variable's value is reinitialized, when you ask for it specifically.

If *symbol* has a buffer-local binding in the current buffer, `defvar` sets the default value, not the local value. See Section 10.9 [Buffer-Local Variables], page 157.

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 XEmacs help functions (see Chapter 27 [Documentation], page 381) look for this property.

If the first character of *doc-string* is '`*`', it means that this variable is considered a user option. This lets users set the variable conveniently using the commands `set-variable` and `edit-options`.

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 defines *symbol* as a value and initializes it. It 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 standardly preloaded. 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 3.1415 "Pi to five places.")
     ⇒ pi
(setq pi 3)
     ⇒ pi
pi
     ⇒ 3
```

user-variable-p *variable* Function

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

User option variables are distinguished from other variables by the first character of the `variable-documentation` property. If the property exists and is a string, and its first character is '`*`', then the variable is a user option.

If a user option variable has a `variable-interactive` property, the `set-variable` command uses that value to control reading the new value for the variable. The property's value is used as if it were the argument to `interactive`.

Warning: If the `defconst` and `defvar` special forms are used while the variable has a local binding, they set the local binding's value; the global binding is not changed. This is not what we really want. To prevent it, use these special forms at top level in a file, where normally no local binding is in effect, and make sure to load the file before making a local binding for the variable.

10.6 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 122). 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.

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

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

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

```
x                           ; The global value is unchanged.
     ⇒ 3
```

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

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

set *symbol value* Function

This function sets *symbol*'s value to *value*, then returns *value*. Since `set` is a function, the expression written for *symbol* is evaluated to obtain the symbol to set.

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

```
(set one 1)
error  Symbol's value as variable is void: one
(set 'one 1)
     ⇒ 1
(set 'two 'one)
     ⇒ one
(set two 2)                 ; two evaluates to symbol one.
     ⇒ 2
one                         ; So it is one that was set.
     ⇒ 2
(let ((one 1))              ; This binding of one is set,
  (set 'one 3)              ;     not the global value.
  one)
     ⇒ 3
one
     ⇒ 2
```

If *symbol* is not actually a symbol, a `wrong-type-argument` error is signaled.

```
(set '(x y) 'z)
error  Wrong type argument: symbolp, (x y)
```

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

> **Common Lisp note:** In Common Lisp, `set` always changes the symbol's special value, ignoring any lexical bindings. In XEmacs Lisp, all variables and all bindings are (in effect) special, so `set` always affects the most local existing binding.

One other function for setting a variable is designed to add an element to a list if it is not already present in the list.

add-to-list *symbol element* Function

This function sets the variable *symbol* by consing *element* onto the old value, if *element* is not already a member of that value. It returns the resulting list, whether updated or not. The value of *symbol* had better be a list already before the call.

The argument *symbol* is not implicitly quoted; `add-to-list` is an ordinary function, like `set` and unlike `setq`. Quote the argument yourself if that is what you want.

Here's a scenario showing how to use `add-to-list`:

```
(setq foo '(a b))
     ⇒ (a b)

(add-to-list 'foo 'c)     ;; Add c.
     ⇒ (c a b)

(add-to-list 'foo 'b)     ;; No effect.
     ⇒ (c a b)

foo                       ;; foo was changed.
     ⇒ (c a b)
```

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

```
(or (member value var)
    (setq var (cons value var)))
```

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

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

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

10.8.1 Scope

XEmacs 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` in `binder` would never be accessible in `user`, because `user` is not textually contained within the function `binder`. However, in dynamically scoped XEmacs 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 XEmacs 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 XEmacs 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 XEmacs Lisp, because after the call to `make-add` exits, the variable `n` is no longer bound to the actual argument 2.

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

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

10.8.3 Implementation of Dynamic Scoping

A simple sample implementation (which is not how XEmacs 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 XEmacs Lisp uses a technique called *shallow binding*. Each variable has a standard place in which its current value is always found—the value cell of the symbol.

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

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

10.8.4 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.

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

10.9 Buffer-Local Variables

Global and local variable bindings are found in most programming languages in one form or another. XEmacs also supports another, unusual kind of variable binding: *buffer-local* bindings, which apply only to one buffer. XEmacs 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.

Local variables in a file you edit are also represented by buffer-local bindings for the buffer that holds the file within XEmacs. See Section 26.1.3 [Auto Major Mode], page 367.

10.9.2 Creating and Deleting Buffer-Local Bindings

make-local-variable *variable* Command

> This function creates a buffer-local binding in the current buffer for *variable* (a symbol). Other buffers are not affected. The value returned is *variable*.
>
> The buffer-local value of *variable* starts out as the same value *variable* previously had. If *variable* was void, it remains void.

```
;; In buffer ‘b1’:
(setq foo 5)                 ; Affects all buffers.
     ⇒ 5
(make-local-variable 'foo)   ; Now it is local in ‘b1’.
     ⇒ foo
foo                          ; That did not change
     ⇒ 5                     ;   the value.
(setq foo 6)                 ; Change the value
     ⇒ 6                     ;   in ‘b1’.
foo
     ⇒ 6

;; In buffer ‘b2’, the value hasn't changed.
(save-excursion
  (set-buffer "b2")
  foo)
     ⇒ 5
```

Making a variable buffer-local within a `let`-binding for that variable does not work. This is because `let` does not distinguish between different kinds of bindings; it knows only which variable the binding was made for.

Please note: do not use `make-local-variable` for a hook variable. Instead, use `make-local-hook`. See Section 26.4 [Hooks], page 378.

make-variable-buffer-local *variable* — Command

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

The value returned is *variable*.

local-variable-p *variable* &optional *buffer* — Function

This returns `t` if *variable* is buffer-local in buffer *buffer* (which defaults to the current buffer); otherwise, `nil`.

buffer-local-variables &optional *buffer* — Function

This function returns a list describing the buffer-local variables in buffer *buffer*. It returns an association list (see Section 5.8 [Association Lists], page 92) 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.

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

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

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

kill-all-local-variables — Function

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

This function also resets certain other information pertaining to the buffer: it sets the local keymap to `nil`, the syntax table to the value of `standard-syntax-table`, and the abbrev table 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` access and change a variable's default value regardless of whether the current buffer has a buffer-local binding. For example, you could use `setq-default` to change the default setting of `paragraph-start` for most buffers;

and this would work even when you are in a C or Lisp mode buffer that has a buffer-local value for this variable.

The special forms `defvar` and `defconst` also set the default value (if they set the variable at all), rather than any 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 151).

default-boundp *symbol* Function

The function `default-boundp` tells you whether *symbol*'s default value is non-void. 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*. It does not evaluate *symbol*, but does evaluate *value*. 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, `setq-default` has the same effect as `setq`. If *symbol* is buffer-local for the current buffer, then this changes the value that other buffers will see (as long as they don't have a buffer-local value), but not the value that the current buffer sees.

```
;; In buffer ‘foo’:
(make-local-variable '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
```

10.10 Variable Aliases

You can define a variable as an *alias* for another. Any time you reference the former variable, the current value of the latter is returned. Any time you change the value of the former variable, the value of the latter is actually changed. This is useful in cases where you want to rename a variable but still make old code work (see Section 27.6 [Obsoleteness], page 389).

defvaralias *variable alias* Function

This function defines *variable* as an alias for *alias*. Thenceforth, any operations performed on *variable* will actually be performed on *alias*. Both *variable* and *alias* should be symbols. If *alias* is `nil`, remove any aliases for *variable*. *alias* can itself be aliased, and the chain of variable aliases will be followed appropriately. If *variable* already has a value, this value will be shadowed until the alias is removed, at which point it will be restored. Currently *variable* cannot be a built-in variable, a variable that has a buffer-local value in any buffer, or the symbols `nil` or `t`.

variable-alias *variable* Function

If *variable* is aliased to another variable, this function returns that variable. *variable* should be a symbol. If *variable* is not aliased, this function returns `nil`.

indirect-variable *object* Function

This function returns the variable at the end of *object*'s variable-alias chain. If *object* is a symbol, follow all variable aliases and return the final (non-aliased) symbol. If *object* is not a symbol, just return it. Signal a `cyclic-variable-indirection` error if there is a loop in the variable chain of symbols.

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 XEmacs Lisp and for other function-like objects.

function
: In XEmacs 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, because it provides a low-level interface to operating system services, or because it needs to run fast. Primitives can be modified or added only by changing the C sources and recompiling the editor. See section "Writing Lisp Primitives" in *XEmacs Internals Manual*.

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

special form
: A *special form* is a primitive that is like a function but does not evaluate all of its arguments in the usual way. It may evaluate only some of the arguments, or may evaluate them in an unusual order, or several times. Many special forms are described in Chapter 9 [Control Structures], page 129.

macro
: A *macro* is a construct defined in Lisp by the programmer. It differs from a function in that it translates a Lisp expression that you write into an equivalent expression to be evaluated instead of the original expression. Macros enable Lisp programmers to do the sorts of things that special forms can do. See Chapter 12 [Macros], page 177, 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 19.2 [Defining Commands], page 282). Such a function can be called from Lisp expressions like other functions; in this case, the fact that the function is a command makes no difference.

Keyboard macros (strings and vectors) are commands also, even though they are not functions. A symbol is a command if its function definition is a command; such symbols can be invoked with `M-x`. The symbol is a function as well if the definition is a function. See Section 19.1 [Command Overview], page 281.

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

compiled function
A *compiled function* is a function that has been compiled by the byte compiler. See Section 2.4.14 [Compiled-Function Type], page 28.

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
```

compiled-function-p *object* — Function

This function returns `t` if *object* is a compiled function. For example:

```
(compiled-function-p (symbol-function 'next-line))
     ⇒ t
```

11.2 Lambda Expressions

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

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

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

11.2.1 Components of a Lambda Expression

The first element of a lambda expression is always the symbol `lambda`. This indicates that the list represents a function. The reason functions are defined to start with `lambda` is so that other lists, intended for other uses, will not accidentally be valid as functions.

The second element is a list of symbols–the argument variable names. This is called the *lambda list*. When a Lisp function is called, the argument values are matched up against the variables in the lambda list, which are given local bindings with the values provided. See Section 10.3 [Local Variables], page 146.

The documentation string is a Lisp string object placed within the function definition to describe the function for the XEmacs help facilities. See Section 11.2.4 [Function Documentation], page 167.

The interactive declaration is a list of the form `(interactive code-string)`. This declares how to provide arguments if the function is used interactively. Functions with this declaration are called *commands*; they can be called using `M-x` or bound to a key. Functions not intended to be called in this way should not have interactive declarations. See Section 19.2 [Defining Commands], page 282, for how to write an interactive declaration.

The rest of the elements are the *body* of the function: the Lisp code to do the work of the function (or, as a Lisp programmer would say, "a list of Lisp forms to evaluate"). The value returned by the function is the value returned by the last element of the body.

11.2.2 A Simple Lambda-Expression Example

Consider for example the following function:

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

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

```
((lambda (a b c) (+ a b c))
 1 2 3)
```

This call evaluates the body of the lambda expression with the variable `a` bound to 1, `b` bound to 2, and `c` bound to 3. Evaluation of the body adds these three numbers, producing the result 6; therefore, this call to the function returns the value 6.

Note that the arguments can be the results of other function calls, as in this example:

```
((lambda (a b c) (+ a b c))
 1 (* 2 3) (- 5 4))
```

This evaluates the arguments `1`, `(* 2 3)`, and `(- 5 4)` from left to right. Then it applies the lambda expression to the argument values 1, 6 and 1 to produce the value 8.

It is not often useful to write a lambda expression as the CAR of a form in this way. You can get the same result, of making local variables and giving them values, using the special form `let` (see Section 10.3 [Local Variables], page 146). 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 172).

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 *length* of the string if you omit it. It is also convenient for certain functions to accept an indefinite number of arguments, as the functions `list` and `+` do.

To specify optional arguments that may be omitted when a function is called, simply include the keyword `&optional` before the optional arguments. To specify a list of zero or more extra arguments, include the keyword `&rest` before one final argument.

Thus, the complete syntax for an argument list is as follows:

```
(required-vars...
 [&optional optional-vars...]
 [&rest rest-var])
```

The square brackets indicate that the `&optional` and `&rest` clauses, and the variables that follow them, are optional.

A call to the function requires one actual argument for each of the *required-vars*. There may be actual arguments for zero or more of the *optional-vars*, and there cannot be any actual arguments beyond that unless the lambda list uses `&rest`. In that case, there may be any number of extra actual arguments.

If actual arguments for the optional and rest variables are omitted, then they always default to `nil`. There is no way for the function to distinguish between an explicit argument of `nil` and an omitted argument. However, the body of the function is free to consider `nil` an abbreviation for some other meaningful value. This is what `substring` does; `nil` as the third argument to `substring` means to use the length of the string supplied.

> **Common Lisp note:** Common Lisp allows the function to specify what default value to use when an optional argument is omitted; XEmacs 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. Suppose three actual arguments are given; which variable would the third argument be for? Similarly, it makes no sense to have any more arguments (either required or optional) after a `&rest` argument.

Here are some examples of argument lists and proper calls:

```
((lambda (n) (1+ n))                ; One required:
 1)                                 ; requires exactly one argument.
     ⇒ 2
((lambda (n &optional n1)           ; One required and one optional:
         (if n1 (+ n n1) (1+ n)))   ; 1 or 2 arguments.
 1 2)
     ⇒ 3
((lambda (n &rest ns)               ; One required and one rest:
```

```
          (+ n (apply '+ ns)))         ; 1 or more arguments.
    1 2 3 4 5)
        ⇒ 15
```

11.2.4 Documentation Strings of Functions

A lambda expression may optionally have a *documentation string* just after the lambda list. This string does not affect execution of the function; it is a kind of comment, but a systematized comment which actually appears inside the Lisp world and can be used by the XEmacs help facilities. See Chapter 27 [Documentation], page 381, for how the *documentation-string* is accessed.

It is a good idea to provide documentation strings for all the functions in your program, even those that are only called from within your program. Documentation strings are like comments, except that they are easier to access.

The first line of the documentation string should stand on its own, because `apropos` displays just this first line. It should consist of one or two complete sentences that summarize the function's purpose.

The start of the documentation string is usually indented in the source file, but since these spaces come before the starting double-quote, they are not part of the string. Some people make a practice of indenting any additional lines of the string so that the text lines up in the program source. *This is a mistake.* The indentation of the following lines is inside the string; what looks nice in the source code will look ugly when displayed by the help commands.

You may wonder how the documentation string could be optional, since there are required components of the function that follow it (the body). Since evaluation of a string returns that string, without any side effects, it has no effect if it is not the last form in the body. Thus, in practice, there is no confusion between the first form of the body and the documentation string; if the only body form is a string then it serves both as the return value and as the documentation.

11.3 Naming a Function

In most computer languages, every function has a name; the idea of a function without a name is nonsensical. In Lisp, a function in the strictest sense has no name. It is simply a list whose first element is `lambda`, 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 111). Then the symbol itself becomes a valid, callable function, equivalent to the list or subr-object that its function cell refers to. The contents of the function cell are also called the symbol's *function definition*. The procedure of using a symbol's function definition in place of the symbol is called *symbol function indirection*; see Section 8.2.4 [Function Indirection], page 123.

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

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

We often identify functions with the symbols used to name them. For example, we often speak of "the function `car`", not distinguishing between the symbol `car` and the primitive subr-object that is its function definition. For most purposes, there is no need to distinguish.

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

A symbol used as a function name may also be used as a variable; these two uses of a symbol are independent and do not conflict.

11.4 Defining Functions

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

defun *name argument-list body-forms* Special Form

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

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

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

As described previously (see Section 11.2 [Lambda Expressions], page 164), *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.

There is no conflict if the same symbol *name* is also used as a variable, since the symbol's value cell is independent of the function cell. See Section 7.1 [Symbol Components], page 111.

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.

define-function *name definition* Function
defalias *name definition* Function

These equivalent special forms define the symbol *name* as a function, with definition *definition* (which can be any valid Lisp function).

The proper place to use `define-function` or `defalias` is where a specific function name is being defined—especially where that name appears explicitly in the source file being loaded. This is because `define-function` and `defalias` record which file defined the function, just like `defun`. (see Section 14.5 [Unloading], page 203).

By contrast, in programs that manipulate function definitions for other purposes, it is better to use `fset`, which does not keep such records.

See also `defsubst`, which defines a function like `defun` and tells the Lisp compiler to open-code it. See Section 11.9 [Inline Functions], page 175.

11.5 Calling Functions

Defining functions is only half the battle. Functions don't do anything until you *call* them, i.e., tell them to run. Calling a function is also known as *invocation*.

The most common way of invoking a function is by evaluating a list. For example, evaluating the list `(concat "a" "b")` calls the function `concat` with arguments `"a"` and `"b"`. See Chapter 8 [Evaluation], page 119, for a description of evaluation.

When you write a list as an expression in your program, the function name is part of the program. This means that you choose which function to call, and how many arguments to give it, when you write the program. Usually that's just what you want. Occasionally you need to decide at run time which function to call. To do that, 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 these example with the examples of `apply`.

apply *function* &rest *arguments* Function

`apply` calls *function* with *arguments*, just like `funcall` but with one difference: the last of *arguments* is a list of arguments to give to *function*, rather than a single argument. We also say that `apply` *spreads* this list so that each individual element becomes an argument.

`apply` returns the result of calling *function*. As with `funcall`, *function* must either be a Lisp function or a primitive function; special forms and macros do not make sense in `apply`.

```
(setq f 'list)
     ⇒ list
(apply f 'x 'y 'z)
error  Wrong type argument: listp, z
(apply '+ 1 2 '(3 4))
     ⇒ 10
(apply '+ '(1 2 3 4))
     ⇒ 10

(apply 'append '((a b c) nil (x y z) nil))
     ⇒ (a b c x y z)
```

For an interesting example of using `apply`, see the description of `mapcar`, in Section 11.6 [Mapping Functions], page 171.

It is common for Lisp functions to accept functions as arguments or find them in data structures (especially in hook variables and property lists) and call them using `funcall` or `apply`. Functions that accept function arguments are often called *functionals*.

Sometimes, when you call a functional, it is useful to supply a no-op function as the argument. Here are two different kinds of no-op function:

identity *arg* Function

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

ignore &rest *args* Function

This function ignores any arguments and returns `nil`.

11.6 Mapping Functions

A *mapping function* applies a given function to each element of a list or other collection. XEmacs 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 113.

mapcar *function sequence* Function

`mapcar` applies *function* to each element of *sequence* in turn, and returns a list of the results.

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.
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 return a string.

```
(mapconcat 'symbol-name
           '(The cat in the hat)
           " ")
     ⇒ "The cat in the hat"

(mapconcat (function (lambda (x) (format "%c" (1+ x))))
           "HAL-8000"
           "")
     ⇒ "IBM.9111"
```

11.7 Anonymous Functions

In Lisp, a function is a list that starts with `lambda`, a byte-code function compiled from such a list, or alternatively a primitive subr-object; names are "extra". Although usually functions are defined with `defun` and given names at the same time, it is occasionally more concise to use an explicit lambda expression—an anonymous function. Such a list is valid wherever a function name is.

Any method of creating such a list makes a valid function. Even this:

```
(setq silly (append '(lambda (x)) (list (list '+ (* 3 4) 'x))))
⇒ (lambda (x) (+ 12 x))
```

This computes a list that looks like `(lambda (x) (+ 12 x))` and makes it the value (*not* the function definition!) of `silly`.

Here is how we might call this function:

```
(funcall silly 1)
⇒ 13
```

(It does *not* work to write `(silly 1)`, because this function is not the *function definition* of `silly`. We have not given `silly` any function definition, just a value as a variable.)

Most of the time, anonymous functions are constants that appear in your program. For example, you might want to pass one as an argument to the function `mapcar`, which applies any given function to each element of a list. Here we 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 XEmacs Lisp compiler that *function-object* is intended to be used only as a function, and therefore can safely be compiled. Contrast this with `quote`, in Section 8.3 [Quoting], page 126.

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 (x) (* x 2))
```

The Lisp compiler cannot assume this list is a function, even though it looks like one, since it does not know what `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 27.2 [Accessing Documentation], page 382, for a realistic example using `function` and an anonymous function.

11.8 Accessing Function Cell Contents

The *function definition* of a symbol is the object stored in the function cell of the symbol. The functions described here access, test, and set the function cell of symbols.

See also the function `indirect-function` in Section 8.2.4 [Function Indirection], page 123.

symbol-function *symbol* Function

This returns the object in the function cell of *symbol*. If the symbol's function cell is void, a `void-function` error is signaled.

This function does not check that the returned object is a legitimate function.

```
(defun bar (n) (+ n 2))
     ⇒ bar
(symbol-function 'bar)
     ⇒ (lambda (n) (+ n 2))
(fset 'baz 'bar)
     ⇒ bar
(symbol-function 'baz)
     ⇒ bar
```

If you have never given a symbol any function definition, we say that that symbol's function cell is *void*. In other words, the function cell does not have any Lisp object in it. If you try to call such a symbol as a function, it signals a `void-function` error.

Note that void is not the same as `nil` or the symbol `void`. The symbols `nil` and `void` are Lisp objects, and can be stored into a function cell just as any other object can be (and they can be valid functions if you define them in turn with `defun`). A void function cell contains no object whatsoever.

You can test the voidness of a symbol's function definition with `fboundp`. After you have given a symbol a function definition, you can make it void once more using `fmakunbound`.

fboundp *symbol* Function

This function returns `t` if the symbol has an object in its function cell, `nil` otherwise. It does not check that the object is a legitimate function.

fmakunbound *symbol* Function

This function makes *symbol*'s function cell void, so that a subsequent attempt to access this cell will cause a `void-function` error. (See also `makunbound`, in Section 10.3 [Local Variables], page 146.)

```
(defun foo (x) x)
     ⇒ x
(foo 1)
     ⇒1
(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`. For example, you can use `fset` to give a symbol `s1` a function definition which is another symbol `s2`; then `s1` serves as an alias for whatever definition `s2` presently has.
- 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"
```

See also the related functions `define-function` and `defalias`, in Section 11.4 [Defining Functions], page 168.

When writing a function that extends a previously defined function, the following idiom is sometimes used:

```
(fset 'old-foo (symbol-function 'foo))
(defun foo ()
  "Just like old-foo, except more so."
  (old-foo)
  (more-so))
```

This does not work properly if `foo` has been defined to autoload. In such a case, when `foo` calls `old-foo`, Lisp attempts to define `old-foo` by loading a file. Since this presumably defines `foo` rather than `old-foo`, it does not produce the proper results. The only way to avoid this problem is to make sure the file is loaded before moving aside the old definition of `foo`.

But it is unmodular and unclean, in any case, for a Lisp file to redefine a function defined elsewhere.

11.9 Inline Functions

You can define an *inline function* by using `defsubst` instead of `defun`. An inline function works just like an ordinary function except for one thing: when you compile a call to the function, the function's definition is open-coded into the caller.

Making a function inline makes explicit calls run faster. But it also has disadvantages. For one thing, it reduces flexibility; if you change the definition of the function, calls already inlined still use the old definition until you recompile them. Since the flexibility of redefining functions is an important feature of XEmacs, you should not make a function inline unless its speed is really crucial.

Another disadvantage is that making a large function inline can increase the size of compiled code both in files and in memory. Since the speed advantage of inline functions is greatest for small functions, you generally should not make large functions inline.

It's possible to define a macro to expand into the same code that an inline function would execute. 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 177.) To convert it into an inline function is very easy; simply replace `defun` with `defsubst`. Since each argument of an inline function is evaluated exactly once, you needn't worry about how many times the body uses the arguments, as you do for macros. (See Section 12.6.1 [Argument Evaluation], page 180.)

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

11.10 Other Topics Related to Functions

Here is a table of several functions that do things related to function calling and function definitions. They are documented elsewhere, but we provide cross references here.

`apply` See Section 11.5 [Calling Functions], page 169.

`autoload` See Section 14.2 [Autoload], page 198.

`call-interactively`
See Section 19.3 [Interactive Call], page 286.

`commandp` See Section 19.3 [Interactive Call], page 286.

`documentation`
See Section 27.2 [Accessing Documentation], page 382.

`eval` See Section 8.1 [Eval], page 120.

`funcall` See Section 11.5 [Calling Functions], page 169.

`ignore` See Section 11.5 [Calling Functions], page 169.

`indirect-function`
See Section 8.2.4 [Function Indirection], page 123.

`interactive`
See Section 19.2.1 [Using Interactive], page 282.

`interactive-p`
See Section 19.3 [Interactive Call], page 286.

`mapatoms` See Section 7.3 [Creating Symbols], page 113.

`mapcar` See Section 11.6 [Mapping Functions], page 171.

`mapconcat`
See Section 11.6 [Mapping Functions], page 171.

`undefined`
See Section 20.8 [Key Lookup], page 324.

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 175.

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. Byte compilation uses this feature.

```
(defmacro inc (var)
    (list 'setq var (list '1+ var)))
     ⇒ inc

(macroexpand '(inc r))
     ⇒ (setq r (1+ r))

(defmacro inc2 (var1 var2)
    (list 'progn (list 'inc var1) (list 'inc var2)))
     ⇒ inc2

(macroexpand '(inc2 r s))
     ⇒ (progn (inc r) (inc s))  ; inc not expanded here.
```

12.3 Macros and Byte Compilation

You might ask why we take the trouble to compute an expansion for a macro and then evaluate the expansion. Why not have the macro body produce the desired results directly? The reason has to do with compilation.

When a macro call appears in a Lisp program being compiled, the Lisp compiler calls the macro definition just as the interpreter would, and receives an expansion. But instead of evaluating this expansion, it compiles the expansion as if it had appeared directly in the program. As a result, the compiled code produces the value and side effects intended for the macro, but executes at full compiled speed. This would not work if the macro body computed the value and side effects itself—they would be computed at compile time, which is not useful.

In order for compilation of macro calls to work, the macros must 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.

Byte-compiling a file executes any `require` calls at top-level in the file. This is in case the file needs the required packages for proper compilation. One way to ensure that necessary macro definitions are available during compilation is to require the files that define them (see Section 14.4 [Named Features], page 201). To avoid loading the macro definition files when someone *runs* the compiled program, write `eval-when-compile` around the `require` calls (see Section 15.5 [Eval During Compile], page 209).

12.4 Defining Macros

A Lisp macro is a list whose CAR is `macro`. Its CDR should be a function; expansion of the macro works by applying the function (with `apply`) to the list of unevaluated argument-expressions from the macro call.

It is possible to use an anonymous Lisp macro just like an anonymous function, but this is never done, because it does not make sense to pass an anonymous macro to functionals such as `mapcar`. In practice, all Lisp macros have names, and they are usually defined with the special form `defmacro`.

defmacro *name argument-list body-forms...* Special Form

`defmacro` defines the symbol *name* as a macro that looks like this:

(macro lambda *argument-list* . *body-forms*)

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

12.5 Backquote

Macros often need to construct large list structures from a mixture of constants and nonconstant parts. To make this easier, use the macro '`' (often called *backquote*).

Backquote allows you to quote a list, but selectively evaluate elements of that list. In the simplest case, it is identical to the special form `quote` (see Section 8.3 [Quoting], page 126). For example, these two forms yield identical results:

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

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

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

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

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

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

Before Emacs version 19.29, '`' used a different syntax which required an extra level of parentheses around the entire backquote construct. Likewise, each ',' or ',@' substitution required an extra level of parentheses surrounding both the ',' or ',@' and the following expression. The old syntax required whitespace between the '`', ',' or ',@' and the following expression.

This syntax is still accepted, but no longer recommended except for compatibility with old Emacs versions.

12.6 Common Problems Using Macros

The basic facts of macro expansion have counterintuitive consequences. This section describes some important consequences that can lead to trouble, and rules to follow to avoid trouble.

12.6.1 Evaluating Macro Arguments Repeatedly

When defining a macro you must pay attention to the number of times the arguments will be evaluated when the expansion is executed. The following macro (used to facilitate iteration) illustrates the problem. This macro allows us to write a simple "for" loop such as one might find in Pascal.

```
(defmacro for (var from init to final do &rest body)
  "Execute a simple \"for\" loop.
For example, (for i from 1 to 10 do (print i))."
  (list 'let (list (list var init))
        (cons 'while (cons (list '<= var final)
                           (append body (list (list 'inc var)))))))
⇒ for

(for i from 1 to 3 do
   (setq square (* i i))
   (princ (format "\n%d %d" i square)))
↦
```

```
(let ((i 1))
  (while (<= i 3)
    (setq square (* i i))
    (princ (format "%d      %d" i square))
    (inc i)))

     ⊣ 1       1
     ⊣ 2       4
     ⊣ 3       9
⇒ nil
```

(The arguments `from`, `to`, and `do` in this macro are "syntactic sugar"; they are entirely ignored. The idea is that you will write noise words (such as `from`, `to`, and `do`) in those positions in the macro call.)

Here's an equivalent definition simplified through use of backquote:

```
(defmacro for (var from init to final do &rest body)
  "Execute a simple \"for\" loop.
For example, (for i from 1 to 10 do (print i))."
  `(let ((,var ,init))
     (while (<= ,var ,final)
       ,@body
       (inc ,var))))
```

Both forms of this definition (with backquote and without) suffer from the defect that *final* is evaluated on every iteration. If *final* is a constant, this is not a problem. If it is a more complex form, say `(long-complex-calculation x)`, this can slow down the execution significantly. If *final* has side effects, executing it more than once is probably incorrect.

A well-designed macro definition takes steps to avoid this problem by producing an expansion that evaluates the argument expressions exactly once unless repeated evaluation is part of the intended purpose of the macro. Here is a correct expansion for the `for` macro:

```
(let ((i 1)
      (max 3))
  (while (<= i max)
    (setq square (* i i))
    (princ (format "%d      %d" i square))
    (inc i)))
```

Here is a macro definition that creates this expansion:

```
(defmacro for (var from init to final do &rest body)
  "Execute a simple for loop: (for i from 1 to 10 do (print i))."
  `(let ((,var ,init)
         (max ,final))
     (while (<= ,var max)
       ,@body
       (inc ,var))))
```

Unfortunately, this 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 113). The uninterned symbol can be bound and referred to just like any other symbol, but since it is created by `for`, we know that it cannot already appear in the user's program. Since it is not interned, there is no way the user can put it into the program later. It will never appear anywhere except where put by `for`. Here is a definition of `for` that works this way:

```
(defmacro for (var from init to final do &rest body)
  "Execute a simple for loop: (for i from 1 to 10 do (print i))."
  (let ((tempvar (make-symbol "max")))
    `(let ((,var ,init)
           (,tempvar ,final))
       (while (<= ,var ,tempvar)
         ,@body
         (inc ,var)))))
```

This creates an uninterned symbol named `max` and puts it in the expansion instead of the usual interned symbol `max` that appears in expressions ordinarily.

12.6.3 Evaluating Macro Arguments in Expansion

Another problem can happen if 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 120). If the argument is supposed to refer to the user's variables, you may have trouble if the user happens to use a variable with the same name as one of the macro arguments. Inside the macro body, the macro argument binding is the most local binding of this variable, so any references inside the form being evaluated do refer to it. Here is an example:

```
(defmacro foo (a)
  (list 'setq (eval a) t))
     ⇒ foo
```

```
(setq x 'b)
(foo x) ↦ (setq b t)
     ⇒ t                    ; and b has been set.
;; but
(setq a 'c)
(foo a) ↦ (setq a t)
     ⇒ t                    ; but this set a, not c.
```

It makes a difference whether the user's variable is named `a` or `x`, because `a` conflicts with the macro argument variable `a`.

Another reason not to call `eval` in a macro definition is that it probably won't do what you intend in a compiled program. The byte-compiler runs macro definitions while compiling the program, when the program's own computations (which you might have wished to access with `eval`) don't occur and its local variable bindings don't exist.

The safe way to work with the run-time value of an expression is to put the expression into the macro expansion, so that its value is computed as part of executing the expansion.

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 empty-object ()
  (list 'quote (cons nil nil)))

(defun initialize (condition)
  (let ((object (empty-object)))
    (if condition
        (setcar object condition))
    object))
```

If `initialize` is interpreted, a new list `(nil)` is constructed each time `initialize` is called. Thus, no side effect survives between calls. If `initialize` is compiled, then the macro `empty-object` is expanded during compilation, producing a single "constant" `(nil)` that is reused and altered each time `initialize` is called.

One way to avoid pathological cases like this is to think of `empty-object` as a funny kind of constant, not as a memory allocation construct. You wouldn't use `setcar` on a constant such as `'(nil)`, so naturally you won't use it on `(empty-object)` either.

13 Writing Customization Definitions

This chapter describes how to declare user options for customization, and also customization groups for classifying them. We use the term *customization item* to include both kinds of customization definitions—as well as face definitions.

13.1 Common Keywords for All Kinds of Items

All kinds of customization declarations (for variables and groups, and for faces) accept keyword arguments for specifying various information. This section describes some keywords that apply to all kinds.

All of these keywords, except `:tag`, can be used more than once in a given item. Each use of the keyword has an independent effect. The keyword `:tag` is an exception because any given item can only display one name.

`:tag` *name*
: Use *name*, a string, instead of the item's name, to label the item in customization menus and buffers.

`:group` *group*
: Put this customization item in group *group*. When you use `:group` in a `defgroup`, it makes the new group a subgroup of *group*.

  If you use this keyword more than once, you can put a single item into more than one group. Displaying any of those groups will show this item. Be careful not to overdo this!

`:link` *link-data*
: Include an external link after the documentation string for this item. This is a sentence containing an active field which references some other documentation.

  There are three alternatives you can use for *link-data*:

  `(custom-manual` *info-node*`)`
  : Link to an Info node; *info-node* is a string which specifies the node name, as in `"(emacs)Top"`. The link appears as '`[manual]`' in the customization buffer.

  `(info-link` *info-node*`)`
  : Like `custom-manual` except that the link appears in the customization buffer with the Info node name.

  `(url-link` *url*`)`
  : Link to a web page; *url* is a string which specifies the URL. The link appears in the customization buffer as *url*.

  You can specify the text to use in the customization buffer by adding `:tag` *name* after the first element of the *link-data*; for example, `(info-link :tag "foo" "(emacs)Top")` makes a link to the Emacs manual which appears in the buffer as '`foo`'.

  An item can have more than one external link; however, most items have none at all.

`:load` *file* Load file *file* (a string) before displaying this customization item. Loading is done with `load-library`, and only if the file is not already loaded.

`:require` *feature*
Require feature *feature* (a symbol) when installing a value for this item (an option or a face) that was saved using the customization feature. This is done by calling `require`.

The most common reason to use `:require` is when a variable enables a feature such as a minor mode, and just setting the variable won't have any effect unless the code which implements the mode is loaded.

13.2 Defining Custom Groups

Each Emacs Lisp package should have one main customization group which contains all the options, faces and other groups in the package. If the package has a small number of options and faces, use just one group and put everything in it. When there are more than twelve or so options and faces, then you should structure them into subgroups, and put the subgroups under the package's main customization group. It is OK to put some of the options and faces in the package's main group alongside the subgroups.

The package's main or only group should be a member of one or more of the standard customization groups. (To display the full list of them, use *`M-x customize`*.) Choose one or more of them (but not too many), and add your group to each of them using the `:group` keyword.

The way to declare new customization groups is with `defgroup`.

defgroup *group members doc* [*keyword value*]... Macro

Declare *group* as a customization group containing *members*. Do not quote the symbol *group*. The argument *doc* specifies the documentation string for the group.

The argument *members* is a list specifying an initial set of customization items to be members of the group. However, most often *members* is `nil`, and you specify the group's members by using the `:group` keyword when defining those members.

If you want to specify group members through *members*, each element should have the form `(`*name widget*`)`. Here *name* is a symbol, and *widget* is a widget type for editing that symbol. Useful widgets are `custom-variable` for a variable, `custom-face` for a face, and `custom-group` for a group.

In addition to the common keywords (see Section 13.1 [Common Keywords], page 185), you can use this keyword in `defgroup`:

`:prefix` *prefix*
If the name of an item in the group starts with *prefix*, then the tag for that item is constructed (by default) by omitting *prefix*.

One group can have any number of prefixes.

13.3 Defining Customization Variables

Use `defcustom` to declare user-editable variables.

defcustom *option default doc* [*keyword value*]... Macro

Declare *option* as a customizable user option variable. Do not quote *option*. The argument *doc* specifies the documentation string for the variable.

If *option* is void, `defcustom` initializes it to *default*. *default* should be an expression to compute the value; be careful in writing it, because it can be evaluated on more than one occasion.

The following additional keywords are defined:

`:type` *type*
: Use *type* as the data type for this option. It specifies which values are legitimate, and how to display the value. See Section 13.4 [Customization Types], page 188, for more information.

`:options` *list*
: Specify *list* as the list of reasonable values for use in this option.

Currently this is meaningful only when the type is `hook`. In that case, the elements of *list* should be functions that are useful as elements of the hook value. The user is not restricted to using only these functions, but they are offered as convenient alternatives.

`:version` *version*
: This option specifies that the variable was first introduced, or its default value was changed, in Emacs version *version*. The value *version* must be a string. For example,

```
(defcustom foo-max 34
  "*Maximum number of foo's allowed."
  :type 'integer
  :group 'foo
  :version "20.3")
```

`:set` *setfunction*
: Specify *setfunction* as the way to change the value of this option. The function *setfunction* should take two arguments, a symbol and the new value, and should do whatever is necessary to update the value properly for this option (which may not mean simply setting the option as a Lisp variable). The default for *setfunction* is `set-default`.

`:get` *getfunction*
: Specify *getfunction* as the way to extract the value of this option. The function *getfunction* should take one argument, a symbol, and should return the "current value" for that symbol (which need not be the symbol's Lisp value). The default is `default-value`.

`:initialize` *function*
: *function* should be a function used to initialize the variable when the `defcustom` is evaluated. It should take two arguments, the

symbol and value. Here are some predefined functions meant for use in this way:

`custom-initialize-set`
: Use the variable's `:set` function to initialize the variable, but do not reinitialize it if it is already non-void. This is the default `:initialize` function.

`custom-initialize-default`
: Like `custom-initialize-set`, but use the function `set-default` to set the variable, instead of the variable's `:set` function. This is the usual choice for a variable whose `:set` function enables or disables a minor mode; with this choice, defining the variable will not call the minor mode function, but customizing the variable will do so.

`custom-initialize-reset`
: Always use the `:set` function to initialize the variable. If the variable is already non-void, reset it by calling the `:set` function using the current value (returned by the `:get` method).

`custom-initialize-changed`
: Use the `:set` function to initialize the variable, if it is already set or has been customized; otherwise, just use `set-default`.

The `:require` option is useful for an option that turns on the operation of a certain feature. Assuming that the package is coded to check the value of the option, you still need to arrange for the package to be loaded. You can do that with `:require`. See Section 13.1 [Common Keywords], page 185. Here is an example, from the library '`paren.el`':

```
(defcustom show-paren-mode nil
  "Toggle Show Paren mode...."
  :set (lambda (symbol value)
 (show-paren-mode (or value 0)))
  :initialize 'custom-initialize-default
  :type 'boolean
  :group 'paren-showing
  :require 'paren)
```

Internally, `defcustom` uses the symbol property `standard-value` to record the expression for the default value, and `saved-value` to record the value saved by the user with the customization buffer. The `saved-value` property is actually a list whose car is an expression which evaluates to the value.

13.4 Customization Types

When you define a user option with `defcustom`, you must specify its *customization type*. That is a Lisp object which describes (1) which values are legitimate and (2) how to display the value in the customization buffer for editing.

You specify the customization type in `defcustom` with the `:type` keyword. The argument of `:type` is evaluated; since types that vary at run time are rarely useful, normally you use a quoted constant. For example:

```
(defcustom diff-command "diff"
  "*The command to use to run diff."
  :type '(string)
  :group 'diff)
```

In general, a customization type is a list whose first element is a symbol, one of the customization type names defined in the following sections. After this symbol come a number of arguments, depending on the symbol. Between the type symbol and its arguments, you can optionally write keyword-value pairs (see Section 13.4.4 [Type Keywords], page 192).

Some of the type symbols do not use any arguments; those are called *simple types*. For a simple type, if you do not use any keyword-value pairs, you can omit the parentheses around the type symbol. For example just `string` as a customization type is equivalent to `(string)`.

13.4.1 Simple Types

This section describes all the simple customization types.

`sexp`
: The value may be any Lisp object that can be printed and read back. You can use `sexp` as a fall-back for any option, if you don't want to take the time to work out a more specific type to use.

`integer`
: The value must be an integer, and is represented textually in the customization buffer.

`number`
: The value must be a number, and is represented textually in the customization buffer.

`string`
: The value must be a string, and the customization buffer shows just the contents, with no delimiting '"' characters and no quoting with '\'.

`regexp`
: Like `string` except that the string must be a valid regular expression.

`character`
: The value must be a character code. A character code is actually an integer, but this type shows the value by inserting the character in the buffer, rather than by showing the number.

`file`
: The value must be a file name, and you can do completion with *M-*⟨TAB⟩.

`(file :must-match t)`
: The value must be a file name for an existing file, and you can do completion with *M-*⟨TAB⟩.

`directory`
: The value must be a directory name, and you can do completion with *M-*⟨TAB⟩.

`symbol`
: The value must be a symbol. It appears in the customization buffer as the name of the symbol.

`function` The value must be either a lambda expression or a function name. When it is a function name, you can do completion with *M-*⟨TAB⟩.

`variable` The value must be a variable name, and you can do completion with *M-*⟨TAB⟩.

`face` The value must be a symbol which is a face name.

`boolean` The value is boolean—either `nil` or `t`. Note that by using `choice` and `const` together (see the next section), you can specify that the value must be `nil` or `t`, but also specify the text to describe each value in a way that fits the specific meaning of the alternative.

13.4.2 Composite Types

When none of the simple types is appropriate, you can use composite types, which build new types from other types. Here are several ways of doing that:

`(restricted-sexp :match-alternatives` *criteria*`)`
: The value may be any Lisp object that satisfies one of *criteria*. *criteria* should be a list, and each elements should be one of these possibilities:

- A predicate—that is, a function of one argument that returns non-`nil` if the argument fits a certain type. This means that objects of that type are acceptable.
- A quoted constant—that is, `'`*object*. This means that *object* itself is an acceptable value.

For example,

```
(restricted-sexp :match-alternatives (integerp 't 'nil))
```

allows integers, `t` and `nil` as legitimate values.

The customization buffer shows all legitimate values using their read syntax, and the user edits them textually.

`(cons` *car-type cdr-type*`)`
: The value must be a cons cell, its CAR must fit *car-type*, and its CDR must fit *cdr-type*. For example, `(cons string symbol)` is a customization type which matches values such as `("foo" . foo)`.

In the customization buffer, the CAR and the CDR are displayed and edited separately, each according to the type that you specify for it.

`(list` *element-types...*`)`
: The value must be a list with exactly as many elements as the *element-types* you have specified; and each element must fit the corresponding *element-type*.

For example, `(list integer string function)` describes a list of three elements; the first element must be an integer, the second a string, and the third a function.

In the customization buffer, the each element is displayed and edited separately, according to the type specified for it.

`(vector element-types...)`
: Like `list` except that the value must be a vector instead of a list. The elements work the same as in `list`.

`(choice alternative-types...)`
: The value must fit at least one of *alternative-types*. For example, `(choice integer string)` allows either an integer or a string.

In the customization buffer, the user selects one of the alternatives using a menu, and can then edit the value in the usual way for that alternative.

Normally the strings in this menu are determined automatically from the choices; however, you can specify different strings for the menu by including the `:tag` keyword in the alternatives. For example, if an integer stands for a number of spaces, while a string is text to use verbatim, you might write the customization type this way,

```
(choice (integer :tag "Number of spaces")
        (string :tag "Literal text"))
```

so that the menu offers '`Number of spaces`' and '`Literal Text`'.

In any alternative for which `nil` is not a valid value, other than a `const`, you should specify a valid default for that alternative using the `:value` keyword. See Section 13.4.4 [Type Keywords], page 192.

`(const value)`
: The value must be *value*—nothing else is allowed.

The main use of `const` is inside of `choice`. For example, `(choice integer (const nil))` allows either an integer or `nil`.

`:tag` is often used with `const`, inside of `choice`. For example,

```
(choice (const :tag "Yes" t)
        (const :tag "No" nil)
        (const :tag "Ask" foo))
```

`(function-item function)`
: Like `const`, but used for values which are functions. This displays the documentation string as well as the function name. The documentation string is either the one you specify with `:doc`, or *function*'s own documentation string.

`(variable-item variable)`
: Like `const`, but used for values which are variable names. This displays the documentation string as well as the variable name. The documentation string is either the one you specify with `:doc`, or *variable*'s own documentation string.

`(set elements...)`
: The value must be a list and each element of the list must be one of the *elements* specified. This appears in the customization buffer as a checklist.

`(repeat element-type)`
: The value must be a list and each element of the list must fit the type *element-type*. This appears in the customization buffer as a list of elements, with '`[INS]`' and '`[DEL]`' buttons for adding more elements or removing elements.

13.4.3 Splicing into Lists

The `:inline` feature lets you splice a variable number of elements into the middle of a list or vector. You use it in a `set`, `choice` or `repeat` type which appears among the element-types of a `list` or `vector`.

Normally, each of the element-types in a `list` or `vector` describes one and only one element of the list or vector. Thus, if an element-type is a `repeat`, that specifies a list of unspecified length which appears as one element.

But when the element-type uses `:inline`, the value it matches is merged directly into the containing sequence. For example, if it matches a list with three elements, those become three elements of the overall sequence. This is analogous to using '`,@`' in the backquote construct.

For example, to specify a list whose first element must be `t` and whose remaining arguments should be zero or more of `foo` and `bar`, use this customization type:

```
(list (const t) (set :inline t foo bar))
```

This matches values such as `(t)`, `(t foo)`, `(t bar)` and `(t foo bar)`.

When the element-type is a `choice`, you use `:inline` not in the `choice` itself, but in (some of) the alternatives of the `choice`. For example, to match a list which must start with a file name, followed either by the symbol `t` or two strings, use this customization type:

```
(list file
      (choice (const t)
              (list :inline t string string)))
```

If the user chooses the first alternative in the choice, then the overall list has two elements and the second element is `t`. If the user chooses the second alternative, then the overall list has three elements and the second and third must be strings.

13.4.4 Type Keywords

You can specify keyword-argument pairs in a customization type after the type name symbol. Here are the keywords you can use, and their meanings:

`:value` *default*

This is used for a type that appears as an alternative inside of `choice`; it specifies the default value to use, at first, if and when the user selects this alternative with the menu in the customization buffer.

Of course, if the actual value of the option fits this alternative, it will appear showing the actual value, not *default*.

If `nil` is not a valid value for the alternative, then it is essential to specify a valid default with `:value`.

`:format` *format-string*

This string will be inserted in the buffer to represent the value corresponding to the type. The following '`%`' escapes are available for use in *format-string*:

'%[*button*%]'
: Display the text *button* marked as a button. The `:action` attribute specifies what the button will do if the user invokes it; its value is a function which takes two arguments—the widget which the button appears in, and the event.

  There is no way to specify two different buttons with different actions.

'%{*sample*%}'
: Show *sample* in a special face specified by `:sample-face`.

'%v'
: Substitute the item's value. How the value is represented depends on the kind of item, and (for variables) on the customization type.

'%d'
: Substitute the item's documentation string.

'%h'
: Like '%d', but if the documentation string is more than one line, add an active field to control whether to show all of it or just the first line.

'%t'
: Substitute the tag here. You specify the tag with the `:tag` keyword.

'%%'
: Display a literal '%'.

`:action` *action*
: Perform *action* if the user clicks on a button.

`:button-face` *face*
: Use the face *face* (a face name or a list of face names) for button text displayed with '%[...%]'.

`:button-prefix` *prefix*
`:button-suffix` *suffix*
: These specify the text to display before and after a button. Each can be:

  `nil` No text is inserted.

  a string The string is inserted literally.

  a symbol The symbol's value is used.

`:tag` *tag*
: Use *tag* (a string) as the tag for the value (or part of the value) that corresponds to this type.

`:doc` *doc*
: Use *doc* as the documentation string for this value (or part of the value) that corresponds to this type. In order for this to work, you must specify a value for `:format`, and use '%d' or '%h' in that value.

  The usual reason to specify a documentation string for a type is to provide more information about the meanings of alternatives inside a `:choice` type or the parts of some other composite type.

`:help-echo` *motion-doc*
: When you move to this item with `widget-forward` or `widget-backward`, it will display the string *motion-doc* in the echo area.

`:match` *function*

Specify how to decide whether a value matches the type. The corresponding value, *function*, should be a function that accepts two arguments, a widget and a value; it should return non-`nil` if the value is acceptable.

14 Loading

Loading a file of Lisp code means bringing its contents into the Lisp environment in the form of Lisp objects. XEmacs finds and opens the file, reads the text, evaluates each form, and then closes the file.

The load functions evaluate all the expressions in a file just as the `eval-current-buffer` function evaluates all the expressions in a buffer. The difference is that the load functions read and evaluate the text in the file as found on disk, not the text in an Emacs buffer.

The loaded file must contain Lisp expressions, either as source code or as byte-compiled code. Each form in the file is called a *top-level form*. There is no special format for the forms in a loadable file; any form in a file may equally well be typed directly into a buffer and evaluated there. (Indeed, most code is tested this way.) Most often, the forms are function definitions and variable definitions.

A file containing Lisp code is often called a *library*. Thus, the "Rmail library" is a file containing code for Rmail mode. Similarly, a "Lisp library directory" is a directory of files containing Lisp code.

14.1 How Programs Do Loading

XEmacs Lisp has several interfaces for loading. For example, `autoload` creates a placeholder object for a function in a file; trying to call the autoloading function loads the file to get the function's real definition (see Section 14.2 [Autoload], page 198). `require` loads a file if it isn't already loaded (see Section 14.4 [Named Features], page 201). 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 is *filename* with '`.elc`' appended. If such a file exists, it is loaded. If there is no file by that name, then `load` looks for a file named '*filename*`.el`'. If that file exists, it is loaded. Finally, if neither of those names is found, `load` looks for a file named *filename* with nothing appended, and loads it if it exists. (The `load` function is not clever about looking at *filename*. In the perverse case of a file named '`foo.el.el`', evaluation of `(load "foo.el")` will indeed find it.)

If the optional argument *nosuffix* is non-`nil`, then the suffixes '`.elc`' and '`.el`' are not tried. In this case, you must specify the precise file name you want.

If *filename* is a relative file name, such as '`foo`' or '`baz/foo.bar`', `load` searches for the file using the variable `load-path`. It appends *filename* to each of the directories listed in `load-path`, and loads the first file it finds whose name matches. The current default directory is tried only if it is specified in `load-path`, where `nil` stands for the default directory. `load` tries all three possible suffixes in the first directory in `load-path`, then all three suffixes in the second directory, and so on.

If you get a warning that '`foo.elc`' is older than '`foo.el`', it means you should consider recompiling '`foo.el`'. See Chapter 15 [Byte Compilation], page 205.

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

Any unhandled errors while loading a file terminate loading. If the load was done for the sake of `autoload`, any function definitions made during the loading are undone.

If `load` can't find the file to load, then normally it signals the error `file-error` (with '`Cannot open load file` *filename*'). But if *missing-ok* is non-`nil`, then `load` just returns `nil`.

You can use the variable `load-read-function` to specify a function for `load` to use instead of `read` for reading expressions. See below.

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

load-path User Option

The value of this variable is a list of directories to search when loading files with `load`. Each element is a string (which must be a directory name) or `nil` (which stands for the current working directory). The value of `load-path` is initialized from the environment variable `EMACSLOADPATH`, if that exists; otherwise its default value is specified in '`emacs/src/paths.h`' when XEmacs is built.

The syntax of `EMACSLOADPATH` is the same as used for `PATH`; '`:`' (or '`;`', according to the operating system) separates directory names, and '`.`' is used for the current default directory. Here is an example of how to set your `EMACSLOADPATH` variable from a `csh` '`.login`' file:

```
setenv EMACSLOADPATH .:/user/bil/emacs:/usr/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"
                    "~/emacs")
              load-path))
```

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

The command line options '`-l`' or '`-load`' specify a Lisp library to load as part of Emacs startup. Since this file might be in the current directory, Emacs 18 temporarily adds the current directory to the front of `load-path` so the file can be found there. Newer Emacs versions also find such files in the current directory, but without altering `load-path`.

Dumping Emacs uses a special value of `load-path`. If the value of `load-path` at the end of dumping is unchanged (that is, still the same special value), the dumped Emacs switches to the ordinary `load-path` value when it starts up, as described above. But if `load-path` has any other value at the end of dumping, that value is used for execution of the dumped Emacs also.

Therefore, if you want to change `load-path` temporarily for loading a few libraries in '`site-init.el`' or '`site-load.el`', you should bind `load-path` locally with `let` around the calls to `load`.

locate-file *filename path-list* &optional *suffixes mode* — Function

This function searches for a file in the same way that `load` does, and returns the file found (if any). (In fact, `load` uses this function to search through `load-path`.) It searches for *filename* through *path-list*, expanded by one of the optional *suffixes* (string of suffixes separated by '`:`'s), checking for access *mode* (0|1|2|4 = exists|executable|writeable|readable), default readable.

`locate-file` keeps hash tables of the directories it searches through, in order to speed things up. It tries valiantly to not get confused in the face of a changing and unpredictable environment, but can occasionally get tripped up. In this case, you will have to call `locate-file-clear-hashing` to get it back on track. See that function for details.

locate-file-clear-hashing *path* — Function

This function clears the hash records for the specified list of directories. `locate-file` uses a hashing scheme to speed lookup, and will correctly track the following environmental changes:

- changes of any sort to the list of directories to be searched.
- addition and deletion of non-shadowing files (see below) from the directories in the list.
- byte-compilation of a .el file into a .elc file.

`locate-file` will primarily get confused if you add a file that shadows (i.e. has the same name as) another file further down in the directory list. In this case, you must call `locate-file-clear-hashing`.

load-in-progress — Variable

This variable is non-`nil` if Emacs is in the process of loading a file, and it is `nil` otherwise.

load-read-function — Variable

This variable specifies an alternate expression-reading function for `load` and `eval-region` to use instead of `read`. The function should accept one argument, just as `read` does.

Normally, the variable's value is `nil`, which means those functions should use `read`.

load-warn-when-source-newer User Option

This variable specifies whether `load` should check whether the source is newer than the binary. If this variable is true, then when a '`.elc`' file is being loaded and the corresponding '`.el`' is newer, a warning message will be printed. The default is `nil`, but it is bound to `t` during the initial loadup.

load-warn-when-source-only User Option

This variable specifies whether `load` should warn when loading a '`.el`' file instead of an '`.elc`'. If this variable is true, then when `load` is called with a filename without an extension, and the '`.elc`' version doesn't exist but the '`.el`' version does, then a message will be printed. If an explicit extension is passed to `load`, no warning will be printed. The default is `nil`, but it is bound to `t` during the initial loadup.

load-ignore-elc-files User Option

This variable specifies whether `load` should ignore '`.elc`' files when a suffix is not given. This is normally used only to bootstrap the '`.elc`' files when building XEmacs, when you use the command '`make all-elc`'. (This forces the '`.el`' versions to be loaded in the process of compiling those same files, so that existing out-of-date '`.elc`' files do not make it mess things up.)

To learn how `load` is used to build XEmacs, see Section B.1 [Building XEmacs], page 765.

14.2 Autoload

The *autoload* facility allows you to make a function or macro known in Lisp, but put off loading the file that defines it. The first call to the function automatically reads the proper file to install the real definition and other associated code, then runs the real definition as if it had been loaded all along.

There are two ways to set up an autoloaded function: by calling `autoload`, and by writing a special "magic" comment in the source before the real definition. `autoload` is the low-level primitive for autoloading; any Lisp program can call `autoload` at any time. Magic comments do nothing on their own; they serve as a guide for the command `update-file-autoloads`, which constructs calls to `autoload` and arranges to execute them when Emacs is built. Magic comments are the most convenient way to make a function autoload, but only for packages installed along with Emacs.

autoload *function filename* &optional *docstring interactive type* Function

This function defines the function (or macro) named *function* so as to load automatically from *filename*. The string *filename* specifies the file to load to get the real definition of *function*.

The argument *docstring* is the documentation string for the function. Normally, this is the identical to the documentation string in the function definition itself. Specifying the documentation string in the call to `autoload` makes it possible to look at the documentation without loading the function's real definition.

If *interactive* is non-`nil`, 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; it's not needed unless the user actually calls *function*, and when that happens, it's time to load the real definition.

You can autoload macros and keymaps as well as ordinary functions. Specify *type* as `macro` if *function* is really a macro. Specify *type* as `keymap` if *function* is really a keymap. Various parts of Emacs need to know this information without loading the real definition.

An autoloaded keymap loads automatically during key lookup when a prefix key's binding is the symbol *function*. Autoloading does not occur for other kinds of access to the keymap. In particular, it does not happen when a Lisp program gets the keymap from the value of a variable and calls `define-key`; not even if the variable name is the same symbol *function*.

If *function* already has a non-void function definition that is not an autoload object, `autoload` does nothing and returns `nil`. If the function cell of *function* is void, or is already an autoload object, then it is defined as an autoload object like this:

```
(autoload filename docstring interactive type)
```

For example,

```
(symbol-function 'run-prolog)
     ⇒ (autoload "prolog" 169681 t nil)
```

In this case, `"prolog"` is the name of the file to load, 169681 refers to the documentation string in the '`DOC`' file (see Section 27.1 [Documentation Basics], page 381), `t` means the function is interactive, and `nil` that it is not a macro or a keymap.

The autoloaded file usually contains other definitions and may require or provide one or more features. If the file is not completely loaded (due to an error in the evaluation of its contents), any function definitions or `provide` calls that occurred during the load are undone. This is to ensure that the next attempt to call any function autoloading from this file will try again to load the file. If not for this, then some of the functions in the file might appear defined, but they might fail to work properly for the lack of certain subroutines defined later in the file and not loaded successfully.

XEmacs 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.

If the autoloaded file fails to define the desired Lisp function or macro, then an error is signaled with data `"Autoloading failed to define function` *function-name*`"`.

A magic autoload comment looks like '`;;;###autoload`', on a line by itself, just before the real definition of the function in its autoloadable source file. The command *`M-x update-file-autoloads`* writes a corresponding `autoload` call into '`loaddefs.el`'. Building Emacs loads '`loaddefs.el`' and thus calls `autoload`. *`M-x update-directory-autoloads`* is even more powerful; it updates autoloads for all files in the current directory.

The same magic comment can copy any kind of form into '`loaddefs.el`'. If the form following the magic comment is not a function definition, it is copied verbatim. You can also use a magic comment to execute a form at build time *without* executing it when the file itself is loaded. To do this, write the form *on the same line* as the magic comment. Since it is in a comment, it does nothing when you load the source file; but `update-file-autoloads` copies it to '`loaddefs.el`', where it is executed while building Emacs.

The following example shows how `doctor` is prepared for autoloading with a magic comment:

```
;;;###autoload
(defun doctor ()
  "Switch to *doctor* buffer and start giving psychotherapy."
  (interactive)
  (switch-to-buffer "*doctor*")
  (doctor-mode))
```

Here's what that produces in '`loaddefs.el`':

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

The backslash and newline immediately following the double-quote are a convention used only in the preloaded Lisp files such as '`loaddefs.el`'; they tell `make-docfile` to put the documentation string in the '`DOC`' file. See Section B.1 [Building XEmacs], page 765.

14.3 Repeated Loading

You may load one file more than once in an Emacs session. For example, after you have rewritten and reinstalled a function definition by editing it in a buffer, you may wish to return to the original version; you can do this by reloading the file it came from.

When you load or reload files, bear in mind that the `load` and `load-library` functions automatically load a byte-compiled file rather than a non-compiled file of similar name. If you rewrite a file that you intend to save and reinstall, 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 Lisp library file, keep in mind that the file 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 149.)

The simplest way to add an element to an alist is like this:

```
(setq minor-mode-alist
      (cons '(leif-mode " Leif") minor-mode-alist))
```

But this would add multiple elements if the library is reloaded. To avoid the problem, write this:

```
(or (assq 'leif-mode minor-mode-alist)
    (setq minor-mode-alist
          (cons '(leif-mode " Leif") minor-mode-alist)))
```

To add an element to a list just once, use `add-to-list` (see Section 10.7 [Setting Variables], page 152).

Occasionally you will want to test explicitly whether a library has already been loaded. Here's one way to test, in a library, whether it has been loaded before:

```
(defvar foo-was-loaded)

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

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

If the library uses `provide` to provide a named feature, you can use `featurep` to test whether the library has been loaded.

14.4 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.

A feature name is a symbol that stands for a collection of functions, variables, etc. The file that defines them should *provide* the feature. Another program that uses them may ensure they are defined by *requiring* the feature. This loads the file of definitions if it hasn't been loaded already.

To require the presence of a feature, call `require` with the feature name as argument. `require` looks in the global variable `features` to see whether the desired feature has been provided already. If not, it loads the feature from the appropriate file. This file should call `provide` at the top level to add the feature to `features`; if it fails to do so, `require` signals an error.

Features are normally named after the files that provide them, 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 'comint)` loads the file '`comint.el`' if it has not yet been loaded. This ensures that `make-comint` is defined.

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

```
(provide 'comint)
```

This adds `comint` to the global `features` list, so that `(require 'comint)` will henceforth know that nothing needs to be done.

When `require` is used at top level in a file, it takes effect when you byte-compile that file (see Chapter 15 [Byte Compilation], page 205) as well as when you load it. This is in case the required package contains macros that the byte compiler must know about.

Although top-level calls to `require` are evaluated during byte compilation, `provide` calls are not. Therefore, you can ensure that a file of definitions is loaded before it is byte-compiled by including a `provide` followed by a `require` for the same feature, as in the following example.

```
(provide 'my-feature)  ; Ignored by byte compiler,
                       ;   evaluated by load.
(require 'my-feature)  ; Evaluated by byte compiler.
```

The compiler ignores the `provide`, then processes the `require` by loading the file in question. Loading the file does execute the `provide` call, so the subsequent `require` call does nothing while loading.

provide *feature* Function

This function announces that *feature* is now loaded, or being loaded, into the current XEmacs session. This means that the facilities associated with *feature* are or will be available for other Lisp programs.

The direct effect of calling `provide` is to add *feature* to the front of the list `features` if it is not already in the list. The argument *feature* must be a symbol. `provide` returns *feature*.

```
features
     ⇒ (bar bish)

(provide 'foo)
     ⇒ foo
features
     ⇒ (foo bar bish)
```

When a file is loaded to satisfy an autoload, and it stops due to an error in the evaluating its contents, any function definitions or `provide` calls that occurred during the load are undone. See Section 14.2 [Autoload], page 198.

require *feature* &optional *filename* Function

This function checks whether *feature* is present in the current XEmacs 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 loading the file fails to provide *feature*, `require` signals an error, '`Required feature` *feature* `was not provided`'.

featurep *fexp* Function

This function returns `t` if feature *fexp* is present in this Emacs. Use this to conditionalize execution of lisp code based on the presence or absence of emacs or environment extensions.

fexp can be a symbol, a number, or a list.

If *fexp* is a symbol, it is looked up in the 'features' variable, and `t` is returned if it is found, `nil` otherwise.

If *fexp* is a number, the function returns `t` if this Emacs has an equal or greater number than `fexp`, `nil` otherwise. Note that minor Emacs version is expected to be 2 decimal places wide, so `(featurep 20.4)` will return `nil` on XEmacs 20.4—you must write `(featurep 20.04)`, unless you wish to match for XEmacs 20.40.

If *fexp* is a list whose car is the symbol `and`, the function returns `t` if all the features in its cdr are present, `nil` otherwise.

If *fexp* is a list whose car is the symbol `or`, the function returns `t` if any the features in its cdr are present, `nil` otherwise.

If *fexp* is a list whose car is the symbol `not`, the function returns `t` if the feature is not present, `nil` otherwise.

Examples:

```
(featurep 'xemacs)
     ⇒ ; t on XEmacs.

(featurep '(and xemacs gnus))
     ⇒ ; t on XEmacs with Gnus loaded.

(featurep '(or tty-frames (and emacs 19.30)))
     ⇒ ; t if this Emacs supports TTY frames.

(featurep '(or (and xemacs 19.15) (and emacs 19.34)))
     ⇒ ; t on XEmacs 19.15 and later, or on
                   ; FSF Emacs 19.34 and later.
```

Please note: The advanced arguments of this function (anything other than a symbol) are not yet supported by FSF Emacs. If you feel they are useful for supporting multiple Emacs variants, lobby Richard Stallman at '`<bug-gnu-emacs@prep.ai.mit.edu>`'.

features Variable

The value of this variable is a list of symbols that are the features loaded in the current XEmacs session. Each symbol was put in this list with a call to `provide`. The order of the elements in the `features` list is not significant.

14.5 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* &optional *force* Command

This command unloads the library that provided feature *feature*. It undefines all functions, macros, and variables defined in that library with `defconst`, `defvar`, `defun`, `defmacro`, `defsubst`, `definf-function` and `defalias`. It then restores any autoloads formerly associated with those symbols. (Loading saves these in the `autoload` property of the symbol.)

Ordinarily, `unload-feature` refuses to unload a library on which other loaded libraries depend. (A library *a* depends on library *b* if *a* contains a `require` for *b*.) If the optional argument *force* is non-`nil`, dependencies are ignored and you can unload any library.

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

load-history Variable

This variable's value is an alist connecting library names with the names of functions and variables they define, the features they provide, and the features they require.

Each element is a list and describes one library. The CAR of the list is the name of the library, as a string. The rest of the list is composed of these kinds of objects:

- Symbols that were defined by this library.
- Lists of the form `(require . `*feature*`)` indicating features that were required.
- Lists of the form `(provide . `*feature*`)` indicating features that were provided.

The value of `load-history` may have one element whose CAR is `nil`. This element describes definitions made with `eval-buffer` on a buffer that is not visiting a file.

The command `eval-region` updates `load-history`, but does so by adding the symbols defined to the element for the file being visited, rather than replacing that element.

14.6 Hooks for Loading

after-load-alist Variable

An alist of expressions to evaluate if and when particular libraries are loaded. Each element looks like this:

`(`*filename forms*`...)`

When `load` is run and the file-name argument is *filename*, the *forms* in the corresponding element are executed at the end of loading.

filename must match exactly! Normally *filename* is the name of a library, with no directory specified, since that is how `load` is normally called. An error in *forms* does not undo the load, but does prevent execution of the rest of the *forms*.

15 Byte Compilation

XEmacs 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-coded 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 XEmacs 20, the compiled code may not run in earlier versions. See Section 15.3 [Docs and Compilation], page 208.

See Section 16.3 [Compilation Errors], page 227, for how to investigate errors occurring in byte compilation.

15.1 Performance of Byte-Compiled Code

A byte-compiled function is not as efficient as a primitive function written in C, but runs much faster than the version written in Lisp. Here is an example:

```
(defun silly-loop (n)
  "Return time before and after N iterations of a loop."
  (let ((t1 (current-time-string)))
    (while (> (setq n (1- n))
              0))
    (list t1 (current-time-string))))
⇒ silly-loop

(silly-loop 5000000)
⇒ ("Fri Nov 28 20:56:16 1997"
    "Fri Nov 28 20:56:39 1997")  ; 23 seconds

(byte-compile 'silly-loop)
⇒ #<compiled-function
(from "loadup.el")
(n)
"...(23)"
[current-time-string t1 n 0]
2
"Return time before and after N iterations of a loop.">

(silly-loop 5000000)
⇒ ("Fri Nov 28 20:57:49 1997"
    "Fri Nov 28 20:57:55 1997")  ; 6 seconds
```

In this example, the interpreted code required 23 seconds to run, whereas the byte-compiled code required 6 seconds. These results are representative, but actual results will vary greatly.

15.2 The Compilation Functions

You can byte-compile an individual function or macro definition with the `byte-compile` function. You can compile a whole file with `byte-compile-file`, or several files with `byte-recompile-directory` or `batch-byte-compile`.

When you run the byte compiler, you may get warnings in a buffer called '`*Compile-Log*`'. These report things 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 178.

Normally, compiling a file does not evaluate the file's contents or load the file. But it does execute any `require` calls at top level in the file. One way to ensure that necessary macro definitions are available during compilation is to require the file that defines them (see Section 14.4 [Named Features], page 201). To avoid loading the macro definition files when someone *runs* the compiled program, write `eval-when-compile` around the `require` calls (see Section 15.5 [Eval During Compile], page 209).

byte-compile *symbol* Function

This function byte-compiles the function definition of *symbol*, replacing the previous definition with the compiled one. The function definition of *symbol* must be the actual code for the function; i.e., the compiler does not follow indirection to another symbol. `byte-compile` returns the new, compiled definition of *symbol*.

If *symbol*'s definition is a compiled-function object, `byte-compile` does nothing and returns `nil`. Lisp records only one function definition for any symbol, and if that is already compiled, non-compiled code is not available anywhere. So there is no way to "compile the same definition again."

```
(defun factorial (integer)
  "Compute factorial of INTEGER."
  (if (= 1 integer) 1
    (* integer (factorial (1- integer)))))
⇒ factorial

(byte-compile 'factorial)
⇒ #<compiled-function
(from "loadup.el")
(integer)
"...(21)"
[integer 1 factorial]
3
"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 or an operand of an instruction. The vector contains all the constants, variable names and function names used by the function, except for certain primitives that are coded as special instructions.

compile-defun &optional *arg* Command

This command reads the defun containing point, compiles it, and evaluates the result. If you use this on a defun that is actually a function definition, the effect is to install a compiled version of that function.

If *arg* is non-`nil`, the result is inserted in the current buffer after the form; otherwise, it is printed in the minibuffer.

byte-compile-file *filename* &optional *load* Command

This function compiles a file of Lisp code named *filename* into a file of byte-code. The output file's name is made by appending '`c`' to the end of *filename*.

If `load` is non-`nil`, the file is loaded after having been compiled.

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* &optional *flag* Command

This function recompiles every '`.el`' file in *directory* that needs recompilation. A file needs recompilation if a '`.elc`' file exists but is older than the '`.el`' file.

When a '`.el`' file has no corresponding '`.elc`' file, then *flag* says what to do. If it is `nil`, these files are ignored. If it is non-`nil`, the user is asked whether to compile each such file.

The returned value of this command is unpredictable.

batch-byte-compile Function

This function runs `byte-compile-file` on files specified on the command line. This function must be used only in a batch execution of Emacs, as it kills Emacs on completion. An error in one file does not prevent processing of subsequent files. (The file that gets the error will not, of course, produce any compiled code.)

```
% emacs -batch -f batch-byte-compile *.el
```

batch-byte-recompile-directory Function

This function is similar to `batch-byte-compile` but runs the command `byte-recompile-directory` on the files remaining on the command line.

byte-recompile-directory-ignore-errors-p Variable

If non-`nil`, this specifies that `byte-recompile-directory` will continue compiling even when an error occurs in a file. This is normally `nil`, but is bound to `t` by `batch-byte-recompile-directory`.

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 due to an explicit call to `byte-code`.

15.3 Documentation Strings and Compilation

Functions and variables loaded from a byte-compiled file access their documentation strings dynamically from the file whenever needed. This saves space within Emacs, and makes loading faster because the documentation strings themselves need not be processed while loading the file. Actual access to the documentation strings becomes slower as a result, but normally not enough to bother users.

Dynamic access to documentation strings does have drawbacks:

- If you delete or move the compiled file after loading it, Emacs can no longer access the documentation strings for the functions and variables in the file.
- If you alter the compiled file (such as by compiling a new version), then further access to documentation strings in this file will give nonsense results.

If your site installs Emacs following the usual procedures, these problems will never normally occur. Installing a new version uses a new directory with a different name; as long as the old version remains installed, its files will remain unmodified in the places where they are expected to be.

However, if you have built Emacs yourself and use it from the directory where you built it, you will experience this problem occasionally if you edit and recompile Lisp files. When it happens, you can cure the problem by reloading the file after recompiling it.

Byte-compiled files made with Emacs 19.29 will not load into older versions because the older versions don't support this feature. You can turn off this feature by setting `byte-compile-dynamic-docstrings` to `nil`. Once this is done, you can compile files that will load into older Emacs versions. You can do this globally, or for one source file by specifying a file-local binding for the variable. Here's one way to do that:

```
-*-byte-compile-dynamic-docstrings: nil;-*-
```

byte-compile-dynamic-docstrings Variable

If this is non-`nil`, the byte compiler generates compiled files that are set up for dynamic loading of documentation strings.

The dynamic documentation string feature writes compiled files that use a special Lisp reader construct, '`#@`*count*'. This construct skips the next *count* characters. It also uses the '`#$`' construct, which stands for "the name of this file, as a string." It is best not to use these constructs in Lisp source files.

15.4 Dynamic Loading of Individual Functions

When you compile a file, you can optionally enable the *dynamic function loading* feature (also known as *lazy loading*). With dynamic function loading, loading the file doesn't fully read the function definitions in the file. Instead, each function definition contains a place-holder which refers to the file. The first time each function is called, it reads the full definition from the file, to replace the place-holder.

The advantage of dynamic function loading is that loading the file becomes much faster. This is a good thing for a file which contains many separate commands, provided that using one of them does not imply you will soon (or ever) use the rest. A specialized mode which provides many keyboard commands often has that usage pattern: a user may invoke the mode, but use only a few of the commands it provides.

The dynamic loading feature has certain disadvantages:

- If you delete or move the compiled file after loading it, Emacs can no longer load the remaining function definitions not already loaded.
- If you alter the compiled file (such as by compiling a new version), then trying to load any function not already loaded will get nonsense results.

If you compile a new version of the file, the best thing to do is immediately load the new compiled file. That will prevent any future problems.

The byte compiler uses the dynamic function loading feature if the variable `byte-compile-dynamic` is non-`nil` at compilation time. Do not set this variable globally, since dynamic loading is desirable only for certain files. Instead, enable the feature for specific source files with file-local variable bindings, like this:

```
-*-byte-compile-dynamic: t;-*-
```

byte-compile-dynamic Variable

If this is non-`nil`, the byte compiler generates compiled files that are set up for dynamic function loading.

fetch-bytecode *function* Function

This immediately finishes loading the definition of *function* from its byte-compiled file, if it is not fully loaded already. The argument *function* may be a compiled-function object or a function name.

15.5 Evaluation During Compilation

These features permit you to write code to be evaluated during compilation of a program.

eval-and-compile *body* Special Form

This form marks *body* to be evaluated both when you compile the containing code and when you run it (whether compiled or not).

You can get a similar result by putting *body* in a separate file and referring to that file with `require`. 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 and not when the compiled program is loaded. 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 eval) ...)`. Elsewhere, the Common Lisp '`#.`' reader macro (but not when interpreting) is closer to what `eval-when-compile` does.

15.6 Compiled-Function Objects

Byte-compiled functions have a special data type: they are *compiled-function objects*.

A compiled-function object is a bit 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 compiled-function object normally begins with '`#<compiled-function`' and ends with '`>`'. However, if the variable `print-readably` is non-`nil`, the object is printed beginning with '`#[`' and ending with '`]`'. This representation can be read directly by the Lisp reader, and is used in byte-compiled files (those ending in '`.elc`').

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

A compiled-function object has a number of different elements. They are:

arglist The list of argument symbols.

instructions
The string containing the byte-code instructions.

constants The vector of Lisp objects referenced by the byte code. These include symbols used as function names and variable names.

stacksize The maximum stack size this function needs.

doc-string The documentation string (if any); otherwise, `nil`. The value may be a number or a list, in case the documentation string is stored in a file. Use the function `documentation` to get the real documentation string (see Section 27.2 [Accessing Documentation], page 382).

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.

domain The domain (if any). This is only meaningful if I18N3 (message-translation) support was compiled into XEmacs. This is a string defining which domain to find the translation for the documentation string and interactive prompt. See Section 54.2.4 [Domain Specification], page 730.

Here's an example of a compiled-function object, in printed representation. It is the definition of the command `backward-sexp`.

```
#<compiled-function
(from "lisp.elc")
(&optional arg)
"...(15)" [arg 1 forward-sexp] 2 854740 "_p">
```

The primitive way to create a compiled-function object is with `make-byte-code`:

make-byte-code &rest *elements* Function

This function constructs and returns a compiled-function object with *elements* as its elements.

Please note: Unlike all other Emacs-lisp functions, calling this with five arguments is *not* the same as calling it with six arguments, the last of which is `nil`. If the *interactive* arg is specified as `nil`, then that means that this function was defined with `(interactive)`. If the arg is not specified, then that means the function is not interactive. This is terrible behavior which is retained for compatibility with old '`.elc`' files which expected these semantics.

You should not try to come up with the elements for a compiled-function object yourself, because if they are inconsistent, XEmacs may crash when you call the function. Always leave it to the byte compiler to create these objects; it makes the elements consistent (we hope).

The following primitives are provided for accessing the elements of a compiled-function object.

compiled-function-arglist *function* Function

This function returns the argument list of compiled-function object *function*.

compiled-function-instructions *function* Function

This function returns a string describing the byte-code instructions of compiled-function object *function*.

compiled-function-constants *function* Function

This function returns the vector of Lisp objects referenced by compiled-function object *function*.

compiled-function-stack-size *function* Function

This function returns the maximum stack size needed by compiled-function object *function*.

compiled-function-doc-string *function* Function

This function returns the doc string of compiled-function object *function*, if available.

compiled-function-interactive *function* Function

This function returns the interactive spec of compiled-function object *function*, if any. The return value is `nil` or a two-element list, the first element of which is the symbol `interactive` and the second element is the interactive spec (a string or Lisp form).

compiled-function-domain *function* Function

This function returns the domain of compiled-function object *function*, if any. The result will be a string or `nil`. See Section 54.2.4 [Domain Specification], page 730.

15.7 Disassembled Byte-Code

People do not write byte-code; that job is left to the byte compiler. But we provide a disassembler to satisfy a cat-like curiosity. The disassembler converts the byte-compiled code into humanly readable form.

The byte-code interpreter is implemented as a simple stack machine. It pushes values onto a stack of its own, then pops them off to use them in calculations whose results are themselves pushed back on the stack. When a byte-code function returns, it pops a value off the stack and returns it as the value of the function.

In addition to the stack, byte-code functions can use, bind, and set ordinary Lisp variables, by transferring values between variables and the stack.

disassemble *object* &optional *stream* Command

This function prints the disassembled code for *object*. If *stream* is supplied, then output goes there. Otherwise, the disassembled code is printed to the stream `standard-output`. The argument *object* can be a function name or a lambda expression.

As a special exception, if this function is used interactively, it outputs to a buffer named '`*Disassemble*`'.

Here are two examples of using the `disassemble` function. We have added explanatory comments to help you relate the byte-code to the Lisp source; these do not appear in the output of `disassemble`. These examples show unoptimized byte-code. Nowadays byte-code is usually optimized, but we did not want to rewrite these examples, since they still serve their purpose.

```
(defun factorial (integer)
  "Compute factorial of an integer."
  (if (= 1 integer) 1
    (* integer (factorial (1- integer)))))
     ⇒ factorial

(factorial 4)
     ⇒ 24

(disassemble 'factorial)
     ⊣ byte-code for factorial:
 doc: Compute factorial of an integer.
 args: (integer)
```

```
0   constant 1            ; Push 1 onto stack.

1   varref   integer      ; Get value of integer
                          ;   from the environment
                          ;   and push the value
                          ;   onto the stack.
2   eqlsign               ; Pop top two values off stack,
                          ;   compare them,
                          ;   and push result onto stack.
3   goto-if-nil 10        ; Pop and test top of stack;
                          ;   if nil, go to 10,
                          ;   else continue.
6   constant 1            ; Push 1 onto top of stack.

7   goto     17           ; Go to 17 (in this case, 1 will be
                          ;   returned by the function).
10  constant *            ; Push symbol * onto stack.

11  varref   integer      ; Push value of integer onto stack.
12  constant factorial    ; Push factorial onto stack.

13  varref   integer      ; Push value of integer onto stack.

14  sub1                  ; Pop integer, decrement value,
                          ;   push new value onto stack.
                          ; Stack now contains:
                          ;   - decremented value of integer
                          ;   - factorial
                          ;   - value of integer
                          ;   - *
15  call     1            ; Call function factorial using
                          ;   the first (i.e., the top) element
                          ;   of the stack as the argument;
                          ;   push returned value onto stack.
                          ; Stack now contains:
                          ;   - result of recursive
                          ;        call to factorial
                          ;   - value of integer
                          ;   - *
16  call     2            ; Using the first two
                          ;   (i.e., the top two)
                          ;   elements of the stack
                          ;   as arguments,
                          ;   call the function *,
                          ;   pushing the result onto the stack.
```

```
17   return                  ; Return the top element
                             ;   of the stack.
     ⇒ nil
```

The `silly-loop` function is somewhat more complex:

```
(defun silly-loop (n)
  "Return time before and after N iterations of a loop."
  (let ((t1 (current-time-string)))
    (while (> (setq n (1- n))
              0))
    (list t1 (current-time-string))))
     ⇒ silly-loop

(disassemble 'silly-loop)
     ⊣ byte-code for silly-loop:
 doc: Return time before and after N iterations of a loop.
 args: (n)

0   constant current-time-string  ; Push
                                  ;   current-time-string
                                  ;   onto top of stack.

1   call     0               ; Call current-time-string
                             ;    with no argument,
                             ;    pushing result onto stack.

2   varbind  t1              ; Pop stack and bind t1
                             ;   to popped value.

3   varref   n               ; Get value of n from
                             ;   the environment and push
                             ;   the value onto the stack.

4   sub1                     ; Subtract 1 from top of stack.

5   dup                      ; Duplicate the top of the stack;
                             ;   i.e., copy the top of
                             ;   the stack and push the
                             ;   copy onto the stack.

6   varset   n               ; Pop the top of the stack,
                             ;   and bind n to the value.

                             ; In effect, the sequence dup varset
                             ;   copies the top of the stack
                             ;   into the value of n
                             ;   without popping it.

7   constant 0               ; Push 0 onto stack.

8   gtr                      ; Pop top two values off stack,
                             ;   test if n is greater than 0
                             ;   and push result onto stack.
```

```
9   goto-if-nil-else-pop 17 ; Goto 17 if n <= 0
                            ;    (this exits the while loop).
                            ;    else pop top of stack
                            ;    and continue

12  constant nil            ; Push nil onto stack
                            ;    (this is the body of the loop).

13  discard                 ; Discard result of the body
                            ;    of the loop (a while loop
                            ;    is always evaluated for
                            ;    its side effects).

14  goto     3              ; Jump back to beginning
                            ;    of while loop.

17  discard                 ; Discard result of while loop
                            ;    by popping top of stack.
                            ;    This result is the value nil that
                            ;    was not popped by the goto at 9.

18  varref   t1             ; Push value of t1 onto stack.

19  constant current-time-string  ; Push
                                  ;    current-time-string
                                  ;    onto top of stack.

20  call     0              ; Call current-time-string again.

21  list2                   ; Pop top two elements off stack,
                            ;    create a list of them,
                            ;    and push list onto stack.

22  unbind   1              ; Unbind t1 in local environment.

23  return                  ; Return value of the top of stack.

    ⇒ nil
```

16 Debugging Lisp Programs

There are three ways to investigate a problem in an XEmacs Lisp program, depending on what you are doing with the program when the problem appears.

- If the problem occurs when you run the program, you can use a Lisp debugger (either the default debugger or Edebug) to investigate what is happening during execution.
- If the problem is syntactic, so that Lisp cannot even read the program, you can use the XEmacs facilities for editing Lisp to localize it.
- If the problem occurs when trying to compile the program with the byte compiler, you need to know how to examine the compiler's input buffer.

Another useful debugging tool is the dribble file. When a dribble file is open, XEmacs copies all keyboard input characters to that file. Afterward, you can examine the file to find out what input was used. See Section 50.8 [Terminal Input], page 706.

For debugging problems in terminal descriptions, the `open-termscript` function can be useful. See Section 50.9 [Terminal Output], page 709.

16.1 The Lisp Debugger

The *Lisp debugger* provides the ability to suspend evaluation of a form. While evaluation is suspended (a state that is commonly known as a *break*), you may examine the run time stack, examine the values of local or global variables, or change those values. Since a break is a recursive edit, all the usual editing facilities of XEmacs are available; you can even run programs that will enter the debugger recursively. See Section 19.10 [Recursive Editing], page 310.

16.1.1 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 an 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.

When this variable is non-`nil`, Emacs does not catch errors that happen in process filter functions and sentinels. Therefore, these errors also can invoke the debugger. See Chapter 49 [Processes], page 673.

debug-ignored-errors User Option

This variable specifies certain kinds of errors that should not enter the debugger. Its value is a list of error condition symbols and/or regular expressions. If the error has any of those condition symbols, or if the error message matches any of the regular expressions, then that error does not enter the debugger, regardless of the value of `debug-on-error`.

The normal value of this variable lists several errors that happen often during editing but rarely result from bugs in Lisp programs.

To debug an error that happens during loading of the '`.emacs`' file, use the option '`-debug-init`', which binds `debug-on-error` to `t` while '`.emacs`' is loaded and inhibits use of `condition-case` to catch init file errors.

If your '`.emacs`' file sets `debug-on-error`, the effect may not last past the end of loading '`.emacs`'. (This is an undesirable byproduct of the code that implements the '`-debug-init`' command line option.) The best way to make '`.emacs`' set `debug-on-error` permanently is with `after-init-hook`, like this:

```
(add-hook 'after-init-hook
          '(lambda () (setq debug-on-error t)))
```

debug-on-signal User Option

This variable is similar to `debug-on-error` but breaks whenever an error is signalled, regardless of whether it would be handled.

16.1.2 Debugging Infinite Loops

When a program loops infinitely and fails to return, your first problem is to stop the loop. On most operating systems, you can do this with *C-g*, which causes quit.

Ordinary quitting gives no information about why the program was looping. To get more information, you can set the variable `debug-on-quit` to non-`nil`. Quitting with *C-g* is not considered an error, and `debug-on-error` has no effect on the handling of *C-g*. Likewise, `debug-on-quit` has no effect on errors.

Once you have the debugger running in the middle of the infinite loop, you can proceed from the debugger using the stepping commands. If you step through the entire loop, you will probably get enough information to solve the problem.

debug-on-quit User Option

This variable determines whether the debugger is called when `quit` is signaled and not handled. If `debug-on-quit` is non-`nil`, then the debugger is called whenever you quit (that is, type *C-g*). If `debug-on-quit` is `nil`, then the debugger is not called when you quit. See Section 19.8 [Quitting], page 306.

16.1.3 Entering the Debugger on a Function Call

To investigate a problem that happens in the middle of a program, one useful technique is to enter the debugger whenever a certain function is called. You can do this to the function in which the problem occurs, and then step through the function, or you can do this to a function called shortly before the problem, step quickly over the call to that function, and then step through its caller.

debug-on-entry *function-name* Command

This function requests *function-name* to invoke the debugger each time it is called. It works by inserting the form `(debug 'debug)` into the function definition as the first form.

Any function defined as Lisp code may be set to break on entry, regardless of whether it is interpreted code or compiled code. If the function is a command, it will enter the debugger when called from Lisp and when called interactively (after the reading of the arguments). You can't debug primitive functions (i.e., those written in C) this way.

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

If the function is already set up to invoke the debugger on entry, `debug-on-entry` does nothing.

Please note: 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)

------ Buffer: *Backtrace* ------
Entering:
* fact(3)
  eval-region(4870 4878 t)
  byte-code("...")
  eval-last-sexp(nil)
  (let ...)
  eval-insert-last-sexp(nil)
* call-interactively(eval-insert-last-sexp)
------ Buffer: *Backtrace* ------

(symbol-function 'fact)
     ⇒ (lambda (n)
          (debug (quote debug))
          (if (zerop n) 1 (* n (fact (1- n)))))
```

cancel-debug-on-entry *function-name* Command

This function undoes the effect of `debug-on-entry` on *function-name*. When called interactively, it prompts for *function-name* in the minibuffer. If *function-name* is `nil` or the empty string, it cancels debugging for all functions.

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*.

16.1.4 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 of `(debug)` isn't ignored, it will alter the execution of the program!) The most common suitable places are inside a `progn` or an implicit `progn` (see Section 9.1 [Sequencing], page 129).

16.1.5 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 XEmacs 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 19.10 [Recursive Editing], page 310) and it is wise to go back to the backtrace buffer and exit the debugger (with the `q` command) when you are finished with it. Exiting the debugger gets out of the recursive edit and kills the backtrace buffer.

The backtrace buffer shows you the functions that are executing and their argument values. It also allows you to specify a stack frame by moving point to the line describing that frame. (A stack frame is the place where the Lisp interpreter records information about a particular invocation of a function.) The frame whose line point is on is considered the *current frame*. Some of the debugger commands operate on the current frame.

The debugger itself must be run byte-compiled, since it makes assumptions about how many stack frames are used for the debugger itself. These assumptions are false if the debugger is running interpreted.

16.1.6 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 XEmacs, 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.)

Here is a list of Debugger mode commands:

`c`
: Exit the debugger and continue execution. This resumes execution of the program as if the debugger had never been entered (aside from the effect of any variables or data structures you may have changed while inside the debugger).

  Continuing when an error or quit was signalled will cause the normal action of the signalling to take place. If you do not want this to happen, but instead want the program execution to continue as if the call to `signal` did not occur, use the `r` command.

`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. The debugger alters certain important variables, and the current buffer, as part of its operation; `e` temporarily restores their outside-the-debugger values so you can examine them. This makes the debugger more transparent. By contrast, `M-:` does nothing special in the debugger; it shows you the variable values within the debugger.

`q`
: Terminate the program being debugged; return to top-level XEmacs command execution.

  If the debugger was entered due to a `C-g` but you really want to quit, and not debug, use the `q` command.

`r`
: Return a value from the debugger. The value is computed by reading an expression with the minibuffer and evaluating it.

The `r` command is useful when the debugger was invoked due to exit from a Lisp call frame (as requested with `b`); then the value specified in the `r` command is used as the value of that frame. It is also useful if you call `debug` and use its return value.

If the debugger was entered at the beginning of a function call, `r` has the same effect as `c`, and the specified return value does not matter.

If the debugger was entered through a call to `signal` (i.e. as a result of an error or quit), then returning a value will cause the call to `signal` itself to return, rather than throwing to top-level or invoking a handler, as is normal. This allows you to correct an error (e.g. the type of an argument was wrong) or continue from a `debug-on-quit` as if it never happened.

Note that some errors (e.g. any error signalled using the `error` function, and many errors signalled from a primitive function) are not continuable. If you return a value from them and continue execution, then the error will immediately be signalled again. Other errors (e.g. wrong-type-argument errors) will be continually resignalled until the problem is corrected.

16.1.7 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, showing the backtrace buffer in Debugger mode.

The Debugger mode `c` and `r` commands exit the recursive edit; then `debug` switches back to the previous buffer and returns to whatever called `debug`. This is the only way the function `debug` can return to its caller.

If the first of the *debugger-args* passed to `debug` is `nil` (or if it is not one of the special values in the table below), then `debug` displays the rest of its arguments 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 XEmacs and the debugger, and not by programmers calling `debug`.

The special values are:

`lambda` A first argument of `lambda` means `debug` was called because of entry to a function when `debug-on-next-call` was non-`nil`. The debugger displays '`Entering:`' as a line of text at the top of the buffer.

`debug` `debug` as first argument indicates a call to `debug` because of entry to a function that was set to debug on entry. The debugger displays

'Entering:', just as in the lambda case. It also marks the stack frame for that function so that it will invoke the debugger when exited.

`t` When the first argument is `t`, this indicates a call to `debug` due to evaluation of a list form when `debug-on-next-call` is non-`nil`. The debugger displays the following as the top line in the buffer:

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

`exit` When the first argument is `exit`, it indicates the exit of a stack frame previously marked to invoke the debugger on exit. The second argument given to `debug` in this case is the value being returned from the frame. The debugger displays '`Return value:`' 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.

16.1.8 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 &optional *stream detailed* 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`.

The backtrace is normally printed to `standard-output`, but this can be changed by specifying a value for *stream*. If *detailed* is non-`nil`, the backtrace also shows places where currently active variable bindings, catches, condition-cases, and unwind-protects were made as well as function calls.

In the following example, a Lisp expression calls `backtrace` explicitly. This prints the backtrace to the stream `standard-output`: in this case, to the buffer '`backtrace-output`'. Each line of the backtrace represents one function call. The line shows the values of the function's arguments if they are all known. If they are still being computed, the line says so. The arguments of special forms are elided.

```
(with-output-to-temp-buffer "backtrace-output"
  (let ((var 1))
    (save-excursion
      (setq var (eval '(progn
                         (1+ var)
                         (list 'testing (backtrace))))))))

     ⇒ nil
```

```
----------- Buffer: backtrace-output ------------
  backtrace()
  (list ...computing arguments...)
  (progn ...)
  eval((progn (1+ var) (list (quote testing) (backtrace))))
  (setq ...)
  (save-excursion ...)
  (let ...)
  (with-output-to-temp-buffer ...)
  eval-region(1973 2142 #<buffer *scratch*>)
  byte-code("...  for eval-print-last-sexp ...")
  eval-print-last-sexp(nil)
* call-interactively(eval-print-last-sexp)
----------- Buffer: backtrace-output ------------
```

The character '`*`' indicates a frame whose debug-on-exit flag is set.

debug-on-next-call Variable

If this variable is non-`nil`, it says to call the debugger before the next `eval`, `apply` or `funcall`. Entering the debugger sets `debug-on-next-call` to `nil`.

The *d* command in the debugger works by setting this variable.

backtrace-debug *level flag* Function

This function sets the debug-on-exit flag of the stack frame *level* levels down the stack, giving it the value *flag*. If *flag* is non-`nil`, this will cause the debugger to be entered when that frame later exits. Even a nonlocal exit through that frame will enter the debugger.

This function is used only by the debugger.

command-debug-status Variable

This variable records the debugging status of the current interactive command. Each time a command is called interactively, this variable is bound to `nil`. The debugger can set this variable to leave information for future debugger invocations during the same command.

The advantage, for the debugger, of using this variable rather than another global variable is that the data will never carry over to a subsequent command invocation.

backtrace-frame *frame-number* Function

The function `backtrace-frame` is intended for use in Lisp debuggers. It returns information about what computation is happening in the stack frame *frame-number* levels down.

If that frame has not evaluated the arguments yet (or is a special form), the value is `(nil` *function arg-forms*`...)`.

If that frame has evaluated its arguments and called its function already, the value is `(t` *function arg-values*`...)`.

In the return value, *function* is whatever was supplied as the CAR of the evaluated list, or a `lambda` expression in the case of a macro call. If the function has a `&rest` argument, that is represented as the tail of the list *arg-values*.

If *frame-number* is out of range, `backtrace-frame` returns `nil`.

16.2 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 does not say where the missing parenthesis belongs. How, then, to find what to change?

If the problem is not simply an imbalance of parentheses, a useful technique is to try *C-M-e* at the beginning of each defun, and see if it goes to the place where that defun appears to end. If it does not, there is a problem in that defun.

However, unmatched parentheses are the most common syntax errors in Lisp, and we can give further advice for those cases.

16.2.1 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.

Now you can go to the beginning of the defun and type *C-M-q*. Usually all the lines from a certain point to the end of the function will shift to the right. There is probably a missing close parenthesis, or a superfluous open parenthesis, near that point. (However, don't assume this is true; study the code to make sure.) Once you have found the discrepancy, undo the *C-M-q* with *C-_*, since the old indentation is probably appropriate to the intended parentheses.

After you think you have fixed the problem, use *C-M-q* again. If the old indentation actually fit the intended nesting of parentheses, and you have put back those parentheses, *C-M-q* should not change anything.

16.2.2 Excess Close Parentheses

To deal with an excess close parenthesis, first insert an open parenthesis at the beginning of the file, back up over it, and type *C-M-f* to find the end of the unbalanced defun.

(Then type *C-*⟨SPC⟩ *C-_* *C-u* *C-*⟨SPC⟩ to set the mark there, undo the insertion of the open parenthesis, and finally return to the mark.)

Then find the actual matching close parenthesis by typing *C-M-f* at the beginning of 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* with *C-_*, since the old indentation is probably appropriate to the intended parentheses.

After you think you have fixed the problem, use *C-M-q* again. If the old indentation actually fit the intended nesting of parentheses, and you have put back those parentheses, *C-M-q* should not change anything.

16.3 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. Either way, here is how to find out where in the file the error occurred.

What you should do is switch to the buffer '` *Compiler Input*`'. (Note that the buffer name starts with a space, so it does not show up in *M-x list-buffers*.) This buffer contains the program being compiled, and point shows how far the byte compiler was able to read.

If the error was due to invalid Lisp syntax, point shows exactly where the invalid syntax was *detected*. The cause of the error is not necessarily near by! Use the techniques in the previous section to find the error.

If the error was detected while compiling a form that had been read successfully, then point is located at the end of the form. In this case, this technique can't localize the error precisely, but can still show you which function to check.

16.4 Edebug

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

- Step through evaluation, stopping before and after each expression.
- Set conditional or unconditional breakpoints, install embedded breakpoints, or a global break event.
- Trace slow or fast stopping briefly at each stop point, or each breakpoint.
- Display expression results and evaluate expressions as if outside of Edebug. Interface with the custom printing package for printing circular structures.
- Automatically reevaluate a list of expressions and display their results each time Edebug updates the display.

- Output trace info on function enter and exit.
- Errors stop before the source causing the error.
- Display backtrace without Edebug calls.
- Allow specification of argument evaluation for macros and defining forms.
- Provide rudimentary coverage testing and display of frequency counts.

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

16.4.1 Using Edebug

To debug an XEmacs Lisp program with Edebug, you must first *instrument* the Lisp code that you want to debug. If you want to just try it now, load '`edebug.el`', move point into a definition and do *C-u C-M-x* (`eval-defun` with a prefix argument). See Section 16.4.2 [Instrumenting], page 229 for alternative ways to instrument code.

Once a function is instrumented, any call to the function activates Edebug. Activating Edebug may stop execution and let you step through the function, or it may update the display and continue execution while checking for debugging commands, depending on the selected Edebug execution mode. The initial execution mode is `step`, by default, which does stop execution. See Section 16.4.3 [Edebug Execution Modes], page 230.

Within Edebug, you normally view an XEmacs buffer showing the source of the Lisp function you are debugging. This is referred to as the *source code buffer*—but note that it is not always the same buffer depending on which function is currently being executed.

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 you can move point yourself.

If you instrument the definition of `fac` (shown below) and then execute `(fac 3)`, here is what you normally see. Point is at the open-parenthesis before `if`.

```
(defun fac (n)
=>⋆(if (< 0 n)
      (* n (fac (1- n)))
    1))
```

The places within a function where Edebug can stop execution are called *stop points*. These occur both before and after each subexpression that is a list, and also after each variable reference. Here we show with periods the stop points found in the function `fac`:

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

While the source code buffer is selected, the special commands of Edebug are available in it, in addition to the commands of XEmacs Lisp mode. (The buffer is temporarily made read-only, however.) For example, you can type the Edebug command ⟨SPC⟩ to execute until the next stop point. If you type ⟨SPC⟩ once after entry to `fac`, here is the display you will see:

```
(defun fac (n)
=>(if ⋆(< 0 n)
```

```
      (* n (fac (1- n)))
    1))
```

When Edebug stops execution after an expression, it displays the expression's value in the echo area.

Other frequently used commands are *b* to set a breakpoint at a stop point, *g* to execute until a breakpoint is reached, and *q* to exit to the top-level command loop. Type *?* to display a list of all Edebug commands.

16.4.2 Instrumenting for Edebug

In order to use Edebug to debug Lisp code, you must first *instrument* the code. Instrumenting a form inserts additional code into it which invokes Edebug at the proper places. Furthermore, if Edebug detects a syntax error while instrumenting, point is left at the erroneous code and an `invalid-read-syntax` error is signaled.

Once you have loaded Edebug, the command *C-M-x* (`eval-defun`) is redefined so that when invoked with a prefix argument on a definition, it instruments the definition before evaluating it. (The source code itself is not modified.) If the variable `edebug-all-defs` is non-`nil`, that inverts the meaning of the prefix argument: then *C-M-x* instruments the definition *unless* it has a prefix argument. The default value of `edebug-all-defs` is `nil`. The command *M-x edebug-all-defs* toggles the value of the variable `edebug-all-defs`.

If `edebug-all-defs` is non-`nil`, then the commands `eval-region`, `eval-current-buffer`, and `eval-buffer` also instrument any definitions they evaluate. Similarly, `edebug-all-forms` controls whether `eval-region` should instrument *any* form, even non-defining forms. This doesn't apply to loading or evaluations in the minibuffer. The command *M-x edebug-all-forms* toggles this option.

Another command, *M-x edebug-eval-top-level-form*, is available to instrument any top-level form regardless of the value of `edebug-all-defs` or `edebug-all-forms`.

Just before Edebug instruments any code, it calls any functions in the variable `edebug-setup-hook` and resets its value to `nil`. You could use this to load up Edebug specifications associated with a package you are using but only when you also use Edebug. For example, '`my-specs.el`' may be loaded automatically when you use `my-package` with Edebug by including the following code in '`my-package.el`'.

```
(add-hook 'edebug-setup-hook
  (function (lambda () (require 'my-specs))))
```

While Edebug is active, the command *I* (`edebug-instrument-callee`) instruments the definition of the function or macro called by the list form after point, if is not already instrumented. If the location of the definition is not known to Edebug, this command cannot be used. After loading Edebug, `eval-region` records the position of every definition it evaluates, even if not instrumenting it. Also see the command *i* (Section 16.4.4 [Jumping], page 231) which steps into the callee.

Edebug knows how to instrument all the standard special forms, an interactive form with an expression argument, anonymous lambda expressions, and other defining forms. (Specifications for macros defined by '`cl.el`' (version 2.03) are provided in '`cl-specs.el`'.) Edebug cannot know what a user-defined macro will do with the arguments of a macro call so you must tell it. See Section 16.4.16 [Instrumenting Macro Calls], page 240 for the details.

Note that a couple ways remain to evaluate expressions without instrumenting them. Loading a file via the `load` subroutine does not instrument expressions for Edebug. Evaluations in the minibuffer via `eval-expression` (*M-ESC*) are not instrumented.

To remove instrumentation from a definition, simply reevaluate it with one of the non-instrumenting commands, or reload the file.

See Section 16.4.9 [Edebug Eval], page 235 for other evaluation functions available inside of Edebug.

16.4.3 Edebug Execution Modes

Edebug supports several execution modes for running the program you are debugging. We call these alternatives *Edebug execution modes*; do not confuse them with major or minor modes. The current Edebug execution mode determines how 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 execution mode by typing a command to continue the program in a certain mode. Here is a table of these commands. All except for *S* resume execution of the program, at least for a certain distance.

S Stop: don't execute any more of the program for now, just wait for more Edebug commands (`edebug-stop`).

⟨SPC⟩ Step: stop at the next stop point encountered (`edebug-step-mode`).

n Next: stop at the next stop point encountered after an expression (`edebug-next-mode`). Also see `edebug-forward-sexp` in Section 16.4.5 [Edebug Misc], page 232.

t Trace: pause one second at each Edebug stop point (`edebug-trace-mode`).

T Rapid trace: update at each stop point, but don't actually pause (`edebug-Trace-fast-mode`).

g Go: run until the next breakpoint (`edebug-go-mode`). See Section 16.4.6 [Breakpoints], page 232.

c Continue: pause for one second at each breakpoint, but don't stop (`edebug-continue-mode`).

C Rapid continue: update at each breakpoint, but don't actually pause (`edebug-Continue-fast-mode`).

G Go non-stop: ignore breakpoints (`edebug-Go-nonstop-mode`). You can still stop the program by hitting any key.

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 initial execution mode comes from the value of the variable `edebug-initial-mode`. By default, this specifies `step` mode. Note that you may reenter the same Edebug level several times if, for example, an instrumented function is called several times from one command.

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 `S` 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 Edebug commands do not work; when you exit from Edebug, to resume the program, whether you are defining or executing a keyboard macro is forgotten. Also, defining or executing a keyboard macro outside of Edebug does not affect the command loop inside Edebug. This is usually an advantage. But see `edebug-continue-kbd-macro`.

16.4.4 Jumping

Commands described here let you jump to a specified location. All, except `i`, use temporary breakpoints to establish the stop point and then switch to `go` mode. Any other breakpoint reached before the intended stop point will also stop execution. See Section 16.4.6 [Breakpoints], page 232 for the details on breakpoints.

`f`
: Run the program forward over one expression (`edebug-forward-sexp`). 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.

  With a prefix argument *n*, the temporary breakpoint is placed *n* sexps beyond point. If the containing list ends before *n* more elements, then the place to stop is after the containing expression.

  Be careful that the position `C-M-f` finds is a place that the program will really get to; this may not be true in a `cond`, for example.

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

`o`
: Continue "out of" an expression (`edebug-step-out`). It places a temporary breakpoint at the end of the sexp containing point.

  If the containing sexp is a function definition itself, it continues until just before the last sexp in the definition. If that is where you are now, it returns from the function and then stops. In other words, this command does not exit the currently executing function unless you are positioned after the last sexp.

`I`
: Step into the function or macro after point after first ensuring that it is instrumented. It does this by calling `edebug-on-entry` and then switching to `go` mode.

  Although the automatic instrumentation is convenient, it is not later automatically uninstrumented.

h Proceed to the stop point near where point is using a temporary breakpoint (`edebug-goto-here`).

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.

16.4.5 Miscellaneous

Some miscellaneous commands are described here.

? Display the help message for Edebug (`edebug-help`).

C-] Abort one level back to the previous command level (`abort-recursive-edit`).

q Return to the top level editor command loop (`top-level`). This exits all recursive editing levels, including all levels of Edebug activity. However, instrumented code protected with `unwind-protect` or `condition-case` forms may resume debugging.

Q Like *q* but don't stop even for protected code (`top-level-nonstop`).

r Redisplay the most recently known expression result in the echo area (`edebug-previous-result`).

d Display a backtrace, excluding Edebug's own functions for clarity (`edebug-backtrace`).

You cannot use debugger commands in the backtrace buffer in Edebug as you would in the standard debugger.

The backtrace buffer is killed automatically when you continue execution.

From the Edebug recursive edit, you may invoke commands that activate Edebug again recursively. Any time Edebug is active, you can quit to the top level with *q* or abort one recursive edit level with *C-]*. You can display a backtrace of all the pending evaluations with *d*.

16.4.6 Breakpoints

There are three more ways to stop execution once it has started: breakpoints, the global break condition, and embedded 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 16.4.1 [Using Edebug], page 228. For setting and unsetting breakpoints, the stop point that is affected is the first one at or after point in the source code buffer. Here are the Edebug commands for breakpoints:

b Set a breakpoint at the stop point at or after point (`edebug-set-breakpoint`). If you use a prefix argument, the breakpoint is temporary (it turns off the first time it stops the program).

u Unset the breakpoint (if any) at the stop point at or after the current point (`edebug-unset-breakpoint`).

`x` *condition* ⟨RET⟩
: Set a conditional breakpoint which stops the program only if *condition* evaluates to a non-`nil` value (`edebug-set-conditional-breakpoint`). 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 definition (`edebug-next-breakpoint`).

While in Edebug, you can set a breakpoint with `b` and unset one with `u`. 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 or reinstrumenting a definition clears all its breakpoints.

A *conditional breakpoint* tests a condition each time the program gets there. To set a conditional breakpoint, use `x`, and specify the condition expression in the minibuffer. Setting a conditional breakpoint at a stop point that already has a conditional breakpoint puts the current condition expression in the minibuffer so you can edit it.

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 `B`, which moves point to the next breakpoint in the definition 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.

16.4.6.1 Global Break Condition

In contrast to breaking when execution reaches specified locations, you can also cause a break when a certain event occurs. The *global break condition* is a condition that is repeatedly evaluated at every stop point. If it evaluates to a non-`nil` value, then execution is stopped or paused depending on the execution mode, just like a breakpoint. Any errors that might occur as a result of evaluating the condition are ignored, as if the result were `nil`.

You can set or edit the condition expression, stored in `edebug-global-break-condition`, using `X` (`edebug-set-global-break-condition`).

Using the global break condition is perhaps the fastest way to find where in your code some event occurs, but since it is rather expensive you should reset the condition to `nil` when not in use.

16.4.6.2 Embedded Breakpoints

Since all breakpoints in a definition are cleared each time you reinstrument it, you might rather create an *embedded breakpoint* which is simply a call to the function `edebug`. You can, of course, make such a call conditional. For example, in the `fac` function, insert the first line as shown below to stop when the argument reaches zero:

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

When the `fac` definition is instrumented and the function is called, Edebug will stop before the call to `edebug`. Depending on the execution mode, Edebug will stop or pause.

However, if no instrumented code is being executed, calling `edebug` will instead invoke `debug`. Calling `debug` will always invoke the standard backtrace debugger.

16.4.7 Trapping Errors

An error may be signaled by subroutines or XEmacs Lisp code. If a signal is not handled by a `condition-case`, this indicates an unrecognized situation has occurred. If Edebug is not active when an unhandled error is signaled, `debug` is run normally (if `debug-on-error` is non-`nil`). But while Edebug is active, `debug-on-error` and `debug-on-quit` are bound to `edebug-on-error` and `edebug-on-quit`, which are both `t` by default. Actually, if `debug-on-error` already has a non-`nil` value, that value is still used.

It is best to change the values of `edebug-on-error` or `edebug-on-quit` when Edebug is not active since their values won't be used until the next time Edebug is invoked at a deeper command level. If you only change `debug-on-error` or `debug-on-quit` while Edebug is active, these changes will be forgotten when Edebug becomes inactive. Furthermore, during Edebug's recursive edit, these variables are bound to the values they had outside of Edebug.

Edebug shows you the last stop point that it knew about before the error was signaled. This may be the location of a call to a function which was not instrumented, within which the error actually occurred. For an unbound variable error, the last known stop point might be quite distant from the offending variable. If the cause of the error is not obvious at first, note that you can also get a full backtrace inside of Edebug (see Section 16.4.5 [Edebug Misc], page 232).

Edebug can also trap signals even if they are handled. If `debug-on-error` is a list of signal names, Edebug will stop when any of these errors are signaled. Edebug shows you the last known stop point just as for unhandled errors. After you continue execution, the error is signaled again (but without being caught by Edebug). Edebug can only trap errors that are handled if they are signaled in Lisp code (not subroutines) since it does so by temporarily replacing the `signal` function.

16.4.8 Edebug Views

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

v View the outside window configuration (`edebug-view-outside`).

p Temporarily display the outside current buffer with point at its outside position (`edebug-bounce-point`). If prefix arg is supplied, sit for that many seconds instead.

`w` Move point back to the current stop point (`edebug-where`) in the source code buffer. Also, if you use this command in another window displaying the same buffer, this window will be used instead to display the buffer in the future.

`W` Toggle the `edebug-save-windows` variable which indicates whether the outside window configuration is saved and restored (`edebug-toggle-save-windows`). Also, each time it is toggled on, make the outside window configuration the same as the current window configuration.

With a prefix argument, `edebug-toggle-save-windows` only toggles saving and restoring of the selected window. To specify a window that is not displaying the source code buffer, you must use `C-xXW` from the global keymap.

You can view the outside window configuration with `v` or just bounce to the current point in the current buffer with `p`, even if it is not normally displayed. After moving point, you may wish to pop back to the stop point with `w` from a source code buffer.

By using `W` twice, Edebug again saves and restores the outside window configuration, but to the current configuration. This is a convenient way to, for example, add another buffer to be displayed whenever Edebug is active. However, the automatic redisplay of '`*edebug*`' and '`*edebug-trace*`' may conflict with the buffers you wish to see unless you have enough windows open.

16.4.9 Evaluation

While within Edebug, you can evaluate expressions "as if" Edebug were not running. Edebug tries to be invisible to the expression's evaluation and printing. Evaluation of expressions that cause side effects will work as expected except for things that Edebug explicitly saves and restores. See Section 16.4.15 [The Outside Context], page 239 for details on this process. Also see Section 16.4.11 [Reading in Edebug], page 237 and Section 16.4.12 [Printing in Edebug], page 237 for topics related to evaluation.

`e` *exp* ⟨RET⟩
Evaluate expression *exp* in the context outside of Edebug (`edebug-eval-expression`). In other words, 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 before point, in the context outside of Edebug (`edebug-eval-last-sexp`).

Edebug supports evaluation of expressions containing references to lexically bound symbols created by the following constructs in '`cl.el`' (version 2.03 or later): `lexical-let`, `macrolet`, and `symbol-macrolet`.

16.4.10 Evaluation List Buffer

You can use the *evaluation list buffer*, called '`*edebug*`', to evaluate expressions interactively. You can also set up the *evaluation list* of expressions to be evaluated automatically each time Edebug updates the display.

E Switch to the evaluation list buffer '`*edebug*`' (`edebug-visit-eval-list`).

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

LFD Evaluate the expression before point, in the outside context, and insert the value in the buffer (`edebug-eval-print-last-sexp`).

C-x C-e Evaluate the expression before point, in the context outside of Edebug (`edebug-eval-last-sexp`).

C-c C-u Build a new evaluation list from the first expression of each group, reevaluate and redisplay (`edebug-update-eval-list`). Groups are separated by comment lines.

C-c C-d Delete the evaluation list group that point is in (`edebug-delete-eval-item`).

C-c C-w Switch back to the source code buffer at the current stop point (`edebug-where`).

You can evaluate expressions in the evaluation list window with *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 unless you add them to the evaluation list with *C-c C-u*. This command builds a new list from the first expression of each *evaluation list group*. Groups are separated by comment lines. Be careful not to add expressions that execute instrumented code otherwise an infinite loop will result.

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, for example, use variables not currently valid do not interrupt your debugging.

Here is an example of what the evaluation list window looks like after several expressions have been added to it:

```
(current-buffer)
#<buffer *scratch*>
;---------------------------------------------------------------
(selected-window)
#<window 16 on *scratch*>
;---------------------------------------------------------------
(point)
196
;---------------------------------------------------------------
bad-var
"Symbol's value as variable is void: bad-var"
;---------------------------------------------------------------
(recursion-depth)
0
;---------------------------------------------------------------
this-command
eval-last-sexp
;---------------------------------------------------------------
```

To delete a group, move point into it and type *C-c C-d*, or simply delete the text for the group and update the evaluation list with *C-c C-u*. When you add a new group, be sure it is separated from its neighbors by a comment line.

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

16.4.11 Reading in Edebug

To instrument a form, Edebug first reads the whole form. Edebug replaces the standard Lisp Reader with its own reader that remembers the positions of expressions. This reader is used by the Edebug replacements for `eval-region`, `eval-defun`, `eval-buffer`, and `eval-current-buffer`.

Another package, '`cl-read.el`', replaces the standard reader with one that understands Common Lisp reader macros. If you use that package, Edebug will automatically load '`edebug-cl-read.el`' to provide corresponding reader macros that remember positions of expressions. If you define new reader macros, you will have to define similar reader macros for Edebug.

16.4.12 Printing in Edebug

If the result of an expression in your program contains a circular reference, you may get an error when Edebug attempts to print it. You can set `print-length` to a non-zero value to limit the print length of lists (the number of cdrs), and in Emacs 19, set `print-level` to a non-zero value to limit the print depth of lists. But you can print such circular structures and structures that share elements more informatively by using the '`cust-print`' package.

To load '`cust-print`' and activate custom printing only for Edebug, simply use the command *M-x edebug-install-custom-print*. To restore the standard print functions, use *M-x edebug-uninstall-custom-print*. You can also activate custom printing for printing in any Lisp code; see the package for details.

Here is an example of code that creates a circular structure:

```
(progn
  (edebug-install-custom-print)
  (setq a '(x y))
  (setcar a a))
```

Edebug will print the result of the `setcar` as '`Result: #1=(#1# y)`'. The '`#1=`' notation names the structure that follows it, and the '`#1#`' notation references the previously named structure. This notation is used for any shared elements of lists or vectors.

Independent of whether '`cust-print`' is active, while printing results Edebug binds `print-length`, `print-level`, and `print-circle` to `edebug-print-length` (`50`), `edebug-print-level` (`50`), and `edebug-print-circle` (`t`) respectively, if these values are non-`nil`. Also, `print-readably` is bound to `nil` since some objects simply cannot be printed readably.

16.4.13 Tracing

In addition to automatic stepping through source code, which is also called *tracing* (see Section 16.4.3 [Edebug Execution Modes], page 230), Edebug can produce a traditional trace listing of execution in a separate buffer, '`*edebug-trace*`'.

If the variable `edebug-trace` is non-nil, each function entry and exit adds lines to the trace buffer. On function entry, Edebug prints '`::::{`' followed by the function name and argument values. On function exit, Edebug prints '`::::}`' followed by the function name and result of the function. The number of '`:`'s is computed from the recursion depth. The balanced braces in the trace buffer can be used to find the matching beginning or end of function calls. These displays may be customized by replacing the functions `edebug-print-trace-before` and `edebug-print-trace-after`, which take an arbitrary message string to print.

The macro `edebug-tracing` provides tracing similar to function enter and exit tracing, but for arbitrary expressions. This macro should be explicitly inserted by you around expressions you wish to trace the execution of. The first argument is a message string (evaluated), and the rest are expressions to evaluate. The result of the last expression is returned.

Finally, you can insert arbitrary strings into the trace buffer with explicit calls to `edebug-trace`. The arguments of this function are the same as for `message`, but a newline is always inserted after each string printed in this way.

`edebug-tracing` and `edebug-trace` insert lines in the trace buffer even if Edebug is not active. Every time the trace buffer is added to, the window is scrolled to show the last lines inserted. (There may be some display problems if you use tracing along with the evaluation list.)

16.4.14 Coverage Testing

Edebug provides a rudimentary coverage tester and display of execution frequency. Frequency counts are always accumulated, both before and after evaluation of each instrumented expression, even if the execution mode is `Go-nonstop`. Coverage testing is only done if the option `edebug-test-coverage` is non-`nil` because this is relatively expensive. Both data sets are displayed by *`M-x edebug-display-freq-count`*.

edebug-display-freq-count Command

> Display the frequency count data for each line of the current definition. The frequency counts are inserted as comment lines after each line, and you can undo all insertions with one `undo` command. The counts are inserted starting under the *(* before an expression or the *)* after an expression, or on the last char of a symbol. The counts are only displayed when they differ from previous counts on the same line.
>
> If coverage is being tested, whenever all known results of an expression are `eq`, the char `=` will be appended after the count for that expression. Note that this is always the case for an expression only evaluated once.
>
> To clear the frequency count and coverage data for a definition, reinstrument it.

For example, after evaluating `(fac 5)` with an embedded breakpoint, and setting `edebug-test-coverage` to `t`, when the breakpoint is reached, the frequency data is looks like this:

```
(defun fac (n)
  (if (= n 0) (edebug))
;#6           1      0 =5
  (if (< 0 n)
;#5         =
      (* n (fac (1- n)))
;#    5               0
    1))
;#   0
```

The comment lines show that `fac` has been called 6 times. The first `if` statement has returned 5 times with the same result each time, and the same is true for the condition on the second `if`. The recursive call of `fac` has not returned at all.

16.4.15 The Outside Context

Edebug tries to be transparent to the program you are debugging. In addition, most evaluations you do within Edebug (see Section 16.4.9 [Edebug Eval], page 235) occur in the same outside context which is temporarily restored for the evaluation. But Edebug is not completely successful and this section explains precisely how it fails. Edebug operation unavoidably alters some data in XEmacs, and this can interfere with debugging certain programs. Also notice that Edebug's protection against change of outside data means that any side effects *intended* by the user in the course of debugging will be defeated.

16.4.15.1 Checking Whether to Stop

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 one time to reduce Edebug's impact on the stack. You could, however, still run out of stack space when using Edebug.
- The state of keyboard macro execution is saved and restored. While Edebug is active, `executing-macro` is bound to `edebug-continue-kbd-macro`.

16.4.15.2 Edebug Display Update

When Edebug needs to display something (e.g., in trace mode), it saves the current window configuration from "outside" Edebug. When you exit Edebug (by continuing the program), it restores the previous window configuration.

XEmacs redisplays only when it pauses. Usually, when you continue execution, the program comes back into Edebug at a breakpoint or after stepping without pausing or reading input in between. In such cases, XEmacs never gets a chance to redisplay the "outside" configuration. What you see is the same window configuration as the last time Edebug was active, with no interruption.

Entry to Edebug for displaying something also saves and restores the following data, but some of these are deliberately not restored if an error or quit signal occurs.

- Which buffer is current, and where point and mark are in the current buffer are saved and restored.
- The Edebug Display Update, is saved and restored if `edebug-save-windows` is non-`nil`. It is not restored on error or quit, but the outside selected window *is* reselected even on error or quit in case a `save-excursion` is active. If the value of `edebug-save-windows` is a list, only the listed windows are saved and restored.

  The window start and horizontal scrolling of the source code buffer are not restored, however, so that the display remains coherent.
- The value of point in each displayed buffer is saved and restored if `edebug-save-displayed-buffer-points` is non-`nil`.
- The variables `overlay-arrow-position` and `overlay-arrow-string` are saved and restored. So you can safely invoke Edebug from the recursive edit elsewhere in the same buffer.
- `cursor-in-echo-area` is locally bound to `nil` so that the cursor shows up in the window.

16.4.15.3 Edebug Recursive Edit

When Edebug is entered and actually reads commands from the user, it saves (and later restores) these additional data:

- The current match data, for whichever buffer was current.
- `last-command`, `this-command`, `last-command-char`, `last-input-char`, `last-input-event`, `last-command-event`, `last-event-frame`, `last-nonmenu-event`, and `track-mouse` . Commands used within Edebug do not affect these variables outside of Edebug.

  The key sequence returned by `this-command-keys` is changed by executing commands within Edebug and there is no way to reset the key sequence from Lisp.

  For Emacs 18, Edebug cannot save and restore the value of `unread-command-char`. Entering Edebug while this variable has a nontrivial value can interfere with execution of the program you are debugging.
- Complex commands executed while in Edebug are added to the variable `command-history`. In rare cases this can alter execution.
- Within Edebug, the recursion depth appears one deeper than the recursion depth outside Edebug. This is not true of the automatically updated evaluation list window.
- `standard-output` and `standard-input` are bound to `nil` by the `recursive-edit`, but Edebug temporarily restores them during evaluations.
- The state of keyboard macro definition is saved and restored. While Edebug is active, `defining-kbd-macro` is bound to `edebug-continue-kbd-macro`.

16.4.16 Instrumenting Macro Calls

When Edebug instruments an expression that calls a Lisp macro, it needs additional advice to do the job properly. This is because there is no way to tell which subexpressions of the macro call may be evaluated. (Evaluation may occur explicitly in the macro body, or when the resulting expansion is evaluated, or any time later.) You must explain the format of macro call arguments by using `def-edebug-spec` to define an *Edebug specification* for each macro.

def-edebug-spec *macro specification* Macro

Specify which expressions of a call to macro *macro* are forms to be evaluated. For simple macros, the *specification* often looks very similar to the formal argument list of the macro definition, but specifications are much more general than macro arguments.

The *macro* argument may actually be any symbol, not just a macro name.

Unless you are using Emacs 19 or XEmacs, this macro is only defined in Edebug, so you may want to use the following which is equivalent: `(put '`*macro* `'edebug-form-spec '`*specification*`)`

Here is a simple example that defines the specification for the `for` macro described in the XEmacs Lisp Reference Manual, followed by an alternative, equivalent specification.

```
(def-edebug-spec for
  (symbolp "from" form "to" form "do" &rest form))

(def-edebug-spec for
  (symbolp ['from form] ['to form] ['do body]))
```

Here is a table of the possibilities for *specification* and how each directs processing of arguments.

- `t` All arguments are instrumented for evaluation.
- `0` None of the arguments is instrumented.
- a symbol The symbol must have an Edebug specification which is used instead. This indirection is repeated until another kind of specification is found. This allows you to inherit the specification for another macro.
- a list The elements of the list describe the types of the arguments of a calling form. The possible elements of a specification list are described in the following sections.

16.4.16.1 Specification List

A *specification list* is required for an Edebug specification if some arguments of a macro call are evaluated while others are not. Some elements in a specification list match one or more arguments, but others modify the processing of all following elements. The latter, called *keyword specifications*, are symbols beginning with '`&`' (e.g. `&optional`).

A specification list may contain sublists which match arguments that are themselves lists, or it may contain vectors used for grouping. Sublists and groups thus subdivide the specification list into a hierarchy of levels. Keyword specifications only apply to the remainder of the sublist or group they are contained in and there is an implicit grouping around a keyword specification and all following elements in the sublist or group.

If a specification list fails at some level, then backtracking may be invoked to find some alternative at a higher level, or if no alternatives remain, an error will be signaled. See Section 16.4.16.2 [Backtracking], page 244 for more details.

Edebug specifications provide at least the power of regular expression matching. Some context-free constructs are also supported: the matching of sublists with balanced parentheses, recursive processing of forms, and recursion via indirect specifications.

Each element of a specification list may be one of the following, with the corresponding type of argument:

`sexp` A single unevaluated expression.

`form` A single evaluated expression, which is instrumented.

`place` A place as in the Common Lisp `setf` place argument. It will be instrumented just like a form, but the macro is expected to strip the instrumentation. Two functions, `edebug-unwrap` and `edebug-unwrap*`, are provided to strip the instrumentation one level or recursively at all levels.

`body` Short for `&rest form`. See `&rest` below.

`function-form`
A function form: either a quoted function symbol, a quoted lambda expression, or a form (that should evaluate to a function symbol or lambda expression). This is useful when function arguments might be quoted with `quote` rather than `function` since the body of a lambda expression will be instrumented either way.

`lambda-expr`
An unquoted anonymous lambda expression.

`&optional`
All following elements in the specification list are optional; as soon as one does not match, Edebug stops matching at this level.

To make just a few elements optional followed by non-optional elements, use `[&optional` *specs*`...]`. To specify that several elements should all succeed together, use `&optional [`*specs*`...]`. See the `defun` example below.

`&rest` All following elements in the specification list are repeated zero or more times. All the elements need not match in the last repetition, however.

To repeat only a few elements, use `[&rest` *specs*`...]`. To specify all elements must match on every repetition, use `&rest [`*specs*`...]`.

`&or` Each of the following elements in the specification list is an alternative, processed left to right until one matches. One of the alternatives must match otherwise the `&or` specification fails.

Each list element following `&or` is a single alternative even if it is a keyword specification. (This breaks the implicit grouping rule.) To group two or more list elements as a single alternative, enclose them in `[...]`.

`&not` Each of the following elements is matched as alternatives as if by using `&or`, but if any of them match, the specification fails. If none of them match, nothing is matched, but the `&not` specification succeeds.

`&define` Indicates that the specification is for a defining form. The defining form itself is not instrumented (i.e. Edebug does not stop before and after the defining form), but forms inside it typically will be instrumented. The `&define` keyword should be the first element in a list specification.

Additional specifications that may only appear after `&define` are described here. See the `defun` example below.

`name` The argument, a symbol, is the name of the defining form. But a defining form need not be named at all, in which case a unique name will be created for it.

The `name` specification may be used more than once in the specification and each subsequent use will append the corresponding symbol argument to the previous name with '`@`' between them. This is useful for generating unique but meaningful names for definitions such as `defadvice` and `defmethod`.

`:name` The element following `:name` should be a symbol; it is used as an additional name component for the definition. This is useful to add a unique, static component to the name of the definition. It may be used more than once. No argument is matched.

`arg` The argument, a symbol, is the name of an argument of the defining form. However, lambda list keywords (symbols starting with '`&`') are not allowed. See `lambda-list` and the example below.

`lambda-list`
This matches the whole argument list of an XEmacs Lisp lambda expression, which is a list of symbols and the keywords `&optional` and `&rest`

`def-body` The argument is the body of code in a definition. This is like `body`, described above, but a definition body must be instrumented with a different Edebug call that looks up information associated with the definition. Use `def-body` for the highest level list of forms within the definition.

`def-form` The argument is a single, highest-level form in a definition. This is like `def-body`, except use this to match a single form rather than a list of forms. As a special case, `def-form` also means that tracing information is not output when the form is executed. See the `interactive` example below.

`nil` This is successful when there are no more arguments to match at the current argument list level; otherwise it fails. See sublist specifications and the backquote example below.

`gate` No argument is matched but backtracking through the gate is disabled while matching the remainder of the specifications at this level. This is primarily used to generate more specific syntax error messages. See Section 16.4.16.2 [Backtracking], page 244 for more details. Also see the `let` example below.

other-symbol

Any other symbol in a specification list may be a predicate or an indirect specification.

If the symbol has an Edebug specification, this *indirect specification* should be either a list specification that is used in place of the symbol, or a function that is called to process the arguments. The specification may be defined with `def-edebug-spec` just as for macros. See the `defun` example below.

Otherwise, the symbol should be a predicate. The predicate is called with the argument and the specification fails if the predicate fails. The argument is not instrumented.

Predicates that may be used include: `symbolp`, `integerp`, `stringp`, `vectorp`, `atom` (which matches a number, string, symbol, or vector), `keywordp`, and `lambda-list-keywordp`. The last two, defined in '`edebug.el`', test whether the argument is a symbol starting with '`:`' and '`&`' respectively.

`[`*elements*`...]`

Rather than matching a vector argument, a vector treats the *elements* as a single *group specification*.

`"`*string*`"` The argument should be a symbol named *string*. This specification is equivalent to the quoted symbol, `'`*symbol*, where the name of *symbol* is the *string*, but the string form is preferred.

`'`*symbol* or `(quote` *symbol*`)`

The argument should be the symbol *symbol*. But use a string specification instead.

`(vector` *elements*`...)`

The argument should be a vector whose elements must match the *elements* in the specification. See the backquote example below.

`(`*elements*`...)`

Any other list is a *sublist specification* and the argument must be a list whose elements match the specification *elements*.

A sublist specification may be a dotted list and the corresponding list argument may then be a dotted list. Alternatively, the last cdr of a dotted list specification may be another sublist specification (via a grouping or an indirect specification, e.g. `(spec . [(more specs...)])`) whose elements match the non-dotted list arguments. This is useful in recursive specifications such as in the backquote example below. Also see the description of a `nil` specification above for terminating such recursion.

Note that a sublist specification of the form `(specs . nil)` means the same as `(specs)`, and `(specs . (sublist-elements...))` means the same as `(specs sublist-elements...)`.

16.4.16.2 Backtracking

If a specification fails to match at some point, this does not necessarily mean a syntax error will be signaled; instead, *backtracking* will take place until all alternatives have been exhausted. Eventually every element of the argument list must be matched by some element in the specification, and every required element in the specification must match some argument.

Backtracking is disabled for the remainder of a sublist or group when certain conditions occur, described below. Backtracking is reenabled when a new alternative is established by `&optional`, `&rest`, or `&or`. It is also reenabled initially when processing a sublist or group specification or an indirect specification.

You might want to disable backtracking to commit to some alternative so that Edebug can provide a more specific syntax error message. Normally, if no alternative matches, Edebug reports that none matched, but if one alternative is committed to, Edebug can report how it failed to match.

First, backtracking is disabled while matching any of the form specifications (i.e. `form`, `body`, `def-form`, and `def-body`). These specifications will match any form so any error must be in the form itself rather than at a higher level.

Second, backtracking is disabled after successfully matching a quoted symbol or string specification, since this usually indicates a recognized construct. If you have a set of alternative constructs that all begin with the same symbol, you can usually work around this constraint by factoring the symbol out of the alternatives, e.g., `["foo" &or [first case] [second case] ...]`.

Third, backtracking may be explicitly disabled by using the `gate` specification. This is useful when you know that no higher alternatives may apply.

16.4.16.3 Debugging Backquote

Backquote (`` ` ``) is a macro that results in an expression that may or may not be evaluated. It is often used to simplify the definition of a macro to return an expression that is evaluated, but Edebug does not know when this is the case. However, the forms inside unquotes (`,` and `,@`) are evaluated and Edebug instruments them.

Nested backquotes are supported by Edebug, but there is a limit on the support of quotes inside of backquotes. Quoted forms (with `'`) are not normally evaluated, but if the quoted form appears immediately within `,` and `,@` forms, Edebug treats this as a backquoted form at the next higher level (even if there is not a next higher level - this is difficult to fix).

If the backquoted forms happen to be code intended to be evaluated, you can have Edebug instrument them by using ``edebug-` `` instead of the regular `` ` ``. Unquoted forms can always appear inside ``edebug-` `` anywhere a form is normally allowed. But `(, form)` may be used in two other places specially recognized by Edebug: wherever a predicate specification would match, and at the head of a list form in place of a function name or lambda expression. The *form* inside a spliced unquote, `(,@ form)`, will be wrapped, but the unquote form itself will not be wrapped since this would interfere with the splicing.

There is one other complication with using ``edebug-` ``. If the ``edebug-` `` call is in a macro and the macro may be called from code that is also instrumented, and if unquoted forms contain any macro arguments bound to instrumented forms, then you should modify the specification for the macro as follows: the specifications for those arguments must use `def-form` instead of `form`. (This is to reestablish the Edebugging context for those external forms.)

For example, the `for` macro (see section "Problems with Macros" in *XEmacs Lisp Reference Manual*) is shown here but with `edebug-‘` substituted for regular `‘`.

```
(defmacro inc (var)
  (list 'setq var (list '1+ var)))

(defmacro for (var from init to final do &rest body)
  (let ((tempvar (make-symbol "max")))
    (edebug-‘ (let (((, var) (, init))
                    ((, tempvar) (, final)))
                (while (<= (, var) (, tempvar))
                  (, body)
                  (inc (, var)))))))
```

Here is the corresponding modified Edebug specification and some code that calls the macro:

```
(def-edebug-spec for
  (symbolp "from" def-form "to" def-form "do" &rest def-form))

(let ((n 5))
  (for i from n to (* n (+ n 1)) do
    (message "%s" i)))
```

After instrumenting the `for` macro and the macro call, Edebug first steps to the beginning of the macro call, then into the macro body, then through each of the unquoted expressions in the backquote showing the expressions that will be embedded in the backquote form. Then when the macro expansion is evaluated, Edebug will step through the `let` form and each time it gets to an unquoted form, it will jump back to an argument of the macro call to step through that expression. Finally stepping will continue after the macro call. Even more convoluted execution paths may result when using anonymous functions.

When the result of an expression is an instrumented expression, it is difficult to see the expression inside the instrumentation. So you may want to set the option `edebug-unwrap-results` to a non-`nil` value while debugging such expressions, but it would slow Edebug down to always do this.

16.4.16.4 Specification Examples

Here we provide several examples of Edebug specifications to show many of its capabilities.

A `let` special form has a sequence of bindings and a body. Each of the bindings is either a symbol or a sublist with a symbol and optional value. In the specification below, notice the `gate` inside of the sublist to prevent backtracking.

```
(def-edebug-spec let
  ((&rest
    &or symbolp (gate symbolp &optional form))
   body))
```

Edebug uses the following specifications for `defun` and `defmacro` and the associated argument list and `interactive` specifications. It is necessary to handle the expression argument of an interactive form specially since it is actually evaluated outside of the function body.

```
(def-edebug-spec defmacro defun)      ; Indirect ref to defun spec
(def-edebug-spec defun
  (&define name lambda-list
           [&optional stringp]        ; Match the doc string, if present.
           [&optional ("interactive" interactive)]
           def-body))

(def-edebug-spec lambda-list
  (([&rest arg]
    [&optional ["&optional" arg &rest arg]]
    &optional ["&rest" arg]
    )))

(def-edebug-spec interactive
  (&optional &or stringp def-form))    ; Notice: def-form
```

The specification for backquote below illustrates how to match dotted lists and use `nil` to terminate recursion. It also illustrates how components of a vector may be matched. (The actual specification provided by Edebug does not support dotted lists because doing so causes very deep recursion that could fail.)

```
(def-edebug-spec ` (backquote-form))  ;; alias just for clarity

(def-edebug-spec backquote-form
  (&or ([&or "," ",@"] &or ("quote" backquote-form) form)
       (backquote-form . [&or nil backquote-form])
       (vector &rest backquote-form)
       sexp))
```

16.4.17 Edebug Options

These options affect the behavior of Edebug:

edebug-setup-hook User Option

Functions to call before Edebug is used. Each time it is set to a new value, Edebug will call those functions once and then `edebug-setup-hook` is reset to `nil`. You could use this to load up Edebug specifications associated with a package you are using but only when you also use Edebug. See Section 16.4.2 [Instrumenting], page 229.

edebug-all-defs User Option

If non-`nil`, normal evaluation of any defining forms (e.g. `defun` and `defmacro`) will instrument them for Edebug. This applies to `eval-defun`, `eval-region`, and `eval-current-buffer`.

Use the command *`M-x edebug-all-defs`* to toggle the value of this variable. You may want to make this variable local to each buffer by calling `(make-local-variable 'edebug-all-defs)` in your `emacs-lisp-mode-hook`. See Section 16.4.2 [Instrumenting], page 229.

edebug-all-forms User Option

If non-`nil`, normal evaluation of any forms by `eval-defun`, `eval-region`, and `eval-current-buffer` will instrument them for Edebug.

Use the command *M-x edebug-all-forms* to toggle the value of this option. See Section 16.4.2 [Instrumenting], page 229.

edebug-save-windows User Option

If non-`nil`, save and restore window configuration on Edebug calls. It takes some time to do this, so if your program does not care what happens to data about windows, you may want to set this variable to `nil`.

If the value is a list, only the listed windows are saved and restored.

M-x edebug-toggle-save-windows may be used to change this variable. This command is bound to *W* in source code buffers. See Section 16.4.15.2 [Edebug Display Update], page 239.

edebug-save-displayed-buffer-points User Option

If non-`nil`, save and restore point in all displayed buffers. This 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.

This is an expensive operation since it visits each window and therefore each displayed buffer twice for each Edebug activation, so it is best to avoid it if you can. See Section 16.4.15.2 [Edebug Display Update], page 239.

edebug-initial-mode User Option

If this variable is non-`nil`, it specifies the initial execution mode for Edebug when it is first activated. Possible values are `step`, `next`, `go`, `Go-nonstop`, `trace`, `Trace-fast`, `continue`, and `Continue-fast`.

The default value is `step`. See Section 16.4.3 [Edebug Execution Modes], page 230.

edebug-trace User Option

Non-`nil` means display a trace of function entry and exit. Tracing output is displayed in a buffer named '`*edebug-trace*`', one function entry or exit per line, indented by the recursion level.

The default value is `nil`.

Also see `edebug-tracing`. See Section 16.4.13 [Tracing], page 238.

edebug-test-coverage User Option

If non-`nil`, Edebug tests coverage of all expressions debugged. This is done by comparing the result of each expression with the previous result. Coverage is considered OK if two different results are found. So to sufficiently test the coverage of your code, try to execute it under conditions that evaluate all expressions more than once, and produce different results for each expression.

Use *M-x edebug-display-freq-count* to display the frequency count and coverage information for a definition. See Section 16.4.14 [Coverage Testing], page 238.

edebug-continue-kbd-macro User Option

If non-`nil`, continue defining or executing any keyboard macro that is executing outside of Edebug. Use this with caution since it is not debugged. See Section 16.4.3 [Edebug Execution Modes], page 230.

edebug-print-length User Option

If non-`nil`, bind `print-length` to this while printing results in Edebug. The default value is `50`. See Section 16.4.12 [Printing in Edebug], page 237.

edebug-print-level User Option

If non-`nil`, bind `print-level` to this while printing results in Edebug. The default value is `50`.

edebug-print-circle User Option

If non-`nil`, bind `print-circle` to this while printing results in Edebug. The default value is `nil`.

edebug-on-error User Option

`debug-on-error` is bound to this while Edebug is active. See Section 16.4.7 [Trapping Errors], page 234.

edebug-on-quit User Option

`debug-on-quit` is bound to this while Edebug is active. See Section 16.4.7 [Trapping Errors], page 234.

edebug-unwrap-results User Option

Non-`nil` if Edebug should unwrap results of expressions. This is useful when debugging macros where the results of expressions are instrumented expressions. But don't do this when results might be circular or an infinite loop will result. See Section 16.4.16.3 [Debugging Backquote], page 245.

edebug-global-break-condition User Option

If non-`nil`, an expression to test for at every stop point. If the result is non-nil, then break. Errors are ignored. See Section 16.4.6.1 [Global Break Condition], page 233.

17 Reading and Printing Lisp Objects

Printing and *reading* are the operations of converting Lisp objects to textual form and vice versa. They use the printed representations and read syntax described in Chapter 2 [Lisp Data Types], page 15.

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).

17.1 Introduction to Reading and Printing

Reading a Lisp object means parsing a Lisp expression in textual form and producing a corresponding Lisp object. This is how Lisp programs get into Lisp from files of Lisp code. We call the text the *read syntax* of the object. For example, the text '`(a . 5)`' is the read syntax for a cons cell whose CAR is `a` and whose CDR is the number 5.

Printing a Lisp object means producing text that represents that object—converting the object to its printed representation. Printing the cons cell described above produces the text '`(a . 5)`'.

Reading and printing are more or less inverse operations: printing the object that results from reading a given piece of text often produces the same text, and reading the text that results from printing an object usually produces a similar-looking object. For example, printing the symbol `foo` produces the text '`foo`', and reading that text returns the symbol `foo`. Printing a list whose elements are `a` and `b` produces the text '`(a b)`', and reading that text produces a list (but not the same list) with elements `a` and `b`.

However, these two operations are not precisely inverses. There are three kinds of exceptions:

- Printing can produce text that cannot be read. For example, buffers, windows, frames, subprocesses and markers print 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.
- Comments can appear at certain points in the middle of an object's read sequence without affecting the result of reading it.

17.2 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 wants 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` supplied as an input stream means to use the value of `standard-input` instead; that value is the *default input stream*, and must be a non-`nil` input stream.

symbol A symbol as input stream is equivalent to the symbol's function definition (if any).

Here is an example of reading from a stream that is a buffer, showing where point is located before and after:

```
---------- Buffer: foo ----------
This⋆ is the contents of foo.
---------- Buffer: foo ----------

(read (get-buffer "foo"))
     ⇒ is
(read (get-buffer "foo"))
     ⇒ the

---------- Buffer: foo ----------
This is the⋆ contents of foo.
---------- Buffer: foo ----------
```

Note that the first read skips a space. Reading skips any amount of whitespace preceding the significant text.

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 open-parenthesis 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, initially positioned at the beginning of the buffer shown. The value read is the symbol `This`.

```
---------- Buffer: foo ----------
This is the contents of foo.
---------- Buffer: foo ----------
```

```
(setq m (set-marker (make-marker) 1 (get-buffer "foo")))
     ⇒ #<marker at 1 in foo>
(read m)
     ⇒ This
m
     ⇒ #<marker at 5 in foo>   ;; Before the first space.
```

Here we read from the contents of a string:

```
(read "(When in) the course")
     ⇒ (When in)
```

The following example reads from the minibuffer. The prompt is: '`Lisp expression: `'. (That is always the prompt used when you read from the stream `t`.) The user's input is shown following the prompt.

```
(read t)
     ⇒ 23
---------- Buffer: Minibuffer ----------
Lisp expression: 23 RET
---------- Buffer: Minibuffer ----------
```

Finally, here is an example of a stream that is a function, named `useless-stream`. Before we use the stream, we initialize the variable `useless-list` to a list of characters. Then each call to the function `useless-stream` obtains the next character in the list or unreads a character by adding it to the front of the list.

```
(setq useless-list (append "XY()" nil))
     ⇒ (88 89 40 41)

(defun useless-stream (&optional unread)
  (if unread
      (setq useless-list (cons unread useless-list))
    (prog1 (car useless-list)
           (setq useless-list (cdr useless-list)))))
     ⇒ useless-stream
```

Now we read using the stream thus constructed:

```
(read 'useless-stream)
     ⇒ XY

useless-list
     ⇒ (40 41)
```

Note that the open and close parentheses remains in the list. The Lisp reader encountered the open parenthesis, decided that it ended the input, and unread it. Another attempt to read from the stream at this point would read '`()`' and return `nil`.

17.3 Input Functions

This section describes the Lisp functions and variables that pertain to reading.

In the functions below, *stream* stands for an input stream (see the previous section). If *stream* is `nil` or omitted, it defaults to the value of `standard-input`.

An `end-of-file` error is signaled if reading encounters an unterminated list, vector, or string.

read &optional *stream* — Function

This function reads one textual Lisp expression from *stream*, returning it as a Lisp object. This is the basic Lisp input function.

read-from-string *string* &optional *start end* — Function

This function reads the first textual Lisp expression from the text in *string*. It returns a cons cell whose CAR is that expression, and whose CDR is an integer giving the position of the next remaining character in the string (i.e., the first one not read).

If *start* is supplied, then reading begins at index *start* in the string (where the first character is at index 0). If *end* is also supplied, then reading stops just before that index, as if the rest of the string were not there.

For example:

```
(read-from-string "(setq x 55) (setq y 5)")
     ⇒ ((setq x 55) . 11)
(read-from-string "\"A short string\"")
     ⇒ ("A short string" . 16)

;; Read starting at the first character.
(read-from-string "(list 112)" 0)
     ⇒ ((list 112) . 10)
;; Read starting at the second character.
(read-from-string "(list 112)" 1)
     ⇒ (list . 5)
;; Read starting at the seventh character,
;;    and stopping at the ninth.
(read-from-string "(list 112)" 6 8)
     ⇒ (11 . 8)
```

standard-input — Variable

This variable holds the default input stream—the stream that `read` uses when the *stream* argument is `nil`.

17.4 Output Streams

An output stream specifies what to do with the characters produced by printing. Most print functions accept an output stream as an optional argument. Here are the possible types of output stream:

buffer The output characters are inserted into *buffer* at point. Point advances as characters are inserted.

marker The output characters are inserted into the buffer that *marker* points into, at the marker position. The marker position advances as characters are inserted. The value of point in the buffer has no effect on printing when the stream is a marker.

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 to the value of `standard-output` instead; 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).

Many of the valid output streams are also valid as input streams. The difference between input and output streams is therefore mostly one of how you use a Lisp object, not a distinction of types of object.

Here is an example of a buffer used as an output stream. Point is initially located as shown immediately before the '`h`' in '`the`'. At the end, point is located directly before that same '`h`'.

```
---------- Buffer: foo ----------
This is t⋆he contents of foo.
---------- Buffer: foo ----------

(print "This is the output" (get-buffer "foo"))
     ⇒ "This is the output"

---------- Buffer: foo ----------
This is t
"This is the output"
⋆he contents of foo.
---------- Buffer: foo ----------
```

Now we show a use of a marker as an output stream. Initially, the marker is in buffer `foo`, between the '`t`' and the '`h`' in the word '`the`'. At the end, the marker has advanced over the inserted text so that it remains positioned before the same '`h`'. Note that the location of point, shown in the usual fashion, has no effect.

```
---------- Buffer: foo ----------
"This is the ⋆output"
---------- Buffer: foo ----------

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 the use of a function as an output stream. The function `eat-output` takes each character that it is given and conses it onto the front of the list `last-output` (see Section 5.5 [Building Lists], page 82). 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
     ⇒ (?\n ?\" ?t ?u ?p ?t ?u ?o ?\  ?e ?h ?t
                ?\  ?s ?i ?\  ?s ?i ?h ?T ?\" ?\n)
```

Now we can put the output in the proper order by reversing the list:

```
(concat (nreverse last-output))
     ⇒ "
\"This is the output\"
"
```

Calling `concat` converts the list to a string so you can see its contents more clearly.

17.5 Output Functions

This section describes the Lisp functions for printing Lisp objects.

Some of the XEmacs printing functions add quoting characters to the output when necessary so that it can be read properly. The quoting characters used are '"' and '\'; they distinguish strings from symbols, and prevent punctuation characters in strings and symbols from being taken as delimiters when reading. See Section 2.1 [Printed Representation], page 15, 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. XEmacs detects such recursion and prints '#*level*' instead of recursively printing an object already being printed. For example, here '#0' indicates a recursive reference to the object at level 0 of the current print operation:

```
(setq foo (list nil))
     ⇒ (nil)
(setcar foo foo)
     ⇒ (#0)
```

In the functions below, *stream* stands for an output stream. (See the previous section for a description of output streams.) If *stream* is `nil` or omitted, it defaults to the value of `standard-output`.

print *object* &optional *stream* Function

The `print` function is a convenient way of printing. It outputs the printed representation of *object* to *stream*, printing in addition one newline before *object* and another after it. Quoting characters are used. `print` returns *object*. For example:

```
(progn (print 'The\ cat\ in)
       (print "the hat")
       (print " came back"))
     ⊣
     ⊣ The\ cat\ in
     ⊣
     ⊣ "the hat"
     ⊣
     ⊣ " came back"
     ⊣
     ⇒ " came back"
```

prin1 *object* &optional *stream* Function

This function outputs the printed representation of *object* to *stream*. It does not print newlines to separate output as `print` does, but it does use quoting characters just like `print`. It returns *object*.

```
(progn (prin1 'The\ cat\ in)
       (prin1 "the hat")
       (prin1 " came back"))
     ⊣ The\ cat\ in"the hat"" came back"
     ⇒ " came back"
```

princ *object* &optional *stream* Function

This function outputs the printed representation of *object* to *stream*. It returns *object*.

This function is intended to produce output that is readable by people, not by `read`, so it doesn't insert quoting characters and doesn't put double-quotes around the contents of strings. It does not add any spacing between calls.

```
(progn
  (princ 'The\ cat)
  (princ " in the \"hat\""))
     ⊣ The cat in the "hat"
     ⇒ " in the \"hat\""
```

terpri &optional *stream* Function

This function outputs a newline to *stream*. The name stands for "terminate print".

write-char *character* &optional *stream* Function

This function outputs *character* to *stream*. It returns *character*.

prin1-to-string *object* &optional *noescape* Function

This function returns a string containing the text that `prin1` would have printed for the same argument.

```
(prin1-to-string 'foo)
     ⇒ "foo"
(prin1-to-string (mark-marker))
     ⇒ "#<marker at 2773 in strings.texi>"
```

If *noescape* is non-`nil`, that inhibits use of quoting characters in the output. (This argument is supported in Emacs versions 19 and later.)

```
(prin1-to-string "foo")
     ⇒ "\"foo\""
(prin1-to-string "foo" t)
     ⇒ "foo"
```

See `format`, in Section 4.7 [String Conversion], page 65, for other ways to obtain the printed representation of a Lisp object as a string.

17.6 Variables Affecting Output

standard-output Variable

The value of this variable is the default output stream—the stream that print functions use when the *stream* argument is `nil`.

print-escape-newlines Variable

If this variable is non-`nil`, then newline characters in strings are printed as '`\n`' and formfeeds are printed as '`\f`'. Normally these characters are printed as actual newlines and formfeeds.

This variable affects the print functions `prin1` and `print`, 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-readably Variable

If non-`nil`, then all objects will be printed in a readable form. If an object has no readable representation, then an error is signalled. When `print-readably` is true, compiled-function objects will be written in '`#[...]`' form instead of in '`#<compiled-function [...]>`' form, and two-element lists of the form '`(quote object)`' will be written as the equivalent '`'object`'. Do not *set* this variable; bind it instead.

print-length Variable

The value of this variable is the maximum number of elements of a list that will be printed. If a list being printed has more than this many elements, it is abbreviated with an ellipsis.

If the value is `nil` (the default), then there is no limit.

```
(setq print-length 2)
     ⇒ 2
(print '(1 2 3 4 5))
     ⊣ (1 2 ...)
     ⇒ (1 2 ...)
```

print-level Variable

The value of this variable is the maximum depth of nesting of parentheses and brackets when printed. Any list or vector at a depth exceeding this limit is abbreviated with an ellipsis. A value of `nil` (which is the default) means no limit.

This variable exists in version 19 and later versions.

print-string-length Variable

The value of this variable is the maximum number of characters of a string that will be printed. If a string being printed has more than this many characters, it is abbreviated with an ellipsis.

print-gensym Variable

If non-`nil`, then uninterned symbols will be printed specially. Uninterned symbols are those which are not present in `obarray`, that is, those which were made with `make-symbol` or by calling `intern` with a second argument.

When `print-gensym` is true, such symbols will be preceded by '`#:`', which causes the reader to create a new symbol instead of interning and returning an existing one. Beware: The '`#:`' syntax creates a new symbol each time it is seen, so if you print an object which contains two pointers to the same uninterned symbol, `read` will not duplicate that structure.

Also, since XEmacs has no real notion of packages, there is no way for the printer to distinguish between symbols interned in no obarray, and symbols interned in an alternate obarray.

float-output-format Variable

This variable holds the format descriptor string that Lisp uses to print floats. This is a '`%`'-spec like those accepted by `printf` in C, but with some restrictions.

It must start with the two characters '`%.`'. After that comes an integer precision specification, and then a letter which controls the format. The letters allowed are '`e`', '`f`' and '`g`'.

- Use '`e`' for exponential notation '*dig*`.`*digits*`e`*expt*'.
- Use '`f`' for decimal point notation '`DIGITS.DIGITS`'.
- Use '`g`' to choose the shorter of those two formats for the number at hand.

The precision in any of these cases is the number of digits following the decimal point. With '`f`', a precision of 0 means to omit the decimal point. 0 is not allowed with '`f`' or '`g`'.

A value of nil means to use '`%.16g`'.

Regardless of the value of `float-output-format`, a floating point number will never be printed in such a way that it is ambiguous with an integer; that is, a floating-point number will always be printed with a decimal point and/or an exponent, even if the digits following the decimal point are all zero. This is to preserve read-equivalence.

18 Minibuffers

A *minibuffer* is a special buffer that XEmacs commands use to read arguments more complicated than the single numeric prefix argument. These arguments include file names, buffer names, and command names (as in `M-x`). The minibuffer is displayed on the bottom line of the frame, in the same place as the echo area, but only while it is in use for reading an argument.

18.1 Introduction to Minibuffers

In most ways, a minibuffer is a normal XEmacs 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 32.7 [Minibuffers and Frames], page 478.)

The minibuffer's window is normally a single line. You can resize it temporarily with the window sizing commands; it reverts to its normal size when the minibuffer is exited. You can resize it permanently by using the window sizing commands in the frame's other window, when the minibuffer is not active. If the frame contains just a minibuffer, you can change the minibuffer's size by changing the frame's size.

If a command uses a minibuffer while there is an active minibuffer, this is called a *recursive minibuffer*. The first minibuffer is named ‘ `*Minibuf-0*`’. Recursive minibuffers are named by incrementing the number at the end of the name. (The names begin with a space so that they won't show up in normal buffer lists.) Of several recursive minibuffers, the innermost (or most recently entered) is the active minibuffer. We usually call this “the” minibuffer. You can permit or forbid recursive minibuffers by setting the variable `enable-recursive-minibuffers`.

Like other buffers, a minibuffer may use any of several local keymaps (see Chapter 20 [Keymaps], page 315); these contain various exit commands and in some cases completion commands (see Section 18.5 [Completion], page 266).

- `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.

18.2 Reading Text Strings with the Minibuffer

Most often, the minibuffer is used to read text as a string. It can also be used to read a Lisp object in textual form. The most basic primitive for minibuffer input is `read-from-minibuffer`; it can do either one.

In most cases, you should not call minibuffer input functions in the middle of a Lisp function. Instead, do all minibuffer input as part of reading the arguments for a command, in the `interactive` spec. See Section 19.2 [Defining Commands], page 282.

read-from-minibuffer *prompt-string* &optional *initial-contents* *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 17.3 [Input Functions], page 253).

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-contents* is a string, `read-from-minibuffer` inserts it into the minibuffer, leaving point at the end. The minibuffer appears with this text as its contents.

The value of *initial-contents* may also be a cons cell of the form `(`*string* `.` *position*`)`. This means to insert *string* in the minibuffer but put point *position* characters from the beginning, rather than at the end.

If *keymap* is non-`nil`, that keymap is the local keymap to use in the minibuffer. If *keymap* is omitted or `nil`, the value of `minibuffer-local-map` is used as the keymap. Specifying a keymap is the most important way to customize the minibuffer for various applications such as completion.

The argument *hist* specifies which history list variable to use for saving the input and for history commands used in the minibuffer. It defaults to `minibuffer-history`. See Section 18.4 [Minibuffer History], page 264.

When the user types a command to exit the minibuffer, `read-from-minibuffer` uses the text in the minibuffer to produce its return value. Normally it simply makes a string containing that text. However, if *read* is non-`nil`, `read-from-minibuffer` reads the text and returns the resulting Lisp object, unevaluated. (See Section 17.3 [Input Functions], page 253, for information about reading.)

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`. The keymap used is `minibuffer-local-map`.

This is a simplified interface to the `read-from-minibuffer` function:

```
(read-string prompt initial)
≡
(read-from-minibuffer prompt initial nil nil nil)
```

minibuffer-local-map — Variable

This is the default local keymap for reading from the minibuffer. By default, it makes the following bindings:

⟨LFD⟩ `exit-minibuffer`

⟨RET⟩ `exit-minibuffer`

C-g `abort-recursive-edit`

M-n `next-history-element`

M-p `previous-history-element`

M-r `next-matching-history-element`

M-s `previous-matching-history-element`

read-no-blanks-input *prompt* &optional *initial* 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, in addition to those of `minibuffer-local-map`:

⟨SPC⟩ `exit-minibuffer`

⟨TAB⟩ `exit-minibuffer`

? `self-insert-and-exit`

18.3 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`.

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 17.5 [Output Functions], page 256.

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 evaluation of the preceding expression,
;;    the following appears in the minibuffer:

---------- Buffer: Minibuffer ----------
Please edit: (forward-word 1)⋆
---------- Buffer: Minibuffer ----------
```

Typing ⟨RET⟩ right away would exit the minibuffer and evaluate the expression, thus moving point forward one word. `edit-and-eval-command` returns `t` in this example.

18.4 Minibuffer History

A *minibuffer history list* records previous minibuffer inputs so the user can reuse them conveniently. A history list is actually a symbol, not a list; it is a variable whose value is a list of strings (previous inputs), most recent first.

There are many separate history lists, used for different kinds of inputs. It's the Lisp programmer's job to specify the right history list for each use of the minibuffer.

The basic minibuffer input functions `read-from-minibuffer` and `completing-read` both accept an optional argument named *hist* which is how you specify the history list. Here are the possible values:

variable Use *variable* (a symbol) as the history list.

(*variable* . *startpos*)
: Use *variable* (a symbol) as the history list, and assume that the initial history position is *startpos* (an integer, counting from zero which specifies the most recent element of the history).

  If you specify *startpos*, then you should also specify that element of the history as the initial minibuffer contents, for consistency.

If you don't specify *hist*, then the default history list `minibuffer-history` is used. For other standard history lists, see below. You can also create your own history list variable; just initialize it to `nil` before the first use.

Both `read-from-minibuffer` and `completing-read` add new elements to the history list automatically, and provide commands to allow the user to reuse items on the list. The only thing your program needs to do to use a history list is to initialize it and to pass its name to the input functions when you wish. But it is safe to modify the list by hand when the minibuffer input functions are not using it.

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.

regexp-history Variable
: A history list for regular expression arguments.

extended-command-history Variable
: A history list for arguments that are names of extended commands.

shell-command-history Variable
: A history list for arguments that are shell commands.

read-expression-history Variable
: A history list for arguments that are Lisp expressions to evaluate.

Info-minibuffer-history Variable
: A history list for Info mode's minibuffer.

Manual-page-minibuffer-history Variable
: A history list for `manual-entry`.

There are many other minibuffer history lists, defined by various libraries. An *M-x apropos* search for '`history`' should prove fruitful in discovering them.

18.5 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 XEmacs 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.

18.5.1 Basic Completion Functions

The two functions `try-completion` and `all-completions` have nothing in themselves to do with minibuffers. We describe them in this chapter so as to keep them near the higher-level completion features that do use the minibuffer.

try-completion *string collection* &optional *predicate* — Function

This function returns the longest common substring of all possible completions of *string* in *collection*. The value of *collection* must be an alist, an obarray, or a function that implements a virtual set of strings (see below).

Completion compares *string* against each of the permissible completions specified by *collection*; if the beginning of the permissible completion equals *string*, it matches. If no permissible completions match, `try-completion` returns `nil`. If only one permissible completion matches, and the match is exact, then `try-completion` returns `t`. Otherwise, the value is the longest initial sequence common to all the permissible completions that match.

If *collection* is an alist (see Section 5.8 [Association Lists], page 92), the CARs of the alist elements form the set of permissible completions.

If *collection* is an obarray (see Section 7.3 [Creating Symbols], page 113), the names of all symbols in the obarray form the set of permissible completions. The global variable `obarray` holds an obarray containing the names of all interned Lisp symbols.

Note that the only valid way to make a new obarray is to create it empty and then add symbols to it one by one using `intern`. Also, you cannot intern a given symbol in more than one obarray.

If the argument *predicate* is non-`nil`, then it must be a function of one argument. It is used to test each possible match, and the match is accepted only if *predicate* returns non-`nil`. The argument given to *predicate* is either a cons cell from the alist (the CAR of which is a string) or else it is a symbol (*not* a symbol name) from the obarray.

You can also use a symbol that is a function as *collection*. Then the function is solely responsible for performing completion; `try-completion` returns whatever this function returns. The function is called with three arguments: *string*, *predicate* and `nil`. (The reason for the third argument is so that the same function can be used in `all-completions` and do the appropriate thing in either case.) See Section 18.5.6 [Programmed Completion], page 274.

In the first of the following examples, the string '`foo`' is matched by three of the alist CARs. All of the matches begin with the characters '`fooba`', so that is the result. In the second example, there is only one possible match, and it is exact, so the value is `t`.

```
(try-completion
 "foo"
 '(("foobar1" 1) ("barfoo" 2) ("foobaz" 3) ("foobar2" 4)))
     ⇒ "fooba"

(try-completion "foo" '(("barfoo" 2) ("foo" 3)))
     ⇒ t
```

In the following example, numerous symbols begin with the characters '`forw`', and all of them begin with the word '`forward`'. In most of the symbols, this is followed with a '`-`', but not in all, so no more than '`forward`' can be completed.

```
(try-completion "forw" obarray)
     ⇒ "forward"
```

Finally, in the following example, only two of the three possible matches pass the predicate `test` (the string '`foobaz`' is too short). Both of those begin with the string '`foobar`'.

```
(defun test (s)
  (> (length (car s)) 6))
     ⇒ test
(try-completion
 "foo"
 '(("foobar1" 1) ("barfoo" 2) ("foobaz" 3) ("foobar2" 4))
 'test)
     ⇒ "foobar"
```

all-completions *string collection* &optional *predicate nospace* Function

This function returns a list of all possible completions of *string*. The 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`; then `all-completions` returns whatever the function returns. See Section 18.5.6 [Programmed Completion], page 274.

If *nospace* is non-`nil`, completions that start with a space are ignored unless *string* also starts with a space.

Here is an example, using the function `test` shown in the example for `try-completion`:

```
(defun test (s)
  (> (length (car s)) 6))
     ⇒ test

(all-completions
 "foo"
 '(("foobar1" 1) ("barfoo" 2) ("foobaz" 3) ("foobar2" 4))
 'test)
     ⇒ ("foobar1" "foobar2")
```

completion-ignore-case Variable

If the value of this variable is non-`nil`, XEmacs does not consider case significant in completion.

18.5.2 Completion and the Minibuffer

This section describes the basic interface for reading from the minibuffer with completion.

completing-read *prompt collection* &optional *predicate require-match initial hist* 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 usual minibuffer exit commands won't exit unless the input completes to an element of *collection*. If *require-match* is neither `nil` nor `t`, then the exit commands won't exit unless the input typed is itself an element of *collection*. If *require-match* is `nil`, the exit commands work regardless of the input in the minibuffer.

The user can exit with null input by typing ⟨RET⟩ with an empty minibuffer. Then `completing-read` returns `nil`. This is how the user requests whatever default the command uses for the value being read. The user can return using ⟨RET⟩ in this way regardless of the value of *require-match*.

The function `completing-read` works by calling `read-minibuffer`. It uses `minibuffer-local-completion-map` as the keymap if *require-match* is `nil`, and uses `minibuffer-local-must-match-map` if *require-match* is non-`nil`. See Section 18.5.3 [Completion Commands], page 269.

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 18.4 [Minibuffer History], page 264.

Completion ignores case when comparing the input against the possible matches, if the built-in variable `completion-ignore-case` is non-`nil`. See Section 18.5.1 [Basic Completion], page 266.

Here's an example of using `completing-read`:

```
(completing-read
 "Complete a foo: "
 '(("foobar1" 1) ("barfoo" 2) ("foobaz" 3) ("foobar2" 4))
 nil t "fo")

;; After evaluation of the preceding expression,
;;    the following appears in the minibuffer:

---------- Buffer: Minibuffer ----------
Complete a foo: fo⋆
---------- Buffer: Minibuffer ----------
```

If the user then types ⟨DEL⟩ ⟨DEL⟩ *b* ⟨RET⟩, `completing-read` returns `barfoo`.

The `completing-read` function binds three variables to pass information to the commands that actually do completion. These variables are `minibuffer-completion-table`, `minibuffer-completion-predicate` and `minibuffer-completion-confirm`. For more information about them, see Section 18.5.3 [Completion Commands], page 269.

18.5.3 Minibuffer Commands That Do Completion

This section describes the keymaps, commands and user options used in the minibuffer to do completion.

minibuffer-local-completion-map Variable

`completing-read` uses this value as the local keymap when an exact match of one of the completions is not required. By default, this keymap makes the following bindings:

?	`minibuffer-completion-help`
⟨SPC⟩	`minibuffer-complete-word`
⟨TAB⟩	`minibuffer-complete`

with other characters bound as in `minibuffer-local-map` (see Section 18.2 [Text from Minibuffer], page 261).

minibuffer-local-must-match-map Variable

`completing-read` uses this value as the local keymap when an exact match of one of the completions is required. Therefore, no keys are bound to `exit-minibuffer`, the command that exits the minibuffer unconditionally. By default, this keymap makes the following bindings:

?	`minibuffer-completion-help`
⟨SPC⟩	`minibuffer-complete-word`
⟨TAB⟩	`minibuffer-complete`
⟨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 minibuffer completion commands such as `minibuffer-complete-word`.

minibuffer-completion-predicate Variable

This variable's value is the predicate that `completing-read` passes to `try-completion`. The variable is also used by the other minibuffer completion functions.

minibuffer-complete-word Command

This function completes the minibuffer contents by at most a single word. Even if the minibuffer contents have only one completion, `minibuffer-complete-word` does not add any characters beyond the first character that is not a word constituent. See Chapter 38 [Syntax Tables], page 569.

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—the command is programmed to work without confirmation when run twice in succession.

minibuffer-completion-confirm Variable

When the value of this variable is non-`nil`, XEmacs 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 17 [Read and Print], page 251, 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.

18.5.4 High-Level Completion Functions

This section describes the higher-level convenient functions for reading certain sorts of names with completion.

In most cases, you should not call these functions in the middle of a Lisp function. When possible, do all minibuffer input as part of reading the arguments for a command, in the `interactive` spec. See Section 19.2 [Defining Commands], page 282.

read-buffer *prompt* &optional *default existing* Function

This function reads the name of a buffer and returns it as a string. The argument *default* is the default name to use, the value to return if the user exits with an empty minibuffer. If non-`nil`, it should be a string or a buffer. It is mentioned in the prompt, but is not inserted in the minibuffer as initial input.

If *existing* is non-`nil`, then the name specified must be that of an existing buffer. The usual commands to exit the minibuffer do not exit if the text is not valid, and ⟨RET⟩ does completion to attempt to find a valid name. (However, *default* is not checked for validity; it is returned, whatever it is, if the user exits with the minibuffer empty.)

In the following example, the user enters '`minibuffer.t`', and then types ⟨RET⟩. The argument *existing* is `t`, and the only buffer name starting with the given input is '`minibuffer.texi`', so that name is the value.

```
(read-buffer "Buffer name? " "foo" t)
;; After evaluation of the preceding expression,
;;    the following prompt appears,
;;    with an empty minibuffer:

---------- Buffer: Minibuffer ----------
Buffer name? (default foo) ⋆
---------- Buffer: Minibuffer ----------
```

```
;; The user types minibuffer.t RET.
     ⇒ "minibuffer.texi"
```

read-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 19.3 [Interactive Call], page 286.

```
(read-command "Command name? ")

;; After evaluation of the preceding expression,
;;    the following prompt appears with an empty minibuffer:

---------- Buffer: Minibuffer ----------
Command name?
---------- Buffer: Minibuffer ----------
```

If the user types *forward-c* ⟨RET⟩, then this function returns `forward-char`.

The `read-command` function is a simplified interface to the function `completing-read`. It uses the variable `obarray` so as to complete in the set of extant Lisp symbols, and it uses the `commandp` predicate so as to accept only command names:

```
(read-command prompt)
≡
(intern (completing-read prompt obarray
                         'commandp t nil))
```

read-variable *prompt* Function

This function reads the name of a user variable and returns it as a symbol.

```
(read-variable "Variable name? ")

;; After evaluation of the preceding expression,
;;    the following prompt appears,
;;    with an empty minibuffer:

---------- Buffer: Minibuffer ----------
Variable name? ⋆
---------- Buffer: Minibuffer ----------
```

If the user then types *fill-p* ⟨RET⟩, `read-variable` returns `fill-prefix`.

This function is similar to `read-command`, but uses the predicate `user-variable-p` instead of `commandp`:

```
(read-variable prompt)
≡
(intern
 (completing-read prompt obarray
                  'user-variable-p t nil))
```

18.5.5 Reading File Names

Here is another high-level completion function, designed for reading a file name. It provides special features including automatic insertion of the default directory.

read-file-name *prompt* &optional *directory default existing initial* Function

This function reads a file name in the minibuffer, prompting with *prompt* and providing completion. If *default* is non-`nil`, then the function returns *default* if the user just types ⟨RET⟩. *default* is not checked for validity; it is returned, whatever it is, if the user exits with the minibuffer empty.

If *existing* is non-`nil`, then the user must specify the name of an existing file; ⟨RET⟩ performs completion to make the name valid if possible, and then refuses to exit if it is not valid. If the value of *existing* is neither `nil` nor `t`, then ⟨RET⟩ also requires confirmation after completion. If *existing* is `nil`, then the name of a nonexistent file is acceptable.

The argument *directory* specifies the directory to use for completion of relative file names. If `insert-default-directory` is non-`nil`, *directory* is also inserted in the minibuffer as initial input. It defaults to the current buffer's value of `default-directory`.

If you specify *initial*, that is an initial file name to insert in the buffer (after with *directory*, if that is inserted). In this case, point goes at the beginning of *initial*. The default for *initial* is `nil`—don't insert any file name. To see what *initial* does, try the command `C-x C-v`.

Here is an example:

```
(read-file-name "The file is ")

;; After evaluation of the preceding expression,
;;    the following appears in the minibuffer:

---------- Buffer: Minibuffer ----------
The file is /gp/gnu/elisp/⋆
---------- Buffer: Minibuffer ----------
```

Typing `manual` ⟨TAB⟩ results in the following:

```
---------- Buffer: Minibuffer ----------
The file is /gp/gnu/elisp/manual.texi⋆
---------- Buffer: Minibuffer ----------
```

If the user types ⟨RET⟩, `read-file-name` returns the file name as the string `"/gp/gnu/elisp/manual.texi"`.

insert-default-directory User Option

This variable is used by `read-file-name`. Its value controls whether `read-file-name` starts by placing the name of the default directory in the minibuffer, plus the initial file name if any. If the value of this variable is `nil`, then `read-file-name` does not place any initial input in the minibuffer (unless you specify initial input with the *initial* argument). In that case, the default directory is still used for completion of relative file names, but is not displayed.

For example:

```
;; Here the minibuffer starts out with the default directory.
(let ((insert-default-directory t))
  (read-file-name "The file is "))

---------- Buffer: Minibuffer ----------
The file is ~lewis/manual/⋆
---------- Buffer: Minibuffer ----------

;; Here the minibuffer is empty and only the prompt
;;    appears on its line.
(let ((insert-default-directory nil))
  (read-file-name "The file is "))

---------- Buffer: Minibuffer ----------
The file is ⋆
---------- Buffer: Minibuffer ----------
```

18.5.6 Programmed Completion

Sometimes it is not possible to create an alist or an obarray containing all the intended possible completions. In such a case, you can supply your own function to compute the completion of a given string. This is called *programmed completion*.

To use this feature, pass a symbol with a function definition as the *collection* argument to `completing-read`. The function `completing-read` arranges to pass your completion function along to `try-completion` and `all-completions`, which will then let your function do all the work.

The completion function should accept three arguments:

- The string to be completed.
- The predicate function to filter possible matches, or `nil` if none. Your function should call the predicate for each possible match, and ignore the possible match if the predicate returns `nil`.
- A flag specifying the type of operation.

There are three flag values for three operations:

- `nil` specifies `try-completion`. The completion function should return the completion of the specified string, or `t` if the string is 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 that are functions) as well as function symbols as *collection*, but this is impossible. Lists as completion tables are already assigned another meaning—as alists. It would be unreliable to fail to handle an alist normally because it is also a possible function. So you must arrange for any function you wish to use for completion to be encapsulated in a symbol.

Emacs uses programmed completion when completing file names. See Section 28.8.6 [File Name Completion], page 411.

18.6 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. Variations of these functions can be used to ask a yes-or-no question using a dialog box, or optionally using one.

If either of these functions is called in a command that was invoked using the mouse, then it uses a dialog box or pop-up menu to ask the question. Otherwise, it uses keyboard input.

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 45.3 [The Echo Area], page 648), which uses the same screen space as the minibuffer. The cursor moves to the echo area while the question is being asked.

The answers and their meanings, even '`y`' and '`n`', are not hardwired. The keymap `query-replace-map` specifies them. See Section 37.5 [Search and Replace], page 560.

In the following example, the user first types *q*, which is invalid. At the next prompt the user types *y*.

```
(y-or-n-p "Do you need a lift? ")

;; After evaluation of the preceding expression,
;;     the following prompt appears in the echo area:

---------- Echo area ----------
Do you need a lift? (y or n)
---------- Echo area ----------

;; If the user then types q, the following appears:
```

```
---------- Echo area ----------
Please answer y or n.  Do you need a lift? (y or n)
---------- Echo area ----------

;; When the user types a valid answer,
;;    it is displayed after the question:

---------- Echo area ----------
Do you need a lift? (y or n) y
---------- Echo area ----------
```

We show successive lines of echo area messages, but only one actually appears on the screen at a time.

yes-or-no-p *prompt* Function

This function asks the user a question, expecting input in the minibuffer. It returns `t` if the user enters '`yes`', `nil` if the user types '`no`'. The user must type ⟨RET⟩ to finalize the response. Upper and lower case are equivalent.

`yes-or-no-p` starts by displaying *prompt* in the echo area, followed by '`(yes or no) `'. The user must type one of the expected responses; otherwise, the function responds '`Please answer yes or no.`', waits about two seconds and repeats the request.

`yes-or-no-p` requires more work from the user than `y-or-n-p` and is appropriate for more crucial decisions.

Here is an example:

```
(yes-or-no-p "Do you really want to remove everything? ")

;; After evaluation of the preceding expression,
;;    the following prompt appears,
;;    with an empty minibuffer:

---------- Buffer: minibuffer ----------
Do you really want to remove everything? (yes or no)
---------- Buffer: minibuffer ----------
```

If the user first types `y` ⟨RET⟩, which is invalid because this function demands the entire word '`yes`', it responds by displaying these prompts, with a brief pause between them:

```
---------- Buffer: minibuffer ----------
Please answer yes or no.
Do you really want to remove everything? (yes or no)
---------- Buffer: minibuffer ----------
```

yes-or-no-p-dialog-box *prompt* Function

This function asks the user a "y or n" question with a popup dialog box. It returns `t` if the answer is "yes". *prompt* is the string to display to ask the question.

The following functions ask a question either in the minibuffer or a dialog box, depending on whether the last user event (which presumably invoked this command) was a keyboard or mouse event. When XEmacs is running on a window system, the functions `y-or-n-p` and `yes-or-no-p` are replaced with the following functions, so that menu items bring up dialog boxes instead of minibuffer questions.

y-or-n-p-maybe-dialog-box *prompt* Function

This function asks user a "y or n" question, using either a dialog box or the minibuffer, as appropriate.

yes-or-no-p-maybe-dialog-box *prompt* Function

This function asks user a "yes or no" question, using either a dialog box or the minibuffer, as appropriate.

18.7 Asking Multiple Y-or-N Questions

When you have a series of similar questions to ask, such as "Do you want to save this buffer" for each buffer in turn, you should use `map-y-or-n-p` to ask the collection of questions, rather than asking each question individually. This gives the user certain convenient facilities such as the ability to answer the whole series at once.

map-y-or-n-p *prompter actor list* &optional *help action-alist* Function

This function, 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 the objects to ask questions about. It should be either a list of objects or a generator function. If it is a function, it should expect no arguments, and should return either the next object to ask about, or `nil` meaning stop asking questions.

The argument *prompter* specifies how to ask each question. If *prompter* is a string, the question text is computed like this:

`(format` *prompter object*`)`

where *object* is the next object to ask about (as obtained from *list*).

If not a string, *prompter* should be a function of one argument (the next object to ask about) and should return the question text. If the value is a string, that is the question to ask the user. The function can also return `t` meaning do act on this object (and don't ask the user), or `nil` meaning ignore this object (and don't ask the user).

The argument *actor* says how to act on the answers that the user gives. It should be a function of one argument, and it is called with each object that the user says yes for. Its argument is always an object obtained from *list*.

If the argument *help* is given, it should be a list of this form:

`(`*singular plural action*`)`

where *singular* is a string containing a singular noun that describes the objects conceptually being acted on, *plural* is the corresponding plural noun, and *action* is a transitive verb describing what *actor* does.

If you don't specify *help*, the default is `("object" "objects" "act on")`.

Each time a question is asked, the user may enter `y`, `Y`, or ⟨SPC⟩ to act on that object; `n`, `N`, or ⟨DEL⟩ to skip that object; `!` to act on all following objects; ⟨ESC⟩ or `q` to exit (skip all following objects); `.` (period) to act on the current object and then exit; or `C-h` to get help. These are the same answers that `query-replace` accepts. The keymap `query-replace-map` defines their meaning for `map-y-or-n-p` as well as for `query-replace`; see Section 37.5 [Search and Replace], page 560.

You can use *action-alist* to specify additional possible answers and what they mean. It is an alist of elements of the form `(`*char function help*`)`, each of which defines one additional answer. In this element, *char* is a character (the answer); *function* is a function of one argument (an object from *list*); *help* is a string.

When the user responds with *char*, `map-y-or-n-p` calls *function*. If it returns non-`nil`, the object is considered "acted upon", and `map-y-or-n-p` advances to the next object in *list*. If it returns `nil`, the prompt is repeated for the same object.

If `map-y-or-n-p` is called in a command that was invoked using the mouse—more precisely, if `last-nonmenu-event` (see Section 19.4 [Command Loop Info], page 288) is either `nil` or a list—then it uses a dialog box or pop-up menu to ask the question. In this case, it does not use keyboard input or the echo area. You can force use of the mouse or use of keyboard input by binding `last-nonmenu-event` to a suitable value around the call.

The return value of `map-y-or-n-p` is the number of objects acted on.

18.8 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 19.4 [Command Loop Info], page 288).

previous-history-element *n* Command

This command replaces the minibuffer contents with the value of the *n*th previous (older) history element.

next-history-element *n* Command

This command replaces the minibuffer contents with the value of the *n*th more recent history element.

previous-matching-history-element *pattern* Command

This command replaces the minibuffer contents with the value of the previous (older) history element that matches *pattern* (a regular expression).

next-matching-history-element *pattern* Command

This command replaces the minibuffer contents with the value of the next (newer) history element that matches *pattern* (a regular expression).

minibuffer-prompt Function

This function returns the prompt string of the currently active minibuffer. If no minibuffer is active, it returns `nil`.

minibuffer-prompt-width Function

This function returns the display width of the prompt string of the currently active minibuffer. If no minibuffer is active, it returns 0.

minibuffer-setup-hook Variable

This is a normal hook that is run whenever the minibuffer is entered. See Section 26.4 [Hooks], page 378.

minibuffer-exit-hook Variable

This is a normal hook that is run whenever the minibuffer is exited. See Section 26.4 [Hooks], page 378.

minibuffer-help-form Variable

The current value of this variable is used to rebind `help-form` locally inside the minibuffer (see Section 27.5 [Help Functions], page 387).

active-minibuffer-window Function

This function returns the currently active minibuffer window, or `nil` if none is currently active.

minibuffer-window &optional *frame* Function

This function returns the minibuffer window used for frame *frame*. If *frame* is `nil`, that stands for the current frame. Note that the minibuffer window used by a frame need not be part of that frame—a frame that has no minibuffer of its own necessarily uses some other frame's minibuffer window.

window-minibuffer-p *window* Function

This function returns non-`nil` if *window* is a minibuffer window.

It is not correct to determine whether a given window is a minibuffer by comparing it with the result of `(minibuffer-window)`, because there can be more than one minibuffer window if there is more than one frame.

minibuffer-window-active-p *window* Function

This function returns non-`nil` if *window*, assumed to be a minibuffer window, is currently active.

minibuffer-scroll-window Variable

If the value of this variable is non-`nil`, it should be a window object. When the function `scroll-other-window` is called in the minibuffer, it scrolls this window.

Finally, some functions and variables deal with recursive minibuffers (see Section 19.10 [Recursive Editing], page 310):

minibuffer-depth Function

This function returns the current depth of activations of the minibuffer, a non-negative integer. If no minibuffers are active, it returns zero.

enable-recursive-minibuffers User Option

If this variable is non-`nil`, you can invoke commands (such as `find-file`) that use minibuffers even while 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.

In FSF Emacs 19, if a command name has a property `enable-recursive-minibuffers` that is non-`nil`, then the command can use the minibuffer to read arguments even if it is invoked from the minibuffer. The minibuffer command `next-matching-history-element` (normally *`M-s`* in the minibuffer) uses this feature.

This is not implemented in XEmacs because it is a kludge. If you want to explicitly set the value of `enable-recursive-minibuffers` in this fashion, just use an evaluated interactive spec and bind `enable-recursive-minibuffers` while reading from the minibuffer. See the definition of `next-matching-history-element` in '`lisp/prim/minibuf.el`'.

19 Command Loop

When you run XEmacs, it enters the *editor command loop* almost immediately. This loop reads events, 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.

19.1 Command Loop Overview

The command loop in XEmacs is a standard event loop, reading events one at a time with `next-event` and handling them with `dispatch-event`. An event is typically a single user action, such as a keypress, mouse movement, or menu selection; but they can also be notifications from the window system, informing XEmacs that (for example) part of its window was just uncovered and needs to be redrawn. See Section 19.5 [Events], page 290. Pending events are held in a first-in, first-out list called the *event queue*: events are read from the head of the list, and newly arriving events are added to the tail. In this way, events are always processed in the order in which they arrive.

`dispatch-event` does most of the work of handling user actions. The first thing it must do is put the events together into a key sequence, which is a sequence of events that translates into a command. It does this by consulting the active keymaps, which specify what the valid key sequences are and how to translate them into commands. See Section 20.8 [Key Lookup], page 324, for information on how this is done. The result of the translation should be a keyboard macro or an interactively callable function. If the key is *M-x*, then it reads the name of another command, which it then calls. This is done by the command `execute-extended-command` (see Section 19.3 [Interactive Call], page 286).

To execute a command requires first reading the arguments for it. This is done by calling `command-execute` (see Section 19.3 [Interactive Call], page 286). For commands written in Lisp, the `interactive` specification says how to read the arguments. This may use the prefix argument (see Section 19.9 [Prefix Command Arguments], page 308) or may read with prompting in the minibuffer (see Chapter 18 [Minibuffers], page 261). 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 19.13 [Keyboard Macros], page 313).

To terminate the execution of a running command, type *C-g*. This character causes *quitting* (see Section 19.8 [Quitting], page 306).

pre-command-hook Variable

> The editor command loop runs this normal hook before each command. At that time, `this-command` contains the command that is about to run, and `last-command` describes the previous command. See Section 26.4 [Hooks], page 378.

post-command-hook Variable

The editor command loop runs this normal hook after each command. (In FSF Emacs, it is also run when the command loop is entered, or reentered after an error or quit.) At that time, `this-command` describes the command that just ran, and `last-command` describes the command before that. See Section 26.4 [Hooks], page 378.

Quitting is suppressed while running `pre-command-hook` and `post-command-hook`. If an error happens while executing one of these hooks, it terminates execution of the hook, but that is all it does.

19.2 Defining Commands

A Lisp function becomes a command when its body contains, at top level, a form that calls the special form `interactive`. This form does nothing when actually executed, but its presence serves as a flag to indicate that interactive calling is permitted. Its argument controls the reading of arguments for an interactive call.

19.2.1 Using `interactive`

This section describes how to write the `interactive` form that makes a Lisp function an interactively-callable command.

interactive *arg-descriptor* Special Form

This special form declares that the function in which it appears is a command, and that it may therefore be called interactively (via *M-x* or by entering a key sequence bound to it). The argument *arg-descriptor* declares how to compute the arguments to the command when the command is called interactively.

A command may be called from Lisp programs like any other function, but then the caller supplies the arguments and *arg-descriptor* has no effect.

The `interactive` form has its effect because the command loop (actually, its subroutine `call-interactively`) scans through the function definition looking for it, before calling the function. Once the function is called, all its body forms including the `interactive` form are executed, but at this time `interactive` simply returns `nil` without even evaluating its argument.

There are three possibilities for the argument *arg-descriptor*:

- It may be omitted or `nil`; then the command is called with no arguments. This leads quickly to an error if the command requires one or more arguments.
- It may be a Lisp expression that is not a string; then it should be a form that is evaluated to get a list of arguments to pass to the command.

  If this expression reads keyboard input (this includes using the minibuffer), keep in mind that the integer value of point or the mark before reading input may be incorrect after reading input. This is because the current buffer may be receiving subprocess output; if subprocess output arrives while the command is waiting for input, it could relocate point and the mark.

  Here's an example of what *not* to do:

```
(interactive
 (list (region-beginning) (region-end)
       (read-string "Foo: " nil 'my-history)))
```

Here's how to avoid the problem, by examining point and the mark only after reading the keyboard input:

```
(interactive
 (let ((string (read-string "Foo: " nil 'my-history)))
   (list (region-beginning) (region-end) string)))
```

- It may be a string; then its contents should consist of a code character followed by a prompt (which some code characters use and some ignore). The prompt ends either with the end of the string or with a newline. Here is a simple example:

  ```
  (interactive "bFrobnicate buffer: ")
  ```

  The code letter '`b`' says to read the name of an existing buffer, with completion. The buffer name is the sole argument passed to the command. The rest of the string is a prompt.

  If there is a newline character in the string, it terminates the prompt. If the string does not end there, then the rest of the string should contain another code character and prompt, specifying another argument. You can specify any number of arguments in this way.

  The prompt string can use '`%`' to include previous argument values (starting with the first argument) in the prompt. This is done using `format` (see Section 4.10 [Formatting Strings], page 67). For example, here is how you could read the name of an existing buffer followed by a new name to give to that buffer:

  ```
  (interactive "bBuffer to rename: \nsRename buffer %s to: ")
  ```

  If the first character in the string is '`*`', then an error is signaled if the buffer is read-only.

  If the first character in the string is '`@`', and if the key sequence used to invoke the command includes any mouse events, then the window associated with the first of those events is selected before the command is run.

  If the first character in the string is '`_`', then this command will not cause the region to be deactivated when it completes; that is, `zmacs-region-stays` will be set to `t` when the command exits successfully.

  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 '`_`').

function-interactive *function* Function

This function retrieves the interactive specification of *function*, which may be any funcallable object. The specification will be returned as the list of the symbol `interactive` and the specs. If *function* is not interactive, `nil` will be returned.

19.2.2 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 18.5 [Completion], page 266). `?` displays a list of possible completions.

Existing Require the name of an existing object. An invalid name is not accepted; the commands to exit the minibuffer do not exit if the current input is not valid.

Default A default value of some sort is used if the user enters no text in the minibuffer. The default depends on the code character.

No I/O This code letter computes an argument without reading any input. Therefore, it does not use a prompt string, and any prompt string you supply is ignored.

Even though the code letter doesn't use a prompt string, you must follow it with a newline if it is not the last code character in the string.

Prompt A prompt immediately follows the code character. The prompt ends either with the end of the string or with a newline.

Special This code character is meaningful only at the beginning of the interactive string, and it does not look for a prompt or a newline. It is a single, isolated character.

Here are the code character descriptions for use with `interactive`:

'`*`' Signal an error if the current buffer is read-only. Special.

'`@`' Select the window mentioned in the first mouse event in the key sequence that invoked this command. Special.

'`_`' Do not cause the region to be deactivated when this command completes. Special.

'`a`' A function name (i.e., a symbol satisfying `fboundp`). Existing, Completion, Prompt.

'`b`' The name of an existing buffer. By default, uses the name of the current buffer (see Chapter 30 [Buffers], page 431). Existing, Completion, Default, Prompt.

'`B`' A buffer name. The buffer need not exist. By default, uses the name of a recently used buffer other than the current buffer. Completion, Default, Prompt.

'`c`' A character. The cursor does not move into the echo area. Prompt.

'`C`' A command name (i.e., a symbol satisfying `commandp`). Existing, Completion, Prompt.

'`d`' The position of point, as an integer (see Section 34.1 [Point], page 487). No I/O.

'`D`' A directory name. The default is the current default directory of the current buffer, `default-directory` (see Section 50.3 [System Environment], page 697). Existing, Completion, Default, Prompt.

'`e`' The last mouse-button or misc-user event in the key sequence that invoked the command. No I/O.

You can use '`e`' more than once in a single command's interactive specification. If the key sequence that invoked the command has *n* mouse-button or misc-user events, the *n*th '`e`' provides the *n*th such event.

'`f`' A file name of an existing file (see Section 28.8 [File Names], page 406). 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 20.1 [Keymap Terminology], page 315). 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 vector of events. The cursor does not move into the echo area. Prompt.

This kind of input is used by commands such as `describe-key` and `global-set-key`.

'`K`' A key sequence, whose definition you intend to change. This works like '`k`', except that it suppresses, for the last input event in the key sequence, the conversions that are normally used (when necessary) to convert an undefined key into a defined one.

'`m`' The position of the mark, as an integer. No I/O.

'`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 read a number as with *`n`*. Requires a number. See Section 19.9 [Prefix Command Arguments], page 308. Prompt.

'`p`' The numeric prefix argument. (Note that this '`p`' is lower case.) No I/O.

'`P`' The raw prefix argument. (Note that this '`P`' is upper case.) No I/O.

'`r`' Point and the mark, as two numeric arguments, smallest first. This is the only code letter that specifies two successive arguments rather than one. No I/O.

'`s`' Arbitrary text, read in the minibuffer and returned as a string (see Section 18.2 [Text from Minibuffer], page 261). 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 18.5.4 [High-Level Completion], page 271. Existing, Completion, Prompt.

'`x`' A Lisp object, specified with its read syntax, terminated with a ⟨LFD⟩ or ⟨RET⟩. The object is not evaluated. See Section 18.3 [Object from Minibuffer], page 263. Prompt.

'X' A Lisp form is read as with `x`, but then evaluated so that its value becomes the argument for the command. Prompt.

19.2.3 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
```

19.3 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`, compiled-function objects made from such lambda expressions, autoload objects that are declared as interactive (non-`nil` fourth argument to `autoload`), and some of the primitive functions.

A symbol is `commandp` if its function definition is `commandp`.

Keys and keymaps are not commands. Rather, they are used to look up commands (see Chapter 20 [Keymaps], page 315).

See `documentation` in Section 27.2 [Accessing Documentation], page 382, 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* is not a function or if it cannot be called interactively (i.e., is not a command). Note that keyboard macros (strings and vectors) are not accepted, even though they are considered commands, because they are not functions.

If *record-flag* is the symbol `lambda`, the interactive calling arguments for `command` are read and returned as a list, but the function is not called on them.

If *record-flag* is `t`, 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 19.12 [Command History], page 312.

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 keyboard macro.

A string or vector as *command* is executed with `execute-kbd-macro`. A function is passed to `call-interactively`, along with the optional *record-flag*.

A symbol is handled by using its function definition in its place. A symbol with an `autoload` definition counts as a command if it was declared to stand for an interactively callable function. Such a definition is handled by loading the specified library and then rechecking the definition of the symbol.

execute-extended-command *prefix-argument* Command

This function reads a command name from the minibuffer using `completing-read` (see Section 18.5 [Completion], page 266). Then it uses `command-execute` to call the specified command. Whatever that command returns becomes the value of `execute-extended-command`.

If the command asks for a prefix argument, it receives the value *prefix-argument*. If `execute-extended-command` is called interactively, the current raw prefix argument is used for *prefix-argument*, and thus passed on to whatever command is run.

`execute-extended-command` is the normal definition of *M-x*, so it uses the string '`M-x `' as a prompt. (It would be better to take the prompt from the events used to invoke `execute-extended-command`, but that is painful to implement.) A description of the value of the prefix argument, if any, also becomes part of the prompt.

```
(execute-extended-command 1)
---------- Buffer: Minibuffer ----------
1 M-x forward-word RET
---------- Buffer: Minibuffer ----------
     ⇒ t
```

interactive-p Function

This function returns `t` if the containing function (the one 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.) If the containing function was called by Lisp evaluation (or with `apply` or `funcall`), then it was not called interactively.

The most common use of `interactive-p` is for deciding whether to print an informative message. As a special exception, `interactive-p` returns `nil` whenever a keyboard macro is being run. This is to suppress the informative messages and speed execution of the macro.

For example:

```
(defun foo ()
  (interactive)
  (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)
```

19.4 Information from the Command Loop

The editor command loop sets several Lisp variables to keep status records for itself and for commands that are run.

last-command Variable

This variable records the name of the previous command executed by the command loop (the one before the current command). Normally the value is a symbol with a function definition, but this is not guaranteed.

The value is copied from `this-command` when a command returns to the command loop, except when the command 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.

The command loop sets this variable just before running a command, and copies its value into `last-command` when the command finishes (unless the command specifies a prefix argument for the following command).

Some commands set this variable during their execution, as a flag for whatever command runs next. In particular, the functions for killing text set `this-command` to `kill-region` so that any kill commands immediately following will know to append the killed text to the previous kill.

If you do not want a particular command to be recognized as the previous command in the case where it got an error, you must code that command to prevent this. One way is to set `this-command` to `t` at the beginning of the command, and set `this-command` back to its proper value at the end, like this:

```
(defun foo (args...)
  (interactive ...)
  (let ((old-this-command this-command))
    (setq this-command t)
    ...do the work...
    (setq this-command old-this-command)))
```

this-command-keys Function

This function returns a vector containing the key and mouse events that invoked the present command, plus any previous commands that generated the prefix argument for this command. (Note: this is not the same as in FSF Emacs, which can return a string.) See Section 19.5 [Events], page 290.

This function copies the vector and the events; it is safe to keep and modify them.

```
(this-command-keys)
;; Now use C-u C-x C-e to evaluate that.
     ⇒ [#<keypress-event control-U> #<keypress-event control-X> #<keypress-event control-E>]
```

last-command-event 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.

This variable is off limits: you may not set its value or modify the event that is its value, as it is destructively modified by `read-key-sequence`. If you want to keep a pointer to this value, you must use `copy-event`.

Note that this variable is an alias for `last-command-char` in FSF Emacs.

```
last-command-event
;; Now type C-u C-x C-e.
     ⇒ #<keypress-event control-E>
```

last-command-char Variable

If the value of `last-command-event` is a keyboard event, then this is the nearest character equivalent to it (or `nil` if there is no character equivalent). `last-command-char` is the character that `self-insert-command` will insert in the buffer. Remember that there is *not* a one-to-one mapping between keyboard events and XEmacs characters: many keyboard events have no corresponding character, and when the Mule feature is available, most characters can not be input on standard keyboards, except possibly with help from an input method. So writing code that examines this variable to determine what key has been typed is bad practice, unless you are certain that it will be one of a small set of characters.

This variable exists for compatibility with Emacs version 18.

```
last-command-char
;; Now use C-u C-x C-e to evaluate that.
     ⇒ ?\^E
```

current-mouse-event Variable

This variable holds the mouse-button event which invoked this command, or `nil`. This is what `(interactive "e")` returns.

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.

19.5 Events

The XEmacs command loop reads a sequence of *events* that represent keyboard or mouse activity. Unlike in Emacs 18 and in FSF Emacs, events are a primitive Lisp type that must be manipulated using their own accessor and settor primitives. This section describes the representation and meaning of input events in detail.

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.

For information about how exactly the XEmacs command loop works, See Section 19.6 [Reading Input], page 301.

eventp *object* Function

This function returns non-`nil` if *event* is an input event.

19.5.1 Event Types

Events represent keyboard or mouse activity or status changes of various sorts, such as process input being available or a timeout being triggered. The different event types are as follows:

key-press event
: A key was pressed. Note that modifier keys such as "control", "shift", and "alt" do not generate events; instead, they are tracked internally by XEmacs, and non-modifier key presses generate events that specify both the key pressed and the modifiers that were held down at the time.

button-press event
button-release event
: A button was pressed or released. Along with the button that was pressed or released, button events specify the modifier keys that were held down at the time and the position of the pointer at the time.

motion event
: The pointer was moved. Along with the position of the pointer, these events also specify the modifier keys that were held down at the time.

misc-user event
: A menu item was selected, the scrollbar was used, or a drag or a drop occurred.

process event
: Input is available on a process.

timeout event
: A timeout has triggered.

magic event
: Some window-system-specific action (such as a frame being resized or a portion of a frame needing to be redrawn) has occurred. The contents of this event are not accessible at the E-Lisp level, but `dispatch-event` knows what to do with an event of this type.

eval event
: This is a special kind of event specifying that a particular function needs to be called when this event is dispatched. An event of this type is sometimes placed in the event queue when a magic event is processed. This kind of event should generally just be passed off to `dispatch-event`. See Section 19.6.3 [Dispatching an Event], page 303.

19.5.2 Contents of the Different Types of Events

Every event, no matter what type it is, contains a timestamp (which is typically an offset in milliseconds from when the X server was started) indicating when the event occurred. In addition, many events contain a *channel*, which specifies which frame the event occurred on, and/or a value indicating which modifier keys (shift, control, etc.) were held down at the time of the event.

The contents of each event are as follows:

key-press event

channel

timestamp

key
: Which key was pressed. This is an integer (in the printing ASCII range: `>`32 and `<`127) or a symbol such as `left` or `right`. Note that many physical keys are actually treated as two separate keys, depending on whether the shift key is pressed; for example, the "a" key is treated as either "a" or "A" depending on the state of the shift key, and the "1" key is similarly treated as either "1" or "!" on most keyboards. In such cases, the shift key does not show up in the modifier list. For other keys, such as `backspace`, the shift key shows up as a regular modifier.

modifiers
: Which modifier keys were pressed. As mentioned above, the shift key is not treated as a modifier for many keys and will not show up in this list in such cases.

button-press event
button-release event

channel

timestamp

button
: What button went down or up. Buttons are numbered starting at 1.

modifiers
: Which modifier keys were pressed. The special business mentioned above for the shift key does *not* apply to mouse events.

x
y
: The position of the pointer (in pixels) at the time of the event.

pointer-motion event

channel

timestamp

x
y
: The position of the pointer (in pixels) after it moved.

modifiers
: Which modifier keys were pressed. The special business mentioned above for the shift key does *not* apply to mouse events.

misc-user event

timestamp

function
: The E-Lisp function to call for this event. This is normally either `eval` or `call-interactively`.

object
: The object to pass to the function. This is normally the callback that was specified in the menu description.

button — What button went down or up. Buttons are numbered starting at 1.

modifiers — Which modifier keys were pressed. The special business mentioned above for the shift key does *not* apply to mouse events.

x

y — The position of the pointer (in pixels) at the time of the event.

process_event

timestamp

process — The Emacs "process" object in question.

timeout event

timestamp

function — The E-Lisp function to call for this timeout. It is called with one argument, the event.

object — Some Lisp object associated with this timeout, to make it easier to tell them apart. The function and object for this event were specified when the timeout was set.

magic event

timestamp

(The rest of the information in this event is not user-accessible.)

eval event

timestamp

function — An E-Lisp function to call when this event is dispatched.

object — The object to pass to the function. The function and object are set when the event is created.

event-type *event* — Function

Return the type of *event*.

This will be a symbol; one of

`key-press`
: A key was pressed.

`button-press`
: A mouse button was pressed.

`button-release`
: A mouse button was released.

`motion`
: The mouse moved.

`misc-user`
: Some other user action happened; typically, this is a menu selection, scrollbar action, or drag and drop action.

`process` Input is available from a subprocess.

`timeout` A timeout has expired.

`eval` This causes a specified action to occur when dispatched.

`magic` Some window-system-specific event has occurred.

19.5.3 Event Predicates

The following predicates return whether an object is an event of a particular type.

key-press-event-p *object* Function
This is true if *object* is a key-press event.

button-event-p *object object* Function
This is true if *object* is a mouse button-press or button-release event.

button-press-event-p *object* Function
This is true if *object* is a mouse button-press event.

button-release-event-p *object* Function
This is true if *object* is a mouse button-release event.

motion-event-p *object* Function
This is true if *object* is a mouse motion event.

mouse-event-p *object* Function
This is true if *object* is a mouse button-press, button-release or motion event.

eval-event-p *object* Function
This is true if *object* is an eval event.

misc-user-event-p *object* Function
This is true if *object* is a misc-user event.

process-event-p *object* Function
This is true if *object* is a process event.

timeout-event-p *object* Function
This is true if *object* is a timeout event.

event-live-p *object* Function
This is true if *object* is any event that has not been deallocated.

19.5.4 Accessing the Position of a Mouse Event

Unlike other events, mouse events (i.e. motion, button-press, button-release, and drag or drop type misc-user events) occur in a particular location on the screen. Many primitives are provided for determining exactly where the event occurred and what is under that location.

19.5.4.1 Frame-Level Event Position Info

The following functions return frame-level information about where a mouse event occurred.

event-frame *event* Function

This function returns the "channel" or frame that the given mouse motion, button press, button release, or misc-user event occurred in. This will be `nil` for non-mouse events.

event-x-pixel *event* Function

This function returns the X position in pixels of the given mouse event. The value returned is relative to the frame the event occurred in. This will signal an error if the event is not a mouse event.

event-y-pixel *event* Function

This function returns the Y position in pixels of the given mouse event. The value returned is relative to the frame the event occurred in. This will signal an error if the event is not a mouse event.

19.5.4.2 Window-Level Event Position Info

The following functions return window-level information about where a mouse event occurred.

event-window *event* Function

Given a mouse motion, button press, button release, or misc-user event, compute and return the window on which that event occurred. This may be `nil` if the event occurred in the border or over a toolbar. The modeline is considered to be within the window it describes.

event-buffer *event* Function

Given a mouse motion, button press, button release, or misc-user event, compute and return the buffer of the window on which that event occurred. This may be `nil` if the event occurred in the border or over a toolbar. The modeline is considered to be within the window it describes. This is equivalent to calling `event-window` and then calling `window-buffer` on the result if it is a window.

event-window-x-pixel *event* Function

This function returns the X position in pixels of the given mouse event. The value returned is relative to the window the event occurred in. This will signal an error if the event is not a mouse-motion, button-press, button-release, or misc-user event.

event-window-y-pixel *event* Function

This function returns the Y position in pixels of the given mouse event. The value returned is relative to the window the event occurred in. This will signal an error if the event is not a mouse-motion, button-press, button-release, or misc-user event.

19.5.4.3 Event Text Position Info

The following functions return information about the text (including the modeline) that a mouse event occurred over or near.

event-over-text-area-p *event* Function

Given a mouse-motion, button-press, button-release, or misc-user event, this function returns `t` if the event is over the text area of a window. Otherwise, `nil` is returned. The modeline is not considered to be part of the text area.

event-over-modeline-p *event* Function

Given a mouse-motion, button-press, button-release, or misc-user event, this function returns `t` if the event is over the modeline of a window. Otherwise, `nil` is returned.

event-x *event* Function

This function returns the X position of the given mouse-motion, button-press, button-release, or misc-user event in characters. This is relative to the window the event occurred over.

event-y *event* Function

This function returns the Y position of the given mouse-motion, button-press, button-release, or misc-user event in characters. This is relative to the window the event occurred over.

event-point *event* Function

This function returns the character position of the given mouse-motion, button-press, button-release, or misc-user event. If the event did not occur over a window, or did not occur over text, then this returns `nil`. Otherwise, it returns an index into the buffer visible in the event's window.

event-closest-point *event* Function

This function returns the character position of the given mouse-motion, button-press, button-release, or misc-user event. If the event did not occur over a window or over text, it returns the closest point to the location of the event. If the Y pixel position overlaps a window and the X pixel position is to the left of that window, the closest point is the beginning of the line containing the Y position. If the Y pixel position overlaps a window and the X pixel position is to the right of that window, the closest point is the end of the line containing the Y position. If the Y pixel position is above a window, 0 is returned. If it is below a window, the value of `(window-end)` is returned.

19.5.4.4 Event Glyph Position Info

The following functions return information about the glyph (if any) that a mouse event occurred over.

event-over-glyph-p *event* Function

Given a mouse-motion, button-press, button-release, or misc-user event, this function returns `t` if the event is over a glyph. Otherwise, `nil` is returned.

event-glyph-extent *event* Function

If the given mouse-motion, button-press, button-release, or misc-user event happened on top of a glyph, this returns its extent; else `nil` is returned.

event-glyph-x-pixel *event* Function

Given a mouse-motion, button-press, button-release, or misc-user event over a glyph, this function returns the X position of the pointer relative to the upper left of the glyph. If the event is not over a glyph, it returns `nil`.

event-glyph-y-pixel *event* Function

Given a mouse-motion, button-press, button-release, or misc-user event over a glyph, this function returns the Y position of the pointer relative to the upper left of the glyph. If the event is not over a glyph, it returns `nil`.

19.5.4.5 Event Toolbar Position Info

event-over-toolbar-p *event* Function

Given a mouse-motion, button-press, button-release, or misc-user event, this function returns `t` if the event is over a toolbar. Otherwise, `nil` is returned.

event-toolbar-button *event* Function

If the given mouse-motion, button-press, button-release, or misc-user event happened on top of a toolbar button, this function returns the button. Otherwise, `nil` is returned.

19.5.4.6 Other Event Position Info

event-over-border-p *event* Function

Given a mouse-motion, button-press, button-release, or misc-user event, this function returns `t` if the event is over an internal toolbar. Otherwise, `nil` is returned.

19.5.5 Accessing the Other Contents of Events

The following functions allow access to the contents of events other than the position info described in the previous section.

event-timestamp *event* Function

This function returns the timestamp of the given event object.

event-device *event* Function

This function returns the device that the given event occurred on.

event-key *event* Function

This function returns the Keysym of the given key-press event. This will be the ASCII code of a printing character, or a symbol.

event-button *event* Function

This function returns the button-number of the given button-press or button-release event.

event-modifiers *event* Function

This function returns a list of symbols, the names of the modifier keys which were down when the given mouse or keyboard event was produced.

event-modifier-bits *event* Function

This function returns a number representing the modifier keys which were down when the given mouse or keyboard event was produced.

event-function *event* Function

This function returns the callback function of the given timeout, misc-user, or eval event.

event-object *event* Function

This function returns the callback function argument of the given timeout, misc-user, or eval event.

event-process *event* Function

This function returns the process of the given process event.

19.5.6 Working With Events

XEmacs provides primitives for creating, copying, and destroying event objects. Many functions that return events take an event object as an argument and fill in the fields of this event; or they make accept either an event object or `nil`, creating the event object first in the latter case.

make-event &optional *type plist* Function

This function creates a new event structure. If no arguments are specified, the created event will be empty. To specify the event type, use the *type* argument. The allowed types are `empty`, `key-press`, `button-press`, `button-release`, `motion`, or `misc-user`.

plist is a property list, the properties being compatible to those returned by `event-properties`. For events other than `empty`, it is mandatory to specify certain properties. For `empty` events, *plist* must be `nil`. The list is *canonicalized*, which means that if a property keyword is present more than once, only the first instance is taken into account. Specifying an unknown or illegal property signals an error.

The following properties are allowed:

`channel` The event channel. This is a frame or a console. For mouse events (of type `button-press`, `button-release` and `motion`), this must be a frame. For key-press events, it must be a console. If channel is unspecified by *plist*, it will be set to the selected frame or selected console, as appropriate.

`key` The event key. This is either a symbol or a character. It is allowed (and required) only for key-press events.

`button` The event button. This an integer, either 1, 2 or 3. It is allowed only for button-press and button-release events.

`modifiers`
The event modifiers. This is a list of modifier symbols. It is allowed for key-press, button-press, button-release and motion events.

`x` The event X coordinate. This is an integer. It is relative to the channel's root window, and is allowed for button-press, button-release and motion events.

`y` The event Y coordinate. This is an integer. It is relative to the channel's root window, and is allowed for button-press, button-release and motion events. This means that, for instance, to access the toolbar, the `y` property will have to be negative.

`timestamp`
The event timestamp, a non-negative integer. Allowed for all types of events.

WARNING: the event object returned by this function may be a reused one; see the function `deallocate-event`.

The events created by `make-event` can be used as non-interactive arguments to the functions with an `(interactive "e")` specification.

Here are some basic examples of usage:

```
;; Create an empty event.
(make-event)
     ⇒ #<empty-event>

;; Try creating a key-press event.
(make-event 'key-press)
     error  Undefined key for keypress event

;; Creating a key-press event, try 2
(make-event 'key-press '(key home))
     ⇒ #<keypress-event home>

;; Create a key-press event of dubious fame.
(make-event 'key-press '(key escape modifiers (meta alt control shift)))
     ⇒ #<keypress-event control-meta-alt-shift-escape>

;; Create a M-button1 event at coordinates defined by variables
;; x and y.
(make-event 'button-press `(button 1 modifiers (meta) x ,x y ,y))
     ⇒ #<buttondown-event meta-button1>

;; Create a similar button-release event.
(make-event 'button-release `(button 1 modifiers (meta) x ,x y ,x))
     ⇒ #<buttonup-event meta-button1up>
```

```
;; Create a mouse-motion event.
(make-event 'motion '(x 20 y 30))
     ⇒ #<motion-event 20, 30>

(event-properties (make-event 'motion '(x 20 y 30)))
     ⇒ (channel #<x-frame "emacs" 0x8e2> x 20 y 30
         modifiers nil timestamp 0)
```

In conjunction with `event-properties`, you can use `make-event` to create modified copies of existing events. For instance, the following code will return an `equal` copy of *event*:

```
(make-event (event-type event)
            (event-properties event))
```

Note, however, that you cannot use `make-event` as the generic replacement for `copy-event`, because it does not allow creating all of the event types.

To create a modified copy of an event, you can use the canonicalization feature of *plist*. The following example creates a copy of *event*, but with `modifiers` reset to `nil`.

```
(make-event (event-type event)
            (append '(modifiers nil)
                    (event-properties event)))
```

copy-event *event1* &optional *event2* Function

This function makes a copy of the given event object. If a second argument is given, the first event is copied into the second and the second is returned. If the second argument is not supplied (or is `nil`) then a new event will be made.

deallocate-event *event* Function

This function allows the given event structure to be reused. You **MUST NOT** use this event object after calling this function with it. You will lose. It is not necessary to call this function, as event objects are garbage-collected like all other objects; however, it may be more efficient to explicitly deallocate events when you are sure that that is safe.

19.5.7 Converting Events

XEmacs provides some auxiliary functions for converting between events and other ways of representing keys. These are useful when working with ASCII strings and with keymaps.

character-to-event *ch* &optional *event device* Function

This function converts a numeric ASCII value to an event structure, replete with modifier bits. *ch* is the character to convert, and *event* is the event object to fill in. This function contains knowledge about what the codes "mean" – for example, the number 9 is converted to the character ⟨Tab⟩, not the distinct character ⟨Control-I⟩.

Note that *ch* does not have to be a numeric value, but can be a symbol such as `clear` or a list such as `(control backspace)`.

If `event` is not `nil`, it is modified; otherwise, a new event object is created. In both cases, the event is returned.

Optional third arg *device* is the device to store in the event; this also affects whether the high bit is interpreted as a meta key. A value of `nil` means use the selected device but always treat the high bit as meta.

Beware that `character-to-event` and `event-to-character` are not strictly inverse functions, since events contain much more information than the ASCII character set can encode.

event-to-character *event* &optional *allow-extra-modifiers allow-meta allow-non-ascii* — Function

This function returns the closest ASCII approximation to *event*. If the event isn't a keypress, this returns `nil`.

If *allow-extra-modifiers* is non-`nil`, then this is lenient in its translation; it will ignore modifier keys other than ⟨control⟩ and ⟨meta⟩, and will ignore the ⟨shift⟩ modifier on those characters which have no shifted ASCII equivalent (⟨Control-Shift-A⟩ for example, will be mapped to the same ASCII code as ⟨Control-A⟩).

If *allow-meta* is non-`nil`, then the ⟨Meta⟩ modifier will be represented by turning on the high bit of the byte returned; otherwise, `nil` will be returned for events containing the ⟨Meta⟩ modifier.

If *allow-non-ascii* is non-`nil`, then characters which are present in the prevailing character set (see Chapter 20 [Keymaps], page 315) will be returned as their code in that character set, instead of the return value being restricted to ASCII.

Note that specifying both *allow-meta* and *allow-non-ascii* is ambiguous, as both use the high bit; ⟨M-x⟩ and ⟨oslash⟩ will be indistinguishable.

events-to-keys *events* &optional *no-mice* — Function

Given a vector of event objects, this function returns a vector of key descriptors, or a string (if they all fit in the ASCII range). Optional arg *no-mice* means that button events are not allowed.

19.6 Reading Input

The editor command loop reads keyboard input using the function `next-event` and constructs key sequences out of the events using `dispatch-event`. Lisp programs can also use the function `read-key-sequence`, which reads input a key sequence at a time. See also `momentary-string-display` in Section 45.8 [Temporary Displays], page 656, and `sit-for` in Section 19.7 [Waiting], page 305. See Section 50.8 [Terminal Input], page 706, for functions and variables for controlling terminal input modes and debugging terminal input.

For higher-level input facilities, see Chapter 18 [Minibuffers], page 261.

19.6.1 Key Sequence Input

Lisp programs can read input a key sequence at a time by calling `read-key-sequence`; for example, `describe-key` uses it to read the key to describe.

read-key-sequence *prompt* Function

This function reads a sequence of keystrokes or mouse clicks and returns it as a vector of events. 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.

The vector and the event objects it contains are freshly created, and will not be side-effected by subsequent calls to this function.

The function `read-key-sequence` suppresses quitting: *C-g* typed while reading with this function works like any other character, and does not set `quit-flag`. See Section 19.8 [Quitting], page 306.

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.

If the user selects a menu item while we are prompting for a key sequence, the returned value will be a vector of a single menu-selection event (a misc-user event). An error will be signalled if you pass this value to `lookup-key` or a related function.

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 ----------

     ⇒ [#<keypress-event control-X> #<keypress-event control-
F>]
```

If an input character is an upper-case letter and has no key binding, but its lower-case equivalent has one, then `read-key-sequence` converts the character to lower case. Note that `lookup-key` does not perform case conversion in this way.

19.6.2 Reading One Event

The lowest level functions for command input are those which read a single event. These functions often make a distinction between *command events*, which are user actions (keystrokes and mouse actions), and other events, which serve as communication between XEmacs and the window system.

next-event &optional *event prompt* Function

This function reads and returns the next available event from the window system or terminal driver, waiting if necessary until an event is available. Pass this object to `dispatch-event` to handle it. If an event object is supplied, it is filled in and returned; otherwise a new event object will be created.

Events can come directly from the user, from a keyboard macro, or from `unread-command-events`.

In most cases, the function `next-command-event` is more appropriate.

next-command-event &optional *event* Function

This function returns the next available "user" event from the window system or terminal driver. Pass this object to `dispatch-event` to handle it. If an event object is supplied, it is filled in and returned, otherwise a new event object will be created.

The event returned will be a keyboard, mouse press, or mouse release event. If there are non-command events available (mouse motion, sub-process output, etc) then these will be executed (with `dispatch-event`) and discarded. This function is provided as a convenience; it is equivalent to the Lisp code

```
(while (progn
 (next-event event)
        (not (or (key-press-event-p event)
                 (button-press-event-p event)
                 (button-release-event-p event)
                 (menu-event-p event))))
   (dispatch-event event))
```

Here is what happens if you call `next-command-event` and then press the right-arrow function key:

```
(next-command-event)
     ⇒ #<keypress-event right>
```

read-char Function

This function reads and returns a character of command input. If a mouse click is detected, an error is signalled. The character typed is returned as an ASCII value. This function is retained for compatibility with Emacs 18, and is most likely the wrong thing for you to be using: consider using `next-command-event` instead.

enqueue-eval-event *function object* Function

This function adds an eval event to the back of the queue. The eval event will be the next event read after all pending events.

19.6.3 Dispatching an Event

dispatch-event *event* Function

Given an event object returned by `next-event`, this function executes it. This is the basic function that makes XEmacs respond to user input; it also deals with notifications from the window system (such as Expose events).

19.6.4 Quoted Character Input

You can use the function `read-quoted-char` to ask the user to specify a character, and allow the user to specify a control or meta character conveniently, either literally or as an octal character code. The command `quoted-insert` uses this function.

read-quoted-char &optional *prompt* Function

This function is like `read-char`, except that if the first character read is an octal digit (0-7), it reads up to two more octal digits (but stopping if a non-octal digit is found) and returns the character represented by those digits in octal.

Quitting is suppressed when the first character is read, so that the user can enter a *C-g*. See Section 19.8 [Quitting], page 306.

If *prompt* is supplied, it specifies a string for prompting the user. The prompt string is always displayed in the echo area, followed by a single '-'.

In the following example, the user types in the octal number 177 (which is 127 in decimal).

```
(read-quoted-char "What character")

---------- Echo Area ----------
What character-177
---------- Echo Area ----------

     ⇒ 127
```

19.6.5 Miscellaneous Event Input Features

This section describes how to "peek ahead" at events without using them up, how to check for pending input, and how to discard pending input.

See also the variables `last-command-event` and `last-command-char` (Section 19.4 [Command Loop Info], page 288).

unread-command-events Variable

This variable holds a list of events waiting to be read as command input. The events are used in the order they appear in the list, and removed one by one as they are used.

The variable is needed because in some cases a function reads 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 by the functions to read command input.

For example, the function that implements numeric prefix arguments reads any number of digits. When it finds a non-digit event, it must unread the event so that it can be read normally by the command loop. Likewise, incremental search uses this feature to unread events with no special meaning in a search, because these events should exit the search and then execute normally.

unread-command-event Variable

This variable holds a single event to be read as command input.

This variable is mostly obsolete now that you can use `unread-command-events` instead; it exists only to support programs written for versions of XEmacs prior to 19.12.

input-pending-p Function

This function determines whether any command input is currently available to be read. It returns immediately, with value `t` if there is available input, `nil` otherwise. On rare occasions it may return `t` when no input is available.

last-input-event Variable

This variable is set to the last keyboard or mouse button event received.

This variable is off limits: you may not set its value or modify the event that is its value, as it is destructively modified by `read-key-sequence`. If you want to keep a pointer to this value, you must use `copy-event`.

Note that this variable is an alias for `last-input-char` in FSF Emacs.

In the example below, a character is read (the character *1*). It becomes the value of `last-input-event`, while *C-e* (from the *C-x C-e* command used to evaluate this expression) remains the value of `last-command-event`.

```
(progn (print (next-command-event))
       (print last-command-event)
       last-input-event)
     ⊣ #<keypress-event 1>
     ⊣ #<keypress-event control-E>
     ⇒ #<keypress-event 1>
```

last-input-char Variable

If the value of `last-input-event` is a keyboard event, then this is the nearest ASCII equivalent to it. Remember that there is *not* a 1:1 mapping between keyboard events and ASCII characters: the set of keyboard events is much larger, so writing code that examines this variable to determine what key has been typed is bad practice, unless you are certain that it will be one of a small set of characters.

This function exists for compatibility with Emacs version 18.

discard-input Function

This function discards the contents of the terminal input buffer and cancels any keyboard macro that might be in the process of definition. It returns `nil`.

In the following example, the user may type a number of characters right after starting the evaluation of the form. After the `sleep-for` finishes sleeping, `discard-input` discards any characters typed during the sleep.

```
(progn (sleep-for 2)
       (discard-input))
     ⇒ nil
```

19.7 Waiting for Elapsed Time or Input

The wait functions are designed to wait for a certain amount of time to pass or until there is input. For example, you may wish to pause in the middle of a computation to allow the user time to view the display. `sit-for` pauses and updates the screen, and returns immediately if input comes in, while `sleep-for` pauses without updating the screen.

Note that in FSF Emacs, the commands `sit-for` and `sleep-for` take two arguments to specify the time (one integer and one float value), instead of a single argument that can be either an integer or a float.

sit-for *seconds* &optional *nodisp* Function

This function performs redisplay (provided there is no pending input from the user), then waits *seconds* seconds, or until input is available. The result is `t` if `sit-for` waited the full time with no input arriving (see `input-pending-p` in Section 19.6.5 [Peeking and Discarding], page 304). Otherwise, the value is `nil`.

The argument *seconds* need not be an integer. If it is a floating point number, `sit-for` waits for a fractional number of seconds.

Redisplay is normally preempted if input arrives, and does not happen at all if input is available before it starts. (You can force screen updating in such a case by using `force-redisplay`. See Section 45.1 [Refresh Screen], page 647.) 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* Function

This function simply pauses for *seconds* seconds without updating the display. This function pays no attention to available input. It returns `nil`.

The argument *seconds* need not be an integer. If it is a floating point number, `sleep-for` waits for a fractional number of seconds.

Use `sleep-for` when you wish to guarantee a delay.

See Section 50.5 [Time of Day], page 702, for functions to get the current time.

19.8 Quitting

Typing `C-g` while a Lisp function is running causes XEmacs 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* always quit directly.

When *C-g* does directly quit, it does so by setting the variable `quit-flag` to `t`. XEmacs checks this variable at appropriate times and quits if it is not `nil`. Setting `quit-flag` non-`nil` in any way thus causes a quit.

At the level of C code, quitting cannot happen just anywhere; only at the special places that check `quit-flag`. The reason for this is that quitting at other places might leave an inconsistency in XEmacs's internal state. Because quitting is delayed until a safe place, quitting cannot make XEmacs crash.

Certain functions such as `read-key-sequence` or `read-quoted-char` prevent quitting entirely even though they wait for input. Instead of quitting, *C-g* serves as the requested input. In the case of `read-key-sequence`, this serves to bring about the special behavior of *C-g* in the command loop. In the case of `read-quoted-char`, this is so that *C-q* can be used to quote a *C-g*.

You can prevent quitting for a portion of a Lisp function by binding the variable `inhibit-quit` to a non-`nil` value. Then, although *C-g* still sets `quit-flag` to `t` as usual, the usual result of this—a quit—is prevented. Eventually, `inhibit-quit` will become `nil` again, such as when its binding is unwound at the end of a `let` form. At that time, if `quit-flag` is still non-`nil`, the requested quit happens immediately. This behavior is ideal when you wish to make sure that quitting does not happen within a "critical section" of the program.

In some functions (such as `read-quoted-char`), *C-g* is handled in a special way that does not involve quitting. This is done by reading the input with `inhibit-quit` bound to `t`, and setting `quit-flag` to `nil` before `inhibit-quit` becomes `nil` again. This excerpt from the definition of `read-quoted-char` shows how this is done; it also shows that normal quitting is permitted after the first character of input.

```
(defun read-quoted-char (&optional prompt)
  "...documentation..."
  (let ((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 XEmacs quits immediately, unless `inhibit-quit` is non-`nil`. Typing *C-g* ordinarily sets `quit-flag` non-`nil`, regardless of `inhibit-quit`.

inhibit-quit Variable

This variable determines whether XEmacs 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 136.)

You can specify a character other than *C-g* to use for quitting. See the function `set-input-mode` in Section 50.8 [Terminal Input], page 706.

19.9 Prefix Command Arguments

Most XEmacs commands can use a *prefix argument*, a number specified before the command itself. (Don't confuse prefix arguments with prefix keys.) The prefix argument is at all times represented by a value, which may be `nil`, meaning there is currently no prefix argument. Each command may use the prefix argument or ignore it.

There are two representations of the prefix argument: *raw* and *numeric*. The editor command loop uses the raw representation internally, and so do the Lisp variables that store the information, but commands can request either representation.

Here are the possible values of a raw prefix argument:

- `nil`, meaning there is no prefix argument. Its numeric value is 1, but numerous commands make a distinction between `nil` and the integer 1.
- An integer, which stands for itself.
- A list of one element, which is an integer. This form of prefix argument results from one or a succession of *C-u*'s with no digits. The numeric value is the integer in the list, but some commands make a distinction between such a list and an integer alone.
- The symbol `-`. This indicates that *M--* or *C-u -* was typed, without following digits. The equivalent numeric value is -1, but some commands make a distinction between the integer -1 and the symbol `-`.

We illustrate these possibilities by calling the following function with various prefixes:

```
(defun display-prefix (arg)
  "Display the value of the raw prefix arg."
  (interactive "P")
  (message "%s" arg))
```

Here are the results of calling `display-prefix` with various raw prefix arguments:

```
        M-x display-prefix  ⊣ nil

C-u     M-x display-prefix  ⊣ (4)

C-u C-u M-x display-prefix  ⊣ (16)

C-u 3   M-x display-prefix  ⊣ 3
```

```
M-3     M-x display-prefix  ⊣ 3      ; (Same as C-u 3.)

C-3     M-x display-prefix  ⊣ 3      ; (Same as C-u 3.)

C-u -   M-x display-prefix  ⊣ -

M--     M-x display-prefix  ⊣ -      ; (Same as C-u -.)

C--     M-x display-prefix  ⊣ -      ; (Same as C-u -.)

C-u - 7 M-x display-prefix  ⊣ -7

M-- 7   M-x display-prefix  ⊣ -7     ; (Same as C-u -7.)

C-- 7   M-x display-prefix  ⊣ -7     ; (Same as C-u -7.)
```

XEmacs 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 19.2.1 [Using Interactive], page 282.) Alternatively, functions may look at the value of the prefix argument directly in the variable `current-prefix-arg`, but this is less clean.

prefix-numeric-value *arg* Function

This function returns the numeric meaning of a valid raw prefix argument value, *arg*. The argument may be a symbol, a number, or a list. If it is `nil`, the value 1 is returned; if it is `-`, the value -1 is returned; if it is a number, that number is returned; if it is a list, the CAR of that list (which should be a number) is returned.

current-prefix-arg Variable

This variable holds the raw prefix argument for the *current* command. Commands may examine it directly, but the usual 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.

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.

19.10 Recursive Editing

The XEmacs command loop is entered automatically when XEmacs starts up. This top-level invocation of the command loop never exits; it keeps running as long as XEmacs does. Lisp programs can also invoke the command loop. Since this makes more than one activation of the command loop, we call it *recursive editing*. A recursive editing level has the effect of suspending whatever command invoked it and permitting the user to do arbitrary editing before resuming that command.

The commands available during recursive editing are the same ones available in the top-level editing loop and defined in the keymaps. Only a few special commands exit the recursive editing level; the others return to the recursive editing level when they finish. (The special commands for exiting are always available, but they do nothing when recursive editing is not in progress.)

All command loops, including recursive ones, set up all-purpose error handlers so that an error in a command run from the command loop will not exit the loop.

Minibuffer input is a special kind of recursive editing. It has a few special wrinkles, such as enabling display of the minibuffer and the minibuffer window, but fewer than you might suppose. Certain keys behave differently in the minibuffer, but that is only because of the minibuffer's local map; if you switch windows, you get the usual XEmacs 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 134). If you throw a value other than `t`, then `recursive-edit` returns normally to the function that called it. The command *C-M-c* (`exit-recursive-edit`) does this. Throwing a `t` value causes `recursive-edit` to quit, so that control returns to the command loop one level up. This is called *aborting*, and is done by *C-]* (`abort-recursive-edit`).

Most applications should not use recursive editing, except as part of using the minibuffer. Usually it is more convenient for the user if you change the major mode of the current buffer temporarily to a special major mode, which should have a command to go back to the previous mode. (The `e` command in Rmail uses this technique.) Or, if you wish to give the user different text to edit "recursively", create and select a new buffer in a special mode. In this mode, define a command to complete the processing and go back to the previous buffer. (The *m* command in Rmail does this.)

Recursive edits are useful in debugging. You can insert a call to `debug` into a function definition as a sort of breakpoint, so that you can look around when the function gets there. `debug` invokes a recursive edit but also provides the other features of the debugger.

Recursive editing levels are also used when you type *C-r* in `query-replace` or use *C-x q* (`kbd-macro-query`).

recursive-edit Function

This function invokes the editor command loop. It is called automatically by the initialization of XEmacs, 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 19.8 [Quitting], page 306.

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.

19.11 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, the message saying the command is disabled includes that string. For example:

```
(put 'delete-region 'disabled
     "Text deleted this way cannot be yanked back!\n")
```

See section "Disabling" in *The XEmacs User's Manual*, for the details on what happens when a disabled command is invoked interactively. Disabling a command has no effect on calling it as a function from Lisp programs.

enable-command *command* Command

Allow *command* to be executed without special confirmation from now on, and (if the user confirms) alter the user's '`.emacs`' file so that this will apply to future sessions.

disable-command *command* Command

Require special confirmation to execute *command* from now on, and (if the user confirms) alter the user's '`.emacs`' file so that this will apply to future sessions.

disabled-command-hook Variable

This normal hook is run instead of a disabled command, when the user invokes 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. See Section 26.4 [Hooks], page 378.

By default, `disabled-command-hook` contains a function that asks the user whether to proceed.

19.12 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-:* 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 768).

```
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 18.4 [Minibuffer History], page 264), 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 XEmacs User's Manual*). Within the minibuffer, the history commands used are the same ones available in any minibuffer.

19.13 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. The Lisp representation of a keyboard macro is a string or vector containing the events. Don't confuse keyboard macros with Lisp macros (see Chapter 12 [Macros], page 177).

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.

executing-macro Variable

This variable contains the string or vector that defines the keyboard macro that is currently executing. It is `nil` if no macro is currently executing. A command can test this variable to behave differently when run from an executing macro. Do not set this variable yourself.

defining-kbd-macro Variable

This variable indicates whether a keyboard macro is being defined. A command can test this variable to behave differently while a macro is being defined. The commands `start-kbd-macro` and `end-kbd-macro` set this variable—do not set it yourself.

last-kbd-macro Variable

This variable is the definition of the most recently defined keyboard macro. Its value is a string or vector, or `nil`.

The commands are described in the user's manual (see section "Keyboard Macros" in *The XEmacs User's Manual*).

20 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 input event; this continues until a command is found. The whole process is called *key lookup*.

20.1 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, XEmacs can get the event's definition. Events mapped in keymaps include keypresses, button presses, and button releases (see Section 19.5 [Events], page 290).

A sequence of input events that form a unit is called a *key sequence*, or *key* for short. A sequence of one event is always a key sequence, and so are some multi-event sequences.

A keymap determines a binding or definition for any key sequence. If the key sequence is a single event, its binding is the definition of the event in the keymap. The binding of a key sequence of more than one event is found by an iterative process: the binding of the first event is found, and must be a keymap; then the second event's binding is found in that keymap, and so on until all the events in the key sequence are used up.

If the binding of a key sequence is a keymap, we call the key sequence a *prefix key*. Otherwise, we call it a *complete key* (because no more events can be added to it). If the binding is `nil`, we call the key *undefined*. Examples of prefix keys are *C-c*, *C-x*, and *C-x 4*. Examples of defined complete keys are *X*, ⟨RET⟩, and *C-x 4 C-f*. Examples of undefined complete keys are *C-x C-g*, and *C-c 3*. See Section 20.6 [Prefix Keys], page 319, 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-n* 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 20.7 [Active Keymaps], page 320, for details.

20.2 Format of Keymaps

A keymap is a primitive type that associates events with their bindings. Note that this is different from Emacs 18 and FSF Emacs, where keymaps are lists.

keymapp *object* Function

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

20.3 Creating Keymaps

Here we describe the functions for creating keymaps.

make-keymap &optional *name* Function

This function constructs and returns a new keymap object. All entries in it are `nil`, meaning "command undefined".

Optional argument *name* specifies a name to assign to the keymap, as in `set-keymap-name`. This name is only a debugging convenience; it is not used except when printing the keymap.

make-sparse-keymap &optional *name* Function

This function constructs and returns a new keymap object. All entries in it are `nil`, meaning "command undefined". The only difference between this function and `make-keymap` is that this function returns a "smaller" keymap (one that is expected to contain fewer entries). As keymaps dynamically resize, the distinction is not great.

Optional argument *name* specifies a name to assign to the keymap, as in `set-keymap-name`. This name is only a debugging convenience; it is not used except when printing the keymap.

set-keymap-name *keymap new-name* Function

This function assigns a "name" to a keymap. The name is only a debugging convenience; it is not used except when printing the keymap.

keymap-name *keymap* Function

This function returns the "name" of a keymap, as assigned using `set-keymap-name`.

copy-keymap *keymap* Function

This function returns a copy of *keymap*. Any keymaps that appear directly as bindings in *keymap* are also copied recursively, and so on to any number of levels. However, recursive copying does not take place when the definition of a character is a symbol whose function definition is a keymap; the same symbol appears in the new copy.

```
(setq map (copy-keymap (current-local-map)))
⇒ #<keymap 3 entries 0x21f80>

(eq map (current-local-map))
    ⇒ nil
```

20.4 Inheritance and Keymaps

A keymap can inherit the bindings of other keymaps. The other keymaps are called the keymap's *parents*, and are set with `set-keymap-parents`. When searching for a binding for a key sequence in a particular keymap, that keymap itself will first be searched; then, if no binding was found in the map and it has parents, the first parent keymap will be searched; then that keymap's parent will be searched, and so on, until either a binding for the key sequence is found, or a keymap without a parent is encountered. At this point, the search will continue with the next parent of the most recently encountered keymap that has another parent, etc. Essentially, a depth-first search of all the ancestors of the keymap is conducted.

`(current-global-map)` is the default parent of all keymaps.

set-keymap-parents *keymap parents* Function

This function sets the parent keymaps of *keymap* to the list *parents*.

If you change the bindings in one of the keymaps in *parents* using `define-key` or other key-binding functions, these changes are visible in *keymap* unless shadowed by bindings in that map or in earlier-searched ancestors. The converse is not true: if you use `define-key` to change *keymap*, that affects the bindings in that map, but has no effect on any of the keymaps in *parents*.

keymap-parents *keymap* Function

This function returns the list of parent keymaps of *keymap*, or `nil` if *keymap* has no parents.

As an alternative to specifying a parent, you can also specify a *default binding* that is used whenever a key is not otherwise bound in the keymap. This is useful for terminal emulators, for example, which may want to trap all keystrokes and pass them on in some modified format. Note that if you specify a default binding for a keymap, neither the keymap's parents nor the current global map are searched for key bindings.

set-keymap-default-binding *keymap command* Function

This function sets the default binding of *keymap* to *command*, or `nil` if no default is desired.

keymap-default-binding *keymap* Function

This function returns the default binding of *keymap*, or `nil` if it has none.

20.5 Key Sequences

Contrary to popular belief, the world is not ASCII. When running under a window manager, XEmacs can tell the difference between, for example, the keystrokes *`control-h`*, *`control-shift-h`*, and *`backspace`*. You can, in fact, bind different commands to each of these.

A *key sequence* is a set of keystrokes. A *keystroke* is a keysym and some set of modifiers (such as ⟨CONTROL⟩ and ⟨META⟩). A *keysym* is what is printed on the keys on your keyboard.

A keysym may be represented by a symbol, or (if and only if it is equivalent to an ASCII character in the range 32 - 255) by a character or its equivalent ASCII code. The *`A`* key may be represented by the symbol `A`, the character `?A`, or by the number 65. The *break* key may be represented only by the symbol `break`.

A keystroke may be represented by a list: the last element of the list is the key (a symbol, character, or number, as above) and the preceding elements are the symbolic names of modifier keys (⟨CONTROL⟩, ⟨META⟩, ⟨SUPER⟩, ⟨HYPER⟩, ⟨ALT⟩, and ⟨SHIFT⟩). Thus, the sequence *control-b* is represented by the forms `(control b)`, `(control ?b)`, and `(control 98)`. A keystroke may also be represented by an event object, as returned by the `next-command-event` and `read-key-sequence` functions.

Note that in this context, the keystroke *control-b* is *not* represented by the number 2 (the ASCII code for '`^B`') or the character `?\^B`. See below.

The ⟨SHIFT⟩ modifier is somewhat of a special case. You should not (and cannot) use `(meta shift a)` to mean `(meta A)`, since for characters that have ASCII equivalents, the state of the shift key is implicit in the keysym ('`a`' vs. '`A`'). You also cannot say `(shift =)` to mean `+`, as that sort of thing varies from keyboard to keyboard. The ⟨SHIFT⟩ modifier is for use only with characters that do not have a second keysym on the same key, such as `backspace` and `tab`.

A key sequence is a vector of keystrokes. As a degenerate case, elements of this vector may also be keysyms if they have no modifiers. That is, the *A* keystroke is represented by all of these forms:

```
A ?A 65 (A) (?A) (65)
[A] [?A] [65] [(A)] [(?A)] [(65)]
```

the *control-a* keystroke is represented by these forms:

```
(control A) (control ?A) (control 65)
[(control A)] [(control ?A)] [(control 65)]
```

the key sequence *control-c control-a* is represented by these forms:

```
[(control c) (control a)] [(control ?c) (control ?a)]
[(control 99) (control 65)] etc.
```

Mouse button clicks work just like keypresses: `(control button1)` means pressing the left mouse button while holding down the control key. `[(control c) (shift button3)]` means *control-c*, hold ⟨SHIFT⟩, click right.

Commands may be bound to the mouse-button up-stroke rather than the down-stroke as well. `button1` means the down-stroke, and `button1up` means the up-stroke. Different commands may be bound to the up and down strokes, though that is probably not what you want, so be careful.

For backward compatibility, a key sequence may also be represented by a string. In this case, it represents the key sequence(s) that would produce that sequence of ASCII characters in a purely ASCII world. For example, a string containing the ASCII backspace character, `"\^H"`, would represent two key sequences: `(control h)` and `backspace`. Binding a command to this will actually bind both of those key sequences. Likewise for the following pairs:

```
control h backspace
control i    tab
control m    return
control j    linefeed
```

```
control [    escape
control @ control space
```

After binding a command to two key sequences with a form like

```
(define-key global-map "\^X\^I" 'command-1)
```

it is possible to redefine only one of those sequences like so:

```
(define-key global-map [(control x) (control i)] 'command-2)
(define-key global-map [(control x) tab] 'command-3)
```

Of course, all of this applies only when running under a window system. If you're talking to XEmacs through a TTY connection, you don't get any of these features.

event-matches-key-specifier-p *event key-specifier* Function

This function returns non-`nil` if *event* matches *key-specifier*, which can be any valid form representing a key sequence. This can be useful, e.g., to determine if the user pressed `help-char` or `quit-char`.

20.6 Prefix Keys

A *prefix key* has an associated keymap that 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 that is also stored in the variable `ctl-x-map`. Here is a list of the standard prefix keys of XEmacs and their keymaps:

- `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 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` is used for events that follow `C-x 5`.
- The prefix keys `C-x n`, `C-x r` and `C-x a` use keymaps that have no special name.
- `esc-map` is an evil hack that is present for compatibility purposes with Emacs 18. Defining a key in `esc-map` is equivalent to defining the same key in `global-map` but with the ⟨META⟩ prefix added. You should *not* use this in your code. (This map is also the function definition of `ESC-prefix`.)

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 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 20.7 [Active Keymaps], page 320.

If a key is defined as a prefix in more than one active map, then its various definitions are in effect merged: the commands defined in the minor mode keymaps come first, followed by those in the local map's prefix definition, and then by those from the global map.

In the following example, we make *C-p* a prefix key in the local keymap, in such a way that *C-p* is identical to *C-x*. Then the binding for *C-p C-f* is the function `find-file`, just like *C-x C-f*. The key sequence *C-p 6* is not found in any active keymap.

```
(use-local-map (make-sparse-keymap))
    ⇒ nil
(local-set-key "\C-p" ctl-x-map)
    ⇒ nil
(key-binding "\C-p\C-f")
    ⇒ find-file

(key-binding "\C-p6")
    ⇒ nil
```

define-prefix-command *symbol* &optional *mapvar* Function

This function defines *symbol* as a prefix command: it creates a keymap and stores it as *symbol*'s function definition. Storing the symbol as the binding of a key makes the key a prefix key that has a name. If optional argument *mapvar* is not specified, it also sets *symbol* as a variable, to have the keymap as its value. (If *mapvar* is given and is not `t`, its value is stored as the value of *symbol*.) The function returns *symbol*.

In Emacs version 18, only the function definition of *symbol* was set, not the value as a variable.

20.7 Active Keymaps

XEmacs normally contains many keymaps; at any given time, just a few of them are *active* in that they participate in the interpretation of user input. These are the global keymap, the current buffer's local keymap, and the keymaps of any enabled minor modes.

The *global keymap* holds the bindings of keys that are defined regardless of the current buffer, such as *C-f*. The variable `global-map` holds this keymap, which is always active.

Each buffer may have another keymap, its *local keymap*, which may contain new or overriding definitions for keys. The current buffer's local keymap is always active except when `overriding-local-map` or `overriding-terminal-local-map` overrides it. Extents and text properties can specify an alternative local map for certain parts of the buffer; see Section 40.10 [Extents and Events], page 597.

Each minor mode may have a keymap; if it does, the keymap is active when the minor mode is enabled.

The variable `overriding-local-map` and `overriding-terminal-local-map`, if non-`nil`, specify other local keymaps that override the buffer's local map and all the minor mode keymaps.

All the active keymaps are used together to determine what command to execute when a key is entered. XEmacs searches these maps one by one, in order of decreasing precedence, until it finds a binding in one of the maps.

More specifically:

For key-presses, the order of keymaps searched is:

- the `keymap` property of any extent(s) or text properties at point;
- any applicable minor-mode maps;
- the current local map of the current buffer;
- the current global map.

For mouse-clicks, the order of keymaps searched is:

- the current local map of the `mouse-grabbed-buffer` if any;
- the `keymap` property of any extent(s) at the position of the click (this includes modeline extents);
- the `modeline-map` of the buffer corresponding to the modeline under the mouse (if the click happened over a modeline);
- the value of `toolbar-map` in the current buffer (if the click happened over a toolbar);
- the current local map of the buffer under the mouse (does not apply to toolbar clicks);
- any applicable minor-mode maps;
- the current global map.

Note that if `overriding-local-map` or `overriding-terminal-local-map` is non-`nil`, *only* those two maps and the current global map are searched.

The procedure for searching a single keymap is called *key lookup*; see Section 20.8 [Key Lookup], page 324.

Since every buffer that uses the same major mode normally uses the same local keymap, you can think of the keymap as local to the mode. A change to the local keymap of a buffer (using `local-set-key`, for example) is seen also in the other buffers that share that keymap.

The local keymaps that are used for Lisp mode, 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 Section 18.1 [Intro to Minibuffers], page 261.

See Appendix E [Standard Keymaps], page 781, for a list of standard keymaps.

current-keymaps &optional *event-or-keys* Function

This function returns a list of the current keymaps that will be searched for bindings. This lists keymaps such as the current local map and the minor-mode maps, but does not list the parents of those keymaps. *event-or-keys* controls which keymaps will be listed. If *event-or-keys* is a mouse event (or a vector whose last element is a mouse event), the keymaps for that mouse event will be listed. Otherwise, the keymaps for key presses will be listed.

global-map Variable

This variable contains the default global keymap that maps XEmacs keyboard input to commands. The global keymap is normally this keymap. The default global keymap is a full keymap that binds `self-insert-command` to all of the printing characters.

It is normal practice to change the bindings in the global map, but you should not assign this variable any value other than the keymap it starts out with.

current-global-map Function

This function returns the current global keymap. This is the same as the value of `global-map` unless you change one or the other.

```
(current-global-map)
⇒ #<keymap global-map 639 entries 0x221>
```

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) has a number of entries, including one prefix key, *C-x*.

```
(current-local-map)
⇒ #<keymap lisp-interaction-mode-map 5 entries 0x558>
(describe-bindings-internal (current-local-map))
⇒  ; Inserted into the buffer:
backspace backward-delete-char-untabify
linefeed eval-print-last-sexp
delete delete-char
C-j eval-print-last-sexp
C-x << Prefix Command >>
M-tab lisp-complete-symbol
M-; lisp-indent-for-comment
M-C-i lisp-complete-symbol
M-C-q indent-sexp
M-C-x eval-defun
Alt-backspace backward-kill-sexp
Alt-delete kill-sexp

C-x x edebug-defun
```

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* &optional *buffer* Function

This function makes *keymap* the new local keymap of *buffer*. *buffer* defaults to the current buffer. If *keymap* is `nil`, then the buffer has no local keymap. `use-local-map` returns `nil`. Most major mode commands use this function.

minor-mode-map-alist Variable

This variable is an alist describing keymaps that may or may not be active according to the values of certain variables. Its elements look like this:

```
(variable . keymap)
```

The keymap *keymap* is active whenever *variable* has a non-`nil` value. Typically *variable* is the variable that enables or disables a minor mode. See Section 26.2.2 [Keymaps and Minor Modes], page 372.

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.

See also `minor-mode-key-binding`, above. See Section 26.2.2 [Keymaps and Minor Modes], page 372, for more information about minor modes.

modeline-map Variable

This variable holds the keymap consulted for mouse-clicks on the modeline of a window. This variable may be buffer-local; its value will be looked up in the buffer of the window whose modeline was clicked upon.

toolbar-map Variable

This variable holds the keymap consulted for mouse-clicks over a toolbar.

mouse-grabbed-buffer Variable

If non-`nil`, a buffer which should be consulted first for all mouse activity. When a mouse-click is processed, it will first be looked up in the local-map of this buffer, and then through the normal mechanism if there is no binding for that click. This buffer's value of `mode-motion-hook` will be consulted instead of the `mode-motion-hook` of the buffer of the window under the mouse. You should *bind* this, not set it.

overriding-local-map Variable

If non-`nil`, this variable holds a keymap to use instead of the buffer's local keymap and instead of all the minor mode keymaps. This keymap, if any, overrides all other maps that would have been active, except for the current global map.

overriding-terminal-local-map Variable

If non-`nil`, this variable holds a keymap to use instead of the buffer's local keymap and instead of all the minor mode keymaps, but for the selected console only. (In other words, this variable is always console-local; putting a keymap here only applies to keystrokes coming from the selected console. See Chapter 33 [Consoles and Devices], page 483.) This keymap, if any, overrides all other maps that would have been active, except for the current global map.

20.8 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 type of each event in the key sequence; the rest of the event is ignored. In fact, a key sequence used for key lookup may designate mouse events with just their types (symbols) instead of with entire mouse events (lists). See Section 19.5 [Events], page 290. 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 found by looking up an event type in a keymap. (This doesn't include the item string and other extra elements in menu key bindings because `lookup-key` and other key lookup functions don't include them in the returned value.) While any Lisp object may be stored in a keymap as a keymap entry, not all make sense for key lookup. Here is a 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, and has no default binding, that is equivalent to a binding of `nil` for that event 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. See Section 11.1 [What Is a Function], page 163.

array
: The array (either a string or a vector) is a keyboard macro. The events used so far in the lookup form a complete key, and the array is its binding. See Section 19.13 [Keyboard Macros], page 313, for more information. (Note that you cannot use a shortened form of a key sequence here, such as `(control y)`; you must use the full form `[(control y)]`. See Section 20.5 [Key Sequences], page 317.)

list
: The meaning of a list depends on the types of the elements of the list.

  - If the CAR of *list* is `lambda`, then the list is a lambda expression. This is presumed to be a command, and is treated as such (see above).
  - If the CAR of *list* is a keymap and the CDR is an event type, then this is an *indirect entry*:

    ```
    (othermap . othertype)
    ```

    When key lookup encounters an indirect entry, it looks up instead the binding of *othertype* in *othermap* and uses that.

This feature permits you to define one key as an alias for another key. For example, an entry whose CAR is the keymap called `esc-map` and whose CDR is 32 (the code for ⟨SPC⟩) means, "Use the global binding of *Meta*-⟨SPC⟩, whatever that may be."

symbol The function definition of *symbol* is used in place of *symbol*. If that too is a symbol, then this process is repeated, any number of times. Ultimately this should lead to an object that is a keymap, a command or a keyboard macro. A list is allowed if it is a keymap or a command, but indirect entries are not understood when found via symbols.

Note that keymaps and keyboard macros (strings and vectors) are not valid functions, so a symbol with a keymap, string, or vector as its function definition is invalid as a function. It is, however, valid as a key binding. If the definition is a keyboard macro, then the symbol is also valid as an argument to `command-execute` (see Section 19.3 [Interactive Call], page 286).

The symbol `undefined` is worth special mention: it means to treat the key as undefined. Strictly speaking, the key is defined, and its binding is the command `undefined`; but that command does the same thing that is done automatically for an undefined key: it rings the bell (by calling `ding`) but does not signal an error.

`undefined` is used in local keymaps to override a global key binding and make the key "undefined" locally. A local binding of `nil` would fail to do this because it would not override the global binding.

anything else

If any other type of object is found, the events used so far in the lookup form a complete key, and the object is its binding, but the binding is not executable as a command.

In short, a keymap entry may be a keymap, a command, a keyboard macro, a symbol that leads to one of them, or an indirection or `nil`.

20.9 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 you explicitly ask about them.

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* begins with the character whose value is contained in `meta-prefix-char`, that character is implicitly removed and the ⟨META⟩ modifier added to the key. Thus, the first example below is handled by conversion into the second example.

```
(lookup-key (current-global-map) "\ef")
    ⇒ forward-word
(lookup-key (current-global-map) "\M-f")
    ⇒ forward-word
```

Unlike `read-key-sequence`, this function does not modify the specified events in ways that discard information (see Section 19.6.1 [Key Sequence Input], page 301). In particular, it does not convert letters to lower case.

undefined Command

Used in keymaps to undefine keys. If a key sequence is defined to this, invoking this key sequence causes a "key undefined" error, just as if the key sequence had no binding.

key-binding *key* &optional *accept-defaults* Function

This function returns the binding for *key* in the current keymaps, trying all the active keymaps. The result is `nil` if *key* is undefined in the keymaps.

The argument *accept-defaults* controls checking for default bindings, as in `lookup-key` (above).

```
(key-binding "\C-x\C-f")
    ⇒ find-file
(key-binding '(control home))
    ⇒ beginning-of-buffer
(key-binding [escape escape escape])
    ⇒ keyboard-escape-quit
```

local-key-binding *key* &optional *accept-defaults* Function

This function returns the binding for *key* in the current local keymap, or `nil` if it is undefined there.

The argument *accept-defaults* controls checking for default bindings, as in `lookup-key` (above).

global-key-binding *key* &optional *accept-defaults* Function

This function returns the binding for command *key* in the current global keymap, or `nil` if it is undefined there.

The argument *accept-defaults* controls checking for default bindings, as in `lookup-key` (above).

minor-mode-key-binding *key* &optional *accept-defaults* Function

This function returns a list of all the active minor mode bindings of *key*. More precisely, it returns an alist of pairs `(`*modename* `.` *binding*`)`, where *modename* is the variable that enables the minor mode, and *binding* is *key*'s binding in that mode. If *key* has no minor-mode bindings, the value is `nil`.

If the first binding is not a prefix command, 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 two-character sequence to a meta character so it can be looked up in a keymap. For useful results, the value should be a prefix event (see Section 20.6 [Prefix Keys], page 319). The default value is `?\^[` (integer 27), which is the ASCII character usually produced by the ⟨ESC⟩ key.

As long as the value of `meta-prefix-char` remains `?\^[`, key lookup translates ⟨ESC⟩ *b* into *M-b*, which is normally defined as the `backward-word` command. However, if you set `meta-prefix-char` to `?\^X` (i.e. the keystroke *C-x*) or its equivalent ASCII code 24, then XEmacs will translate *C-x b* (whose standard binding is the `switch-to-buffer` command) into *M-b*.

```
meta-prefix-char                    ; The default value.
     ⇒ ?\^[   ; Under XEmacs 20.
     ⇒ 27     ; Under XEmacs 19.
(key-binding "\eb")
     ⇒ backward-word
?\C-x                               ; The print representation
                                    ;     of a character.
     ⇒ ?\^X   ; Under XEmacs 20.
     ⇒ 24     ; Under XEmacs 19.
(setq meta-prefix-char 24)
     ⇒ 24
(key-binding "\C-xb")
     ⇒ backward-word              ; Now, typing C-x b is
                                    ;     like typing M-b.

(setq meta-prefix-char ?\e)         ; Avoid confusion!
                                    ; Restore the default value!
     ⇒ ?\^[   ; Under XEmacs 20.
     ⇒ 27     ; Under XEmacs 19.
```

20.10 Changing Key Bindings

The way to rebind a key is to change its entry in a keymap. If you change a binding in the global keymap, the change is effective in all buffers (though it has no direct effect in buffers that shadow the global binding with a local one). If you change the current buffer's local map, that usually affects all buffers using the same major mode. The `global-set-key` and `local-set-key` functions are convenient interfaces for these operations (see

Section 20.11 [Key Binding Commands], page 330). You can also use `define-key`, a more general function; then you must specify explicitly the map to change.

The way to specify the key sequence that you want to rebind is described above (see Section 20.5 [Key Sequences], page 317).

For the functions below, an error is signaled if *keymap* is not a keymap or if *key* is not a string or vector representing a key sequence. You can use event types (symbols) as shorthand for events that are lists.

define-key *keymap key binding* Function

This function sets the binding for *key* in *keymap*. (If *key* is more than one event long, the change is actually made in another keymap reached from *keymap*.) The argument *binding* can be any Lisp object, but only certain types are meaningful. (For a list of meaningful types, see Section 20.8 [Key Lookup], page 324.) 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.

Here is an example that creates a sparse keymap and makes a number of bindings in it:

```
(setq map (make-sparse-keymap))
    ⇒ #<keymap 0 entries 0xbee>
(define-key map "\C-f" 'forward-char)
    ⇒ forward-char
map
    ⇒ #<keymap 1 entry 0xbee>
(describe-bindings-internal map)
⇒   ; (Inserted in buffer)
C-f             forward-char

;; Build sparse submap for C-x and bind f in that.
(define-key map "\C-xf" 'forward-word)
    ⇒ forward-word
map
    ⇒ #<keymap 2 entries 0xbee>
(describe-bindings-internal map)
⇒   ; (Inserted in buffer)
C-f             forward-char
C-x             << Prefix Command >>

C-x f           forward-word

;; Bind C-p to the ctl-x-map.
(define-key map "\C-p" ctl-x-map)
;; ctl-x-map
⇒ #<keymap Control-X-prefix 77 entries 0x3bf>
```

```
;; Bind C-f to foo in the ctl-x-map.
(define-key map "\C-p\C-f" 'foo)
⇒ foo
map
    ⇒ #<keymap 3 entries 0xbee>
(describe-bindings-internal map)
⇒   ; (Inserted in buffer)
C-f             forward-char
C-p             << Prefix command Control-X-prefix >>
C-x             << Prefix Command >>

C-p tab         indent-rigidly
C-p $           set-selective-display
C-p '           expand-abbrev
C-p (           start-kbd-macro
C-p )           end-kbd-macro
   ...
C-p C-x         exchange-point-and-mark
C-p C-z         suspend-or-iconify-emacs
C-p M-escape    repeat-complex-command
C-p M-C-[       repeat-complex-command

C-x f           forward-word

C-p 4 .         find-tag-other-window
   ...
C-p 4 C-o       display-buffer

C-p 5 0         delete-frame
   ...
C-p 5 C-f       find-file-other-frame

   ...

C-p a i g       inverse-add-global-abbrev
C-p a i l       inverse-add-mode-abbrev
```

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 XEmacs 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.

suppress-keymap *keymap* &optional *nodigits* Function

This function changes the contents of the full keymap *keymap* by making all the printing characters undefined. More precisely, it binds them to the command `undefined`. This makes ordinary insertion of text impossible. `suppress-keymap` returns `nil`.

If *nodigits* is `nil`, then `suppress-keymap` defines digits to run `digit-argument`, and `-` to run `negative-argument`. Otherwise it makes them undefined like the rest of the printing characters.

The `suppress-keymap` function does not make it impossible to modify a buffer, as it does not suppress commands such as `yank` and `quoted-insert`. To prevent any modification of a buffer, make it read-only (see Section 30.7 [Read Only Buffers], page 438).

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 XEmacs.

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)
...
```

20.11 Commands for Binding Keys

This section describes some convenient interactive interfaces for changing key bindings. They work by calling `define-key`.

People often use `global-set-key` in their '`.emacs`' file for simple customization. For example,

```
(global-set-key "\C-x\C-\\" 'next-line)
```

or

```
(global-set-key [(control ?x) (control ?\\)] 'next-line)
```

or

```
(global-set-key [?\C-x ?\C-\\] 'next-line)
```

redefines `C-x C-\` to move down a line.

```
(global-set-key [(meta button1)] 'mouse-set-point)
```

redefines the first (leftmost) mouse button, typed with the Meta key, to set point where you click.

global-set-key *key definition* Command

This function sets the binding of *key* in the current global map to *definition*.

```
(global-set-key key definition)
≡
(define-key (current-global-map) key definition)
```

global-unset-key *key* Command

This function removes the binding of *key* from the current global map.

One use of this function is in preparation for defining a longer key that uses *key* as a prefix—which would not be allowed if *key* has a non-prefix binding. For example:

```
(global-unset-key "\C-l")
    ⇒ nil
(global-set-key "\C-l\C-l" 'redraw-display)
    ⇒ nil
```

This function is implemented simply using `define-key`:

```
(global-unset-key key)
≡
(define-key (current-global-map) key nil)
```

local-set-key *key definition* Command

This function sets the binding of *key* in the current local keymap to *definition*.

```
(local-set-key key definition)
≡
(define-key (current-local-map) key definition)
```

local-unset-key *key* Command

This function removes the binding of *key* from the current local map.

```
(local-unset-key key)
≡
(define-key (current-local-map) key nil)
```

20.12 Scanning Keymaps

This section describes functions used to scan all the current keymaps, or all keys within a keymap, 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 `C-x`, which is displayed as '`[(control x)]`', is a prefix key whose definition is the keymap `#<keymap ((control x) #<keymap emacs-lisp-mode-map 8 entries 0x546>) 1 entry 0x8a2>`. (The strange notation for the keymap's name indicates that this is an internal submap of `emacs-lisp-mode-map`. This is because `lisp-interaction-mode-map` has set up `emacs-lisp-mode-map` as its parent, and `lisp-interaction-mode-map` defines no key sequences beginning with `C-x`.)

```
(current-local-map)
⇒ #<keymap lisp-interaction-mode-map 5 entries 0x558>
(accessible-keymaps (current-local-map))
⇒(([] . #<keymap lisp-interaction-mode-map 5 entries 0x558>)
    ([(control x)] .
     #<keymap ((control x) #<keymap emacs-lisp-mode-map 8 entries 0x546>)
              1 entry 0x8a2>))
```

The following example shows the results of calling `accessible-keymaps` on a large, complex keymap. Notice how some keymaps were given explicit names using `set-keymap-name`; those submaps without explicit names are given descriptive names indicating their relationship to their enclosing keymap.

```
(accessible-keymaps (current-global-map))
⇒ (([] . #<keymap global-map 639 entries 0x221>)
   ([(control c)] . #<keymap mode-specific-command-prefix 1 entry 0x3cb>)
   ([(control h)] . #<keymap help-map 33 entries 0x4ec>)
   ([(control x)] . #<keymap Control-X-prefix 77 entries 0x3bf>)
   ([(meta escape)] .
      #<keymap ((meta escape) #<keymap global-map 639 entries 0x221>)
               3 entries 0x3e0>)
   ([(meta control \[)] .
      #<keymap ((meta escape) #<keymap global-map 639 entries 0x221>)
               3 entries 0x3e0>)
   ([f1] . #<keymap help-map 33 entries 0x4ec>)
   ([(control x) \4] . #<keymap ctl-x-4-prefix 9 entries 0x3c5>)
   ([(control x) \5] . #<keymap ctl-x-5-prefix 8 entries 0x3c8>)
   ([(control x) \6] . #<keymap 13 entries 0x4d2>)
   ([(control x) a] .
      #<keymap (a #<keymap Control-X-prefix 77 entries 0x3bf>)
               8 entries 0x3ef>)
   ([(control x) n] . #<keymap narrowing-prefix 3 entries 0x3dd>)
   ([(control x) r] . #<keymap rectangle-prefix 18 entries 0x3e9>)
   ([(control x) v] . #<keymap vc-prefix-map 13 entries 0x60e>)
   ([(control x) a i] .
     #<keymap (i #<keymap (a #<keymap Control-X-prefix 77 entries 0x3bf>)
                          8 entries 0x3ef>)
              2 entries 0x3f5>))
```

map-keymap *function keymap* &optional *sort-first* — Function

This function applies *function* to each element of `KEYMAP`. *function* will be called with two arguments: a key-description list, and the binding. The order in which the elements of the keymap are passed to the function is unspecified. If the function inserts new elements into the keymap, it may or may not be called with them later. No element of the keymap will ever be passed to the function more than once.

The function will not be called on elements of this keymap's parents (see Section 20.4 [Inheritance and Keymaps], page 317) or upon keymaps which are contained within this keymap (multi-character definitions). It will be called on ⟨META⟩ characters since they are not really two-character sequences.

If the optional third argument *sort-first* is non-`nil`, then the elements of the keymap will be passed to the mapper function in a canonical order. Otherwise, they will be passed in hash (that is, random) order, which is faster.

keymap-fullness *keymap* — Function

This function returns the number of bindings in the keymap.

where-is-internal *definition* &optional *keymaps firstonly noindirect event-or-keys* — Function

This function returns a list of key sequences (of any length) that are bound to *definition* in a set of keymaps.

The argument *definition* can be any object; it is compared with all keymap entries using `eq`.

KEYMAPS can be either a keymap (meaning search in that keymap and the current global keymap) or a list of keymaps (meaning search in exactly those keymaps and no others). If KEYMAPS is nil, search in the currently applicable maps for EVENT-OR-KEYS.

If *keymap* is a keymap, then the maps searched are *keymap* and the global keymap. If *keymap* is a list of keymaps, then the maps searched are exactly those keymaps, and no others. If *keymap* is `nil`, then the maps used are the current active keymaps for *event-or-keys* (this is equivalent to specifying `(current-keymaps` *event-or-keys*`)` as the argument to *keymaps*).

If *firstonly* is non-`nil`, then the value is a single vector representing the first key sequence found, rather than a list of all possible key sequences.

If *noindirect* is non-`nil`, `where-is-internal` doesn't follow indirect keymap bindings. This makes it possible to search for an indirect definition itself.

This function is used by `where-is` (see section "Help" in *The XEmacs Reference Manual*).

```
(where-is-internal 'describe-function)
    ⇒ ([(control h) d] [(control h) f] [f1 d] [f1 f])
```

describe-bindings-internal *map* &optional *all shadow prefix mouse-only-p* Function

This function inserts (into the current buffer) a list of all defined keys and their definitions in *map*. Optional second argument *all* says whether to include even "uninteresting" definitions, i.e. symbols with a non-`nil` `suppress-keymap` property. Third argument *shadow* is a list of keymaps whose bindings shadow those of map; if a binding is present in any shadowing map, it is not printed. Fourth argument *prefix*, if non-`nil`, should be a key sequence; only bindings which start with that key sequence will be printed. Fifth argument *mouse-only-p* says to only print bindings for mouse clicks.

`describe-bindings-internal` is used to implement the help command `describe-bindings`.

describe-bindings *prefix mouse-only-p* Command

This function creates a listing of all defined keys and their definitions. It writes the listing in a buffer named '`*Help*`' and displays it in a window.

If *prefix* is non-`nil`, it should be a prefix key; then the listing includes only keys that start with *prefix*.

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 the second argument (prefix arg, interactively) is non-`nil` then only the mouse bindings are displayed.

20.13 Other Keymap Functions

set-keymap-prompt *keymap new-prompt* Function

This function sets the "prompt" of *keymap* to string *new-prompt*, or `nil` if no prompt is desired. The prompt is shown in the echo-area when reading a key-sequence to be looked-up in this keymap.

keymap-prompt *keymap* &optional *use-inherited* Function

This function returns the "prompt" of the given keymap. If *use-inherited* is non-`nil`, any parent keymaps will also be searched for a prompt.

21 Menus

21.1 Format of Menus

A menu is described using a *menu description*, which is a list of menu items, keyword-value pairs, strings, and submenus. The menu description specifies which items are present in the menu, what function each item invokes, and whether the item is selectable or not. Pop-up menus are directly described with a menu description, while menubars are described slightly differently (see below).

The first element of a menu must be a string, which is the name of the menu. This is the string that will be displayed in the parent menu or menubar, if any. This string is not displayed in the menu itself, except in the case of the top level pop-up menu, where there is no parent. In this case, the string will be displayed at the top of the menu if `popup-menu-titles` is non-`nil`.

Immediately following the first element there may optionally be up to four keyword-value pairs, as follows:

`:included` *form*
: This can be used to control the visibility of a menu. The form is evaluated and the menu will be omitted if the result is `nil`.

`:config` *symbol*
: This is an efficient shorthand for `:included (memq` *symbol* `menubar-configuration)`. See the variable `menubar-configuration`.

`:filter` *function*
: A menu filter is used to sensitize or incrementally create a submenu only when it is selected by the user and not every time the menubar is activated. The filter function is passed the list of menu items in the submenu and must return a list of menu items to be used for the menu. It is called only when the menu is about to be displayed, so other menus may already be displayed. Vile and terrible things will happen if a menu filter function changes the current buffer, window, or frame. It also should not raise, lower, or iconify any frames. Basically, the filter function should have no side-effects.

`:accelerator` *key*
: A menu accelerator is a keystroke which can be pressed while the menu is visible which will immediately activate the item. *key* must be a char or the symbol name of a key. See Section 21.7 [Menu Accelerators], page 345.

The rest of the menu consists of elements as follows:

- A *menu item*, which is a vector in the following form:

  `[` *name callback :keyword value :keyword value* `. . . ]`

  name is a string, the name of the menu item; it is the string to display on the menu. It is filtered through the resource database, so it is possible for resources to override what string is actually displayed.

callback is a form that will be invoked when the menu item is selected. If the callback of a menu item is a symbol, then it must name a command. It will be invoked with `call-interactively`. If it is a list, then it is evaluated with `eval`.

The valid keywords and their meanings are described below.

Note that for compatibility purposes, the form

`[` *name callback active-p* `]`

is also accepted and is equivalent to

`[` *name callback* `:active` *active-p* `]`

and the form

`[` *name callback active-p suffix*`]`

is accepted and is equivalent to

`[` *name callback* `:active` *active-p* `:suffix` *suffix*`]`

However, these older forms are deprecated and should generally not be used.

- If an element of a menu is a string, then that string will be presented in the menu as unselectable text.
- If an element of a menu is a string consisting solely of hyphens, then that item will be presented as a solid horizontal line.
- If an element of a menu is a string beginning with '`--:`', then a particular sort of horizontal line will be displayed, as follows:

'`"--:singleLine"`'
: A solid horizontal line. This is equivalent to a string consisting solely of hyphens.

'`"--:doubleLine"`'
: A solid double horizontal line.

'`"--:singleDashedLine"`'
: A dashed horizontal line.

'`"--:doubleDashedLine"`'
: A dashed double horizontal line.

'`"--:noLine"`'
: No line (but a small space is left).

'`"--:shadowEtchedIn"`'
: A solid horizontal line with a 3-d recessed appearance.

'`"--:shadowEtchedOut"`'
: A solid horizontal line with a 3-d pushed-out appearance.

'`"--:shadowDoubleEtchedIn"`'
: A solid double horizontal line with a 3-d recessed appearance.

'`"--:shadowDoubleEtchedOut"`'
: A solid double horizontal line with a 3-d pushed-out appearance.

'`"--:shadowEtchedInDash"`'
: A dashed horizontal line with a 3-d recessed appearance.

'"--:shadowEtchedOutDash"'
: A dashed horizontal line with a 3-d pushed-out appearance.

'"--:shadowDoubleEtchedInDash"'
: A dashed double horizontal line with a 3-d recessed appearance.

'"--:shadowDoubleEtchedOutDash"'
: A dashed double horizontal line with a 3-d pushed-out appearance.

- If an element of a menu is a list, it is treated as a submenu. The name of that submenu (the first element in the list) will be used as the name of the item representing this menu on the parent.

The possible keywords are as follows:

:active *form*
: *form* will be evaluated when the menu that this item is a part of is about to be displayed, and the item will be selectable only if the result is non-`nil`. If the item is unselectable, it will usually be displayed grayed-out to indicate this.

:suffix *form*
: *form* will be evaluated when the menu that this item is a part of is about to be displayed, and the resulting string is appended to the displayed name. This provides a convenient way of adding the name of a command's "argument" to the menu, like '`Kill Buffer NAME`'.

:keys *string*
: Normally, the keyboard equivalents of commands in menus are displayed when the "callback" is a symbol. This can be used to specify keys for more complex menu items. It is passed through `substitute-command-keys` first.

:style *style*
: Specifies what kind of object this menu item is. *style* be one of the symbols

  `nil` A normal menu item.

  `toggle` A toggle button.

  `radio` A radio button.

  `button` A menubar button.

  The only difference between toggle and radio buttons is how they are displayed. But for consistency, a toggle button should be used when there is one option whose value can be turned on or off, and radio buttons should be used when there is a set of mutually exclusive options. When using a group of radio buttons, you should arrange for no more than one to be marked as selected at a time.

:selected *form*
: Meaningful only when *style* is `toggle`, `radio` or `button`. This specifies whether the button will be in the selected or unselected state. *form* is evaluated, as for `:active`.

:included *form*
: This can be used to control the visibility of a menu item. The form is evaluated and the menu item is only displayed if the result is non-`nil`. Note that this is different from `:active`: If `:active` evaluates to `nil`, the item will be displayed grayed out, while if `:included` evaluates to `nil`, the item will be omitted entirely.

:config *symbol*
: This is an efficient shorthand for `:included (memq` *symbol* `menubar-configuration)`. See the variable `menubar-configuration`.

:accelerator *key*
: A menu accelerator is a keystroke which can be pressed while the menu is visible which will immediately activate the item. *key* must be a char or the symbol name of a key. See Section 21.7 [Menu Accelerators], page 345.

menubar-configuration Variable

This variable holds a list of symbols, against which the value of the `:config` tag for each menubar item will be compared. If a menubar item has a `:config` tag, then it is omitted from the menubar if that tag is not a member of the `menubar-configuration` list.

For example:

```
("File"
 :filter file-menu-filter ; file-menu-filter is a function that takes
; one argument (a list of menu items) and
; returns a list of menu items
 [ "Save As..."    write-file]
 [ "Revert Buffer" revert-buffer :active (buffer-modified-p) ]
 [ "Read Only"     toggle-read-only :style toggle :selected buffer-read-only ]
 )
```

21.2 Format of the Menubar

A menubar is a list of menus, menu items, and strings. The format is similar to that of a menu, except:

- The first item need not be a string, and is not treated specially.
- A string consisting solely of hyphens is not treated specially.
- If an element of a menubar is `nil`, then it is used to represent the division between the set of menubar items which are flush-left and those which are flush-right. (Note: this isn't completely implemented yet.)

21.3 Menubar

current-menubar Variable

This variable holds the description of the current menubar. This may be buffer-local. When the menubar is changed, the function `set-menubar-dirty-flag` has to be called in order for the menubar to be updated on the screen.

default-menubar Constant

This variable holds the menubar description of the menubar that is visible at startup. This is the value that `current-menubar` has at startup.

set-menubar-dirty-flag Function

This function tells XEmacs that the menubar widget has to be updated. Changes to the menubar will generally not be visible until this function is called.

The following convenience functions are provided for setting the menubar. They are equivalent to doing the appropriate action to change `current-menubar`, and then calling `set-menubar-dirty-flag`. Note that these functions copy their argument using `copy-sequence`.

set-menubar *menubar* Function

This function sets the default menubar to be *menubar* (see Section 21.1 [Menu Format], page 337). This is the menubar that will be visible in buffers that have not defined their own, buffer-local menubar.

set-buffer-menubar *menubar* Function

This function sets the buffer-local menubar to be *menubar*. This does not change the menubar in any buffers other than the current one.

Miscellaneous:

menubar-show-keybindings Variable

If true, the menubar will display keyboard equivalents. If false, only the command names will be displayed.

activate-menubar-hook Variable

Function or functions called before a menubar menu is pulled down. These functions are called with no arguments, and should interrogate and modify the value of `current-menubar` as desired.

The functions on this hook are invoked after the mouse goes down, but before the menu is mapped, and may be used to activate, deactivate, add, or delete items from the menus. However, using a filter (with the `:filter` keyword in a menu description) is generally a more efficient way of accomplishing the same thing, because the filter is invoked only when the actual menu goes down. With a complex menu, there can be a quite noticeable and sometimes aggravating delay if all menu modification is implemented using the `activate-menubar-hook`. See above.

These functions may return the symbol `t` to assert that they have made no changes to the menubar. If any other value is returned, the menubar is recomputed. If `t` is returned but the menubar has been changed, then the changes may not show up right away. Returning `nil` when the menubar has not changed is not so bad; more computation will be done, but redisplay of the menubar will still be performed optimally.

menu-no-selection-hook Variable

Function or functions to call when a menu or dialog box is dismissed without a selection having been made.

21.4 Modifying Menus

The following functions are provided to modify the menubar of one of its submenus. Note that these functions modify the menu in-place, rather than copying it and making a new menu.

Some of these functions take a *menu path*, which is a list of strings identifying the menu to be modified. For example, `("File")` names the top-level "File" menu. `("File" "Foo")` names a hypothetical submenu of "File".

Others take a *menu item path*, which is similar to a menu path but also specifies a particular item to be modified. For example, `("File" "Save")` means the menu item called "Save" under the top-level "File" menu. `("Menu" "Foo" "Item")` means the menu item called "Item" under the "Foo" submenu of "Menu".

add-submenu *menu-path submenu* &optional *before* Function

This function adds a menu to the menubar or one of its submenus. If the named menu exists already, it is changed.

menu-path identifies the menu under which the new menu should be inserted. If *menu-path* is `nil`, then the menu will be added to the menubar itself.

submenu is the new menu to add (see Section 21.1 [Menu Format], page 337).

before, if provided, is the name of a menu before which this menu should be added, if this menu is not on its parent already. If the menu is already present, it will not be moved.

add-menu-button *menu-path menu-leaf* &optional *before* Function

This function adds a menu item to some menu, creating the menu first if necessary. If the named item exists already, it is changed.

menu-path identifies the menu under which the new menu item should be inserted.

menu-leaf is a menubar leaf node (see Section 21.1 [Menu Format], page 337).

before, if provided, is the name of a menu before which this item should be added, if this item is not on the menu already. If the item is already present, it will not be moved.

delete-menu-item *menu-item-path* Function

This function removes the menu item specified by *menu-item-path* from the menu hierarchy.

enable-menu-item *menu-item-path* Function

This function makes the menu item specified by *menu-item-path* be selectable.

disable-menu-item *menu-item-path* Function

This function makes the menu item specified by *menu-item-path* be unselectable.

relabel-menu-item *menu-item-path new-name* Function

This function changes the string of the menu item specified by *menu-item-path*. *new-name* is the string that the menu item will be printed as from now on.

The following function can be used to search for a particular item in a menubar specification, given a path to the item.

find-menu-item *menubar menu-item-path* &optional *parent* Function

This function searches *menubar* for the item given by *menu-item-path* starting from *parent* (`nil` means start at the top of *menubar*). This function returns (*item* . *parent*), where *parent* is the immediate parent of the item found (a menu description), and *item* is either a vector, list, or string, depending on the nature of the menu item.

This function signals an error if the item is not found.

The following deprecated functions are also documented, so that existing code can be understood. You should not use these functions in new code.

add-menu *menu-path menu-name menu-items* &optional *before* Function

This function adds a menu to the menubar or one of its submenus. If the named menu exists already, it is changed. This is obsolete; use `add-submenu` instead.

menu-path identifies the menu under which the new menu should be inserted. If *menu-path* is `nil`, then the menu will be added to the menubar itself.

menu-name is the string naming the menu to be added; *menu-items* is a list of menu items, strings, and submenus. These two arguments are the same as the first and following elements of a menu description (see Section 21.1 [Menu Format], page 337).

before, if provided, is the name of a menu before which this menu should be added, if this menu is not on its parent already. If the menu is already present, it will not be moved.

add-menu-item *menu-path item-name function enabled-p* &optional *before* Function

This function adds a menu item to some menu, creating the menu first if necessary. If the named item exists already, it is changed. This is obsolete; use `add-menu-button` instead.

menu-path identifies the menu under which the new menu item should be inserted. *item-name*, *function*, and *enabled-p* are the first, second, and third elements of a menu item vector (see Section 21.1 [Menu Format], page 337).

before, if provided, is the name of a menu item before which this item should be added, if this item is not on the menu already. If the item is already present, it will not be moved.

21.5 Menu Filters

The following filter functions are provided for use in `default-menubar`. You may want to use them in your own menubar description.

file-menu-filter *menu-items* Function

This function changes the arguments and sensitivity of these File menu items:

'`Delete Buffer`'
: Has the name of the current buffer appended to it.

'`Print Buffer`'
: Has the name of the current buffer appended to it.

'`Pretty-Print Buffer`'
: Has the name of the current buffer appended to it.

'`Save Buffer`'
: Has the name of the current buffer appended to it, and is sensitive only when the current buffer is modified.

'`Revert Buffer`'
: Has the name of the current buffer appended to it, and is sensitive only when the current buffer has a file.

'`Delete Frame`'
: Sensitive only when there is more than one visible frame.

edit-menu-filter *menu-items* Function

This function changes the arguments and sensitivity of these Edit menu items:

'`Cut`' Sensitive only when XEmacs owns the primary X Selection (if `zmacs-regions` is `t`, this is equivalent to saying that there is a region selected).

'`Copy`' Sensitive only when XEmacs owns the primary X Selection.

'`Clear`' Sensitive only when XEmacs owns the primary X Selection.

'`Paste`' Sensitive only when there is an owner for the X Clipboard Selection.

'`Undo`' Sensitive only when there is undo information. While in the midst of an undo, this is changed to '`Undo More`'.

buffers-menu-filter *menu-items* Function

This function sets up the Buffers menu. See Section 21.8 [Buffers Menu], page 347, for more information.

21.6 Pop-Up Menus

popup-menu *menu-desc* Function

This function pops up a menu specified by *menu-desc*, which is a menu description (see Section 21.1 [Menu Format], page 337). The menu is displayed at the current mouse position.

popup-menu-up-p Function

This function returns `t` if a pop-up menu is up, `nil` otherwise.

popup-menu-titles Variable

If true (the default), pop-up menus will have title bars at the top.

Some machinery is provided that attempts to provide a higher-level mechanism onto pop-up menus. This only works if you do not redefine the binding for button3.

popup-mode-menu Command

This function pops up a menu of global and mode-specific commands. The menu is computed by combining `global-popup-menu` and `mode-popup-menu`. This is the default binding for button3. You should generally not change this binding.

global-popup-menu Variable

This holds the global popup menu. This is present in all modes. (This is `nil` by default.)

mode-popup-menu Variable

The mode-specific popup menu. Automatically buffer local. This is appended to the default items in `global-popup-menu`.

default-popup-menu Constant

This holds the default value of `mode-popup-menu`.

activate-popup-menu-hook Variable

Function or functions run before a mode-specific popup menu is made visible. These functions are called with no arguments, and should interrogate and modify the value of `global-popup-menu` or `mode-popup-menu` as desired. Note: this hook is only run if you use `popup-mode-menu` for activating the global and mode-specific commands; if you have your own binding for button3, this hook won't be run.

The following convenience functions are provided for displaying pop-up menus.

popup-buffer-menu *event* Function

This function pops up a copy of the '`Buffers`' menu (from the menubar) where the mouse is clicked.

popup-menubar-menu *event* Function

This function pops up a copy of menu that also appears in the menubar.

21.7 Menu Accelerators

Menu accelerators are keyboard shortcuts for accessing the menubar. Accelerator keys can be specified for menus as well as for menu items. An accelerator key for a menu is used to activate that menu when it appears as a submenu of another menu. An accelerator key for a menu item is used to activate that item.

21.7.1 Creating Menu Accelerators

Menu accelerators are specified as part of the menubar format using the :accelerator tag to specify a key or by placing "%_" in the menu or menu item name prior to the letter which is to be used as the accelerator key. The advantage of the second method is that the menu rendering code then knows to draw an underline under that character, which is the canonical way of indicating an accelerator key to a user.

For example, the command

```
(add-submenu nil '("%_Test"
   ["One" (insert "1") :accelerator ?1 :active t]
   ["%_Two" (insert "2")]
   ["%_3" (insert "3")]))
```

will add a new menu to the top level menubar. The new menu can be reached by pressing "t" while the top level menubar is active. When the menu is active, pressing "1" will activate the first item and insert the character "1" into the buffer. Pressing "2" will activate the second item and insert the character "2" into the buffer. Pressing "3" will activate the third item and insert the character "3" into the buffer.

It is possible to activate the top level menubar itself using accelerator keys. See Section 21.7.3 [Menu Accelerator Functions], page 346.

21.7.2 Keyboard Menu Traversal

In addition to immediately activating a menu or menu item, the keyboard can be used to traverse the menus without activating items. The keyboard arrow keys, the return key and the escape key are defined to traverse the menus in a way that should be familiar to users of any of a certain family of popular PC operating systems.

This behavior can be changed by modifying the bindings in menu-accelerator-map. At this point, the online help is your best bet for more information about how to modify the menu traversal keys.

21.7.3 Menu Accelerator Functions

accelerate-menu Function

Make the menubar immediately active and place the cursor on the left most entry in the top level menu. Menu items can be selected as usual.

menu-accelerator-enabled Variable

Whether menu accelerator keys can cause the menubar to become active.

If `menu-force` or `menu-fallback`, then menu accelerator keys can be used to activate the top level menu. Once the menubar becomes active, the accelerator keys can be used regardless of the value of this variable.

`menu-force` is used to indicate that the menu accelerator key takes precedence over bindings in the current keymap(s). `menu-fallback` means that bindings in the current keymap take precedence over menu accelerator keys. Thus a top level menu with an accelerator of "T" would be activated on a keypress of Meta-t if *menu-accelerator-enabled* is `menu-force`. However, if *menu-accelerator-enabled* is `menu-fallback`, then Meta-t will not activate the menubar and will instead run the function transpose-words, to which it is normally bound.

The default value is `nil`.

See also *menu-accelerator-modifiers* and *menu-accelerator-prefix*.

menu-accelerator-map Variable

Keymap consulted to determine the commands to run in response to keypresses occurring while the menubar is active. See Section 21.7.2 [Keyboard Menu Traversal], page 346.

menu-accelerator-modifiers Variable

A list of modifier keys which must be pressed in addition to a valid menu accelerator in order for the top level menu to be activated in response to a keystroke. The default value of `(meta)` mirrors the usage of the alt key as a menu accelerator in popular PC operating systems.

The modifier keys in *menu-accelerator-modifiers* must match exactly the modifiers present in the keypress. The only exception is that the shift modifier is accepted in conjunction with alphabetic keys even if it is not a menu accelerator modifier.

See also *menu-accelerator-enabled* and *menu-accelerator-prefix*.

menu-accelerator-prefix Variable

Prefix key(s) that must be typed before menu accelerators will be activated. Must be a valid key descriptor.

The default value is `nil`.

```
(setq menu-accelerator-prefix ?\C-x)
(setq menu-accelerator-modifiers '(meta control))
(setq menu-accelerator-enabled 'menu-force)
(add-submenu nil '("%_Test"
   ["One" (insert "1") :accelerator ?1 :active t]
   ["%_Two" (insert "2")]
   ["%_3" (insert "3")]))
```

will add the menu "Test" to the top level menubar. Pressing C-x followed by C-M-T will activate the menubar and display the "Test" menu. Pressing C-M-T by itself will not activate the menubar. Neither will pressing C-x followed by anything else.

21.8 Buffers Menu

The following options control how the '`Buffers`' menu is displayed. This is a list of all (or a subset of) the buffers currently in existence, and is updated dynamically.

buffers-menu-max-size User Option

This user option holds the maximum number of entries which may appear on the '`Buffers`' menu. If this is 10, then only the ten most-recently-selected buffers will be shown. If this is `nil`, then all buffers will be shown. Setting this to a large number or `nil` will slow down menu responsiveness.

format-buffers-menu-line *buffer* Function

This function returns a string to represent *buffer* in the '`Buffers`' menu. `nil` means the buffer shouldn't be listed. You can redefine this.

complex-buffers-menu-p User Option

If true, the '`Buffers`' menu will contain several commands, as submenus of each buffer line. If this is false, then there will be only one command: select that buffer.

buffers-menu-switch-to-buffer-function User Option

This user option holds the function to call to select a buffer from the '`Buffers`' menu. `switch-to-buffer` is a good choice, as is `pop-to-buffer`.

22 Dialog Boxes

22.1 Dialog Box Format

A dialog box description is a list.

- The first element of the list is a string to display in the dialog box.
- The rest of the elements are descriptions of the dialog box's buttons. Each one is a vector of three elements:
  - The first element is the text of the button.
  - The second element is the *callback*.
  - The third element is `t` or `nil`, whether this button is selectable.

If the callback of a button is a symbol, then it must name a command. It will be invoked with `call-interactively`. If it is a list, then it is evaluated with `eval`.

One (and only one) of the buttons may be `nil`. This marker means that all following buttons should be flushright instead of flushleft.

The syntax, more precisely:

```
form :=  <something to pass to `eval'>
command :=  <a symbol or string, to pass to `call-interactively'>
callback  :=  command | form
active-p :=  <t, nil, or a form to evaluate to decide whether this
 button should be selectable>
name :=  <string>
partition :=  'nil'
button :=  '['  name callback active-p ']'
dialog :=  '(' name [ button ]+ [ partition [ button ]+ ] ')'
```

22.2 Dialog Box Functions

popup-dialog-box *dbox-desc* Function

This function pops up a dialog box. *dbox-desc* describes how the dialog box will appear (see Section 22.1 [Dialog Box Format], page 349).

See Section 18.6 [Yes-or-No Queries], page 275, for functions to ask a yes/no question using a dialog box.

23 Toolbar

23.1 Toolbar Intro

A *toolbar* is a bar of icons displayed along one edge of a frame. You can view a toolbar as a series of menu shortcuts – the most common menu options can be accessed with a single click rather than a series of clicks and/or drags to select the option from a menu. Consistent with this, a help string (called the *help-echo*) describing what an icon in the toolbar (called a *toolbar button*) does, is displayed in the minibuffer when the mouse is over the button.

In XEmacs, a toolbar can be displayed along any of the four edges of the frame, and two or more different edges can be displaying toolbars simultaneously. The contents, thickness, and visibility of the toolbars can be controlled separately, and the values can be per-buffer, per-frame, etc., using specifiers (see Chapter 41 [Specifiers], page 599).

Normally, there is one toolbar displayed in a frame. Usually, this is the standard toolbar, but certain modes will override this and substitute their own toolbar. In some cases (e.g. the VM package), a package will supply its own toolbar along a different edge from the standard toolbar, so that both can be visible at once. This standard toolbar is usually positioned along the top of the frame, but this can be changed using `set-default-toolbar-position`.

Note that, for each of the toolbar properties (contents, thickness, and visibility), there is a separate specifier for each of the four toolbar positions (top, bottom, left, and right), and an additional specifier for the "default" toolbar, i.e. the toolbar whose position is controlled by `set-default-toolbar-position`. The way this works is that `set-default-toolbar-position` arranges things so that the appropriate position-specific specifiers for the default position inherit from the corresponding default specifiers. That way, if the position-specific specifier does not give a value (which it usually doesn't), then the value from the default specifier applies. If you want to control the default toolbar, you just change the default specifiers, and everything works. A package such as VM that wants to put its own toolbar in a different location from the default just sets the position-specific specifiers, and if the user sets the default toolbar to the same position, it will just not be visible.

23.2 Toolbar Descriptor Format

The contents of a toolbar are specified using a *toolbar descriptor*. The format of a toolbar descriptor is a list of *toolbar button descriptors*. Each toolbar button descriptor is a vector in one of the following formats:

- `[`*glyph-list function enabled-p help*`]`
- `[:style` *2d-or-3d*`]`
- `[:style` *2d-or-3d* `:size` *width-or-height*`]`
- `[:size` *width-or-height* `:style` *2d-or-3d*`]`

Optionally, one of the toolbar button descriptors may be `nil` instead of a vector; this signifies the division between the toolbar buttons that are to be displayed flush-left, and the buttons to be displayed flush-right.

The first vector format above specifies a normal toolbar button; the others specify blank areas in the toolbar.

For the first vector format:

- *glyph-list* should be a list of one to six glyphs (as created by `make-glyph`) or a symbol whose value is such a list. The first glyph, which must be provided, is the glyph used to display the toolbar button when it is in the "up" (not pressed) state. The optional second glyph is for displaying the button when it is in the "down" (pressed) state. The optional third glyph is for when the button is disabled. The last three glyphs are for displaying the button in the "up", "down", and "disabled" states, respectively, but are used when the user has called for captioned toolbar buttons (using `toolbar-buttons-captioned-p`). The function `toolbar-make-button-list` is useful in creating these glyph lists.
- Even if you do not provide separate down-state and disabled-state glyphs, the user will still get visual feedback to indicate which state the button is in. Buttons in the up-state are displayed with a shadowed border that gives a raised appearance to the button. Buttons in the down-state are displayed with shadows that give a recessed appearance. Buttons in the disabled state are displayed with no shadows, giving a 2-d effect.
- If some of the toolbar glyphs are not provided, they inherit as follows:

```
UP:             up
DOWN:           down -> up
DISABLED:       disabled -> up
CAP-UP:         cap-up -> up
CAP-DOWN:       cap-down -> cap-up -> down -> up
CAP-DISABLED:   cap-disabled -> cap-up -> disabled -> up
```

- The second element *function* is a function to be called when the toolbar button is activated (i.e. when the mouse is released over the toolbar button, if the press occurred in the toolbar). It can be any form accepted by `call-interactively`, since this is how it is invoked.
- The third element *enabled-p* specifies whether the toolbar button is enabled (disabled buttons do nothing when they are activated, and are displayed differently; see above). It should be either a boolean or a form that evaluates to a boolean.
- The fourth element *help*, if non-`nil`, should be a string. This string is displayed in the echo area when the mouse passes over the toolbar button.

For the other vector formats (specifying blank areas of the toolbar):

- *2d-or-3d* should be one of the symbols `2d` or `3d`, indicating whether the area is displayed with shadows (giving it a raised, 3-d appearance) or without shadows (giving it a flat appearance).
- *width-or-height* specifies the length, in pixels, of the blank area. If omitted, it defaults to a device-specific value (8 pixels for X devices).

toolbar-make-button-list *up* &optional *down disabled cap-up cap-down cap-disabled* — Function

This function calls `make-glyph` on each arg and returns a list of the results. This is useful for setting the first argument of a toolbar button descriptor (typically, the result of this function is assigned to a symbol, which is specified as the first argument of the toolbar button descriptor).

check-toolbar-button-syntax *button* &optional *noerror* Function

Verify the syntax of entry *button* in a toolbar description list. If you want to verify the syntax of a toolbar description list as a whole, use `check-valid-instantiator` with a specifier type of `toolbar`.

23.3 Specifying the Toolbar

In order to specify the contents of a toolbar, set one of the specifier variables `default-toolbar`, `top-toolbar`, `bottom-toolbar`, `left-toolbar`, or `right-toolbar`. These are specifiers, which means you set them with `set-specifier` and query them with `specifier-specs` or `specifier-instance`. You will get an error if you try to set them using `setq`. The valid instantiators for these specifiers are toolbar descriptors, as described above. See Chapter 41 [Specifiers], page 599, for more information.

Most of the time, you will set `default-toolbar`, which allows the user to choose where the toolbar should go.

default-toolbar Specifier

The position of this toolbar is specified in the function `default-toolbar-position`. If the corresponding position-specific toolbar (e.g. `top-toolbar` if `default-toolbar-position` is `top`) does not specify a toolbar in a particular domain, then the value of `default-toolbar` in that domain, of any, will be used instead.

Note that the toolbar at any particular position will not be displayed unless its thickness (width or height, depending on orientation) is non-zero and its visibility status is true. The thickness is controlled by the specifiers `top-toolbar-height`, `bottom-toolbar-height`, `left-toolbar-width`, and `right-toolbar-width`, and the visibility status is controlled by the specifiers `top-toolbar-visible-p`, `bottom-toolbar-visible-p`, `left-toolbar-visible-p`, and `right-toolbar-visible-p` (see Section 23.4 [Other Toolbar Variables], page 354).

set-default-toolbar-position *position* Function

This function sets the position that the `default-toolbar` will be displayed at. Valid positions are the symbols `top`, `bottom`, `left` and `right`. What this actually does is set the fallback specifier for the position-specific specifier corresponding to the given position to `default-toolbar`, and set the fallbacks for the other position-specific specifiers to `nil`. It also does the same thing for the position-specific thickness and visibility specifiers, which inherit from one of `default-toolbar-height` or `default-toolbar-width`, and from `default-toolbar-visible-p`, respectively (see Section 23.4 [Other Toolbar Variables], page 354).

default-toolbar-position Function

This function returns the position that the `default-toolbar` will be displayed at.

You can also explicitly set a toolbar at a particular position. When redisplay determines what to display at a particular position in a particular domain (i.e. window), it first consults the position-specific toolbar. If that does not yield a toolbar descriptor, the `default-toolbar` is consulted if `default-toolbar-position` indicates this position.

top-toolbar Specifier

Specifier for the toolbar at the top of the frame.

bottom-toolbar Specifier

Specifier for the toolbar at the bottom of the frame.

left-toolbar Specifier

Specifier for the toolbar at the left edge of the frame.

right-toolbar Specifier

Specifier for the toolbar at the right edge of the frame.

toolbar-specifier-p *object* Function

This function returns non-nil if *object* is a toolbar specifier. Toolbar specifiers are the actual objects contained in the toolbar variables described above, and their valid instantiators are toolbar descriptors (see Section 23.2 [Toolbar Descriptor Format], page 351).

23.4 Other Toolbar Variables

The variables to control the toolbar thickness, visibility status, and captioned status are all specifiers. See Chapter 41 [Specifiers], page 599.

default-toolbar-height Specifier

This specifies the height of the default toolbar, if it's oriented horizontally. The position of the default toolbar is specified by the function `set-default-toolbar-position`. If the corresponding position-specific toolbar thickness specifier (e.g. `top-toolbar-height` if `default-toolbar-position` is `top`) does not specify a thickness in a particular domain (a window or a frame), then the value of `default-toolbar-height` or `default-toolbar-width` (depending on the toolbar orientation) in that domain, if any, will be used instead.

default-toolbar-width Specifier

This specifies the width of the default toolbar, if it's oriented vertically. This behaves like `default-toolbar-height`.

Note that `default-toolbar-height` is only used when `default-toolbar-position` is `top` or `bottom`, and `default-toolbar-width` is only used when `default-toolbar-position` is `left` or `right`.

top-toolbar-height Specifier

This specifies the height of the top toolbar.

bottom-toolbar-height Specifier

This specifies the height of the bottom toolbar.

left-toolbar-width Specifier

This specifies the width of the left toolbar.

right-toolbar-width Specifier

This specifies the width of the right toolbar.

Note that all of the position-specific toolbar thickness specifiers have a fallback value of zero when they do not correspond to the default toolbar. Therefore, you will have to set a non-zero thickness value if you want a position-specific toolbar to be displayed.

default-toolbar-visible-p Specifier

This specifies whether the default toolbar is visible. The position of the default toolbar is specified by the function `set-default-toolbar-position`. If the corresponding position-specific toolbar visibility specifier (e.g. `top-toolbar-visible-p` if `default-toolbar-position` is `top`) does not specify a visible-p value in a particular domain (a window or a frame), then the value of `default-toolbar-visible-p` in that domain, if any, will be used instead.

top-toolbar-visible-p Specifier

This specifies whether the top toolbar is visible.

bottom-toolbar-visible-p Specifier

This specifies whether the bottom toolbar is visible.

left-toolbar-visible-p Specifier

This specifies whether the left toolbar is visible.

right-toolbar-visible-p Specifier

This specifies whether the right toolbar is visible.

`default-toolbar-visible-p` and all of the position-specific toolbar visibility specifiers have a fallback value of true.

Internally, toolbar thickness and visibility specifiers are instantiated in both window and frame domains, for different purposes. The value in the domain of a frame's selected window specifies the actual toolbar thickness or visibility that you will see in that frame. The value in the domain of a frame itself specifies the toolbar thickness or visibility that is used in frame geometry calculations.

Thus, for example, if you set the frame width to 80 characters and the left toolbar width for that frame to 68 pixels, then the frame will be sized to fit 80 characters plus a 68-pixel left toolbar. If you then set the left toolbar width to 0 for a particular buffer (or if that buffer does not specify a left toolbar or has a nil value specified for `left-toolbar-visible-p`), you will find that, when that buffer is displayed in the selected window, the window will have a width of 86 or 87 characters – the frame is sized for a 68-pixel left toolbar but the selected window specifies that the left toolbar is not visible, so it is expanded to take up the slack.

toolbar-buttons-captioned-p Specifier

Whether toolbar buttons are captioned. This affects which glyphs from a toolbar button descriptor are chosen. See Section 23.2 [Toolbar Descriptor Format], page 351.

You can also reset the toolbar to what it was when XEmacs started up.

initial-toolbar-spec Constant

The toolbar descriptor used to initialize `default-toolbar` at startup.

24 scrollbars

Not yet documented.

25 Drag and Drop

WARNING: the Drag'n'Drop API is still under development and the interface may change! The current implementation is considered experimental.

Drag'n'drop is a way to transfer information between multiple applications. To do this several GUIs define their own protocols. Examples are OffiX, CDE, Motif, KDE, MSWindows, GNOME, and many more. To catch all these protocols, XEmacs provides a generic API.

One prime idea behind the API is to use a data interface that is transparent for all systems. The author thinks that this is best archived by using URL and MIME data, cause any internet enabled system must support these for email already. XEmacs also already provides powerful interfaces to support these types of data (tm and w3).

25.1 Supported Protocols

The current release of XEmacs only support a small set of Drag'n'drop protocols. Some of these only support limited options available in the API.

25.1.1 OffiX DND

WARNING: If you compile in OffiX, you may not be able to use multiple X displays successfully. If the two servers are from different vendors, the results may be unpredictable.

The OffiX Drag'n'Drop protocol is part of a X API/Widget library created by Cesar Crusius. It is based on X-Atoms and ClientMessage events, and works with any X platform supporting them.

OffiX is supported if 'offix is member of the variable dragdrop-protocols, or the feature 'offix is defined.

Unfortunately it uses it's own data types. Examples are: File, Files, Exe, Link, URL, MIME. The API tries to choose the right type for the data that is dragged from XEmacs (well, not yet...).

XEmacs supports both MIME and URL drags and drops using this API. No application interaction is possible while dragging is in progress.

For information about the OffiX project have a look at http://leb.net/~offix/

25.1.2 CDE dt

CDE stands for Common Desktop Environment. It is based on the Motif widget library. It's drag'n'drop protocol is also an abstraction of the Motif protocol (so it might be possible, that XEmacs will also support the Motif protocol soon).

CDE has three different types: file, buffer, and text. XEmacs only uses file and buffer drags. The API will disallow full URL drags, only file method URLs are passed through.

Buffer drags are always converted to plain text.

25.1.3 MSWindows OLE

Only allows file drags and drops.

25.1.4 Loose ends

The following protocols will be supported soon: Xdnd, Motif, Xde (if I get some specs), KDE OffiX (if KDE can find XEmacs windows).

In particular Xdnd will be one of the protocols that can benefit from the XEmacs API, cause it also uses MIME types to encode dragged data.

25.2 Drop Interface

For each activated low-level protocol, a internal routine will catch incoming drops and convert them to a dragdrop-drop type misc-user-event.

This misc-user-event has its function argument set to `dragdrop-drop-dispatch` and the object contains the data of the drop (converted to URL/MIME specific data). This function will search the variable `experimental-dragdrop-drop-functions` for a function that can handle the dropped data.

To modify the drop behavior, the user can modify the variable `experimental-dragdrop-drop-functions`. Each element of this list specifies a possible handler for dropped data. The first one that can handle the data will return `t` and exit. Another possibility is to set a extent-property with the same name. Extents are checked prior to the variable.

The customization group `drag-n-drop` shows all variables of user interest.

25.3 Drag Interface

This describes the drag API (not implemented yet).

26 Major and Minor Modes

A *mode* is a set of definitions that customize XEmacs and can be turned on and off while you edit. There are two varieties of modes: *major modes*, which are mutually exclusive and used for editing particular kinds of text, and *minor modes*, which provide features that users can enable individually.

This chapter describes how to write both major and minor modes, how to indicate them in the modeline, and how they run hooks supplied by the user. For related topics such as keymaps and syntax tables, see Chapter 20 [Keymaps], page 315, and Chapter 38 [Syntax Tables], page 569.

26.1 Major Modes

Major modes specialize XEmacs 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 XEmacs 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 the copy—or define a *derived mode* (see Section 26.1.5 [Derived Modes], page 370). 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 XEmacs 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 19.10 [Recursive Editing], page 310.

The standard XEmacs 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 complicated and specialized mode.

26.1.1 Major Mode Conventions

The code for existing major modes follows various coding conventions, including conventions for local keymap and syntax table initialization, global names, and hooks. Please follow these conventions when you define a new major mode:

- Define a command whose name ends in '`-mode`', with no arguments, that switches to the new mode in the current buffer. This command should set up the keymap, syntax table, and local variables in an existing buffer without changing the buffer's text.
- Write a documentation string for this command that describes the special commands available in this mode. *C-h m* (`describe-mode`) in your mode will display this string.

  The documentation string may include the special documentation substrings, '`\[`*command*`]`', '`\{`*keymap*`}`', and '`\<`*keymap*`>`', that enable the documentation to adapt automatically to the user's own key bindings. See Section 27.3 [Keys in Documentation], page 384.
- The major mode command should start by calling `kill-all-local-variables`. This is what gets rid of the local variables of the major mode previously in effect.
- The major mode command should set the variable `major-mode` to the major mode command symbol. This is how `describe-mode` discovers which documentation to print.
- The major mode command should set the variable `mode-name` to the "pretty" name of the mode, as a string. This 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 757.
- 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 20.7 [Active Keymaps], page 320, 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.
- The mode may have its own syntax table or may share one with other related modes. If it has its own syntax table, it should store this in a variable named *modename*`-mode-syntax-table`. See Chapter 38 [Syntax Tables], page 569.
- 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 39.2 [Abbrev Tables], page 579.
- Use `defvar` to set mode-related variables, so that they are not reinitialized if they already have a value. (Such reinitialization could discard customizations made by the user.)
- To make a buffer-local binding for an Emacs customization variable, use `make-local-variable` in the major mode command, not `make-variable-buffer-local`. The latter function would make the variable local to every buffer in which it is subsequently set, which would affect buffers that do not use this mode. It is undesirable for a mode to have such global effects. See Section 10.9 [Buffer-Local Variables], page 157.

It's ok to use `make-variable-buffer-local`, if you wish, for a variable used only within a single Lisp package.

- Each major mode should have a *mode hook* named *modename*`-mode-hook`. The major mode command should run that hook, with `run-hooks`, as the very last thing it does. See Section 26.4 [Hooks], page 378.
- The major mode command may also run the hooks of some more basic modes. For example, `indented-text-mode` runs `text-mode-hook` as well as `indented-text-mode-hook`. It may run these other hooks immediately before the mode's own hook (that is, after everything else), or it may run them earlier.
- If something special should be done if the user switches a buffer from this mode to any other major mode, the mode can set a local value for `change-major-mode-hook`.
- 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 XEmacs that new buffers created while the current buffer has Funny mode should not inherit Funny mode. Modes such as Dired, Rmail, and Buffer List use this feature.
- If you want to make the new mode the default for files with certain recognizable names, add an element to `auto-mode-alist` to select the mode for those file names. If you define the mode command to autoload, you should add this element in the same file that calls `autoload`. Otherwise, it is sufficient to add the element in the file that contains the mode definition. See Section 26.1.3 [Auto Major Mode], page 367.
- 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.

change-major-mode-hook Variable

This normal hook is run by `kill-all-local-variables` before it does anything else. This gives major modes a way to arrange for something special to be done if the user switches to a different major mode. For best results, make this variable buffer-local, so that it will disappear after doing its job and will not interfere with the subsequent major mode. See Section 26.4 [Hooks], page 378.

26.1.2 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 modeline.
  (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)
  ;; Having '^' is not clean, but page-delimiter
  ;; has them too, and removing those is a pain.
  (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.

```
(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 XEmacs.
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 modeline.
  (lisp-mode-variables nil)              ; This defines various variables.
  (run-hooks 'emacs-lisp-mode-hook))     ; This permits the user to use a
                                         ;   hook to customize the mode.
```

26.1.3 How XEmacs Chooses a Major Mode

Based on information in the file name or in the file itself, XEmacs 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; you're not supposed to customize it. (If you want Emacs to behave differently in Fundamental mode, change the *global* state of Emacs.)

normal-mode &optional *find-file* Command

This function establishes the proper major mode and 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 and in the '`-*-`' line. The variable `enable-local-variables` controls whether to do so.

If you run `normal-mode` interactively, 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 XEmacs Reference 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`.

ignored-local-variables Variable

This variable holds a list of variables that should not be set by a local variables list. Any value specified for one of these variables is ignored.

In addition to this list, any variable whose name has a non-`nil` `risky-local-variable` property is also ignored.

enable-local-eval User Option

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

set-auto-mode Function

This function selects the major mode that is appropriate for the current buffer. It may base its decision on the value of the '-*-' line, on the visited file name (using `auto-mode-alist`), 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 XEmacs Reference 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`, XEmacs 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.

set-buffer-major-mode *buffer* Function

This function sets the major mode of *buffer* to the value of `default-major-mode`. If that variable is `nil`, it uses the current buffer's major mode (if that is suitable).

The low-level primitives for creating buffers do not use this function, but medium-level commands such as `switch-to-buffer` and `find-file-noselect` use it whenever they create buffers.

initial-major-mode Variable

The value of this variable determines the major mode of the initial '`*scratch*`' buffer. The value should be a symbol that is a major mode command 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 37.2 [Regular Expressions], page 550) 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. An ordinary element of the alist looks like `(`*regexp* `.` *mode-function*`)`.

For example,

```
(("^/tmp/fol/" . text-mode)
 ("\\.texinfo\\'" . texinfo-mode)
 ("\\.texi\\'" . texinfo-mode)
 ("\\.el\\'" . emacs-lisp-mode)
 ("\\.c\\'" . c-mode)
 ("\\.h\\'" . c-mode)
 ...)
```

When you visit a file whose expanded file name (see Section 28.8.4 [File Name Expansion], page 409) matches a *regexp*, `set-auto-mode` calls the corresponding *mode-function*. This feature enables XEmacs to select the proper major mode for most files.

If an element of `auto-mode-alist` has the form `(`*regexp function* `t)`, then after calling *function*, XEmacs searches `auto-mode-alist` again for a match against the portion of the file name that did not match before.

This match-again feature is useful for uncompression packages: an entry of the form `("\\.gz\\'" .` *function*`)` can uncompress the file and then put the uncompressed file in the proper mode according to the name sans '`.gz`'.

Here is an example of how to prepend several pattern pairs to `auto-mode-alist`. (You might use this sort of expression in your '`.emacs`' file.)

```
(setq auto-mode-alist
  (append
   ;; File name starts with a dot.
   '(("/\\.[^/]*\\'" . fundamental-mode)
     ;; File name has no dot.
     ("[^\\./]*\\'" . fundamental-mode)
     ;; File name ends in '.C'.
     ("\\.C\\'" . c++-mode))
   auto-mode-alist))
```

interpreter-mode-alist Variable

This variable specifies major modes to use for scripts that specify a command interpreter in an '`#!`' line. Its value is a list of elements of the form `(`*interpreter* `.` *mode*`)`; for example, `("perl" . perl-mode)` is one element present by default. The element says to use mode *mode* if the file specifies *interpreter*.

This variable is applicable only when the `auto-mode-alist` does not indicate which major mode to use.

hack-local-variables &optional *force* Function

This function parses, and binds or evaluates as appropriate, any local variables for the current buffer.

The handling of `enable-local-variables` documented for `normal-mode` actually takes place here. The argument *force* usually comes from the argument *find-file* given to `normal-mode`.

26.1.4 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 27.2 [Accessing Documentation], page 382.)

major-mode Variable

This variable holds the symbol for the current buffer's major mode. This symbol should have a function definition that is the command to switch to that major mode. The `describe-mode` function uses the documentation string of the function as the documentation of the major mode.

26.1.5 Defining Derived Modes

It's often useful to define a new major mode in terms of an existing one. An easy way to do this is to use `define-derived-mode`.

define-derived-mode *variant parent name docstring body...* Macro

This construct defines *variant* as a major mode command, using *name* as the string form of the mode name.

The new command *variant* is defined to call the function *parent*, then override certain aspects of that parent mode:

- The new mode has its own keymap, named *variant*`-map`. `define-derived-mode` initializes this map to inherit from *parent*`-map`, if it is not already set.
- The new mode has its own syntax table, kept in the variable *variant*`-syntax-table`. `define-derived-mode` initializes this variable by copying *parent*`-syntax-table`, if it is not already set.
- The new mode has its own abbrev table, kept in the variable *variant*`-abbrev-table`. `define-derived-mode` initializes this variable by copying *parent*`-abbrev-table`, if it is not already set.
- The new mode has its own mode hook, *variant*`-hook`, which it runs in standard fashion as the very last thing that it does. (The new mode also runs the mode hook of *parent* as part of calling *parent*.)

In addition, you can specify how to override other aspects of *parent* with *body*. The command *variant* evaluates the forms in *body* after setting up all its usual overrides, just before running *variant*`-hook`.

The argument *docstring* specifies the documentation string for the new mode. If you omit *docstring*, `define-derived-mode` generates a documentation string.

Here is a hypothetical example:

```
(define-derived-mode hypertext-mode
  text-mode "Hypertext"
  "Major mode for hypertext.
\\{hypertext-mode-map}"
  (setq case-fold-search nil))

(define-key hypertext-mode-map
  [down-mouse-3] 'do-hyper-link)
```

26.2 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 activate and deactivate minor modes in any order. A minor mode should be able to have its desired effect regardless of the major mode and regardless of the other minor modes in effect.

Often the biggest problem in implementing a minor mode is finding a way to insert the necessary hook into the rest of XEmacs. Minor mode keymaps make this easier than it used to be.

26.2.1 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 modeline. It can also enable or disable a minor mode keymap. Individual commands or hooks can also check the variable's value.

  If you want the minor mode to be enabled separately in each buffer, make the variable buffer-local.

- Define a command whose name is the same as the mode variable. Its job is to enable and disable the mode by setting the variable.

  The command should accept one optional argument. If the argument is `nil`, it should toggle the mode (turn it on if it is off, and off if it is on). Otherwise, it should turn the mode on if the argument is a positive integer, a symbol other than `nil` or `-`, or a list whose CAR is such an integer or symbol; it should turn the mode off otherwise.

  Here is an example taken from the definition of `transient-mark-mode`. It shows the use of `transient-mark-mode` as a variable that enables or disables the mode's behavior, and also shows the proper way to toggle, enable or disable the minor mode based on the raw prefix argument value.

  ```
  (setq transient-mark-mode
        (if (null arg) (not transient-mark-mode)
          (> (prefix-numeric-value arg) 0)))
  ```

- Add an element to `minor-mode-alist` for each minor mode (see Section 26.3.2 [Modeline Variables], page 374). This element should be a list of the following form:

  (*mode-variable string*)

  Here *mode-variable* is the variable that controls enabling of the minor mode, and *string* is a short string, starting with a space, to represent the mode in the modeline. 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)))
  ```

26.2.2 Keymaps and Minor Modes

Each minor mode can have its own keymap, which is active when the mode is enabled. To set up a keymap for a minor mode, add an element to the alist `minor-mode-map-alist`. See Section 20.7 [Active Keymaps], page 320.

One use of minor mode keymaps is to modify the behavior of certain self-inserting characters so that they do something else as well as self-insert. In general, this is the only way to do that, since the facilities for customizing `self-insert-command` are limited to special cases (designed for abbrevs and Auto Fill mode). (Do not try substituting your own definition of `self-insert-command` for the standard one. The editor command loop handles this function specially.)

26.3 Modeline Format

Each Emacs window (aside from minibuffer windows) includes a modeline, which displays status information about the buffer displayed in the window. The modeline contains information about the buffer, such as its name, associated file, depth of recursive editing, and the major and minor modes.

This section describes how the contents of the modeline are controlled. It is in the chapter on modes because much of the information displayed in the modeline relates to the enabled major and minor modes.

`modeline-format` is a buffer-local variable that holds a template used to display the modeline of the current buffer. All windows for the same buffer use the same `modeline-format` and their modelines appear the same (except for scrolling percentages and line numbers).

The modeline 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 `modeline-format` (see Section 26.3.2 [Modeline Variables], page 374), you may want to force an update of the modeline so as to display the new information.

redraw-modeline &optional *all* Function

Force redisplay of the current buffer's modeline. If *all* is non-`nil`, then force redisplay of all modelines.

The modeline is usually displayed in inverse video. This is controlled using the `modeline` face. See Section 42.1 [Faces], page 615.

26.3.1 The Data Structure of the Modeline

The modeline 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 *modeline construct*, and it is built in recursive fashion out of simpler modeline constructs. The same data structure is used for constructing frame titles (see Section 32.3 [Frame Titles], page 476).

modeline-format Variable

> The value of this variable is a modeline construct with overall responsibility for the modeline format. The value of this variable controls which other variables are used to form the modeline text, and where they appear.

A modeline 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 modeline constructs as their values.

The default value of `modeline-format` incorporates the values of variables such as `mode-name` and `minor-mode-alist`. Because of this, very few modes need to alter `modeline-format`. For most purposes, it is sufficient to alter the variables referenced by `modeline-format`.

A modeline construct may be a list, a symbol, or a string. If the value is a list, each element may be a list, a symbol, or a string.

string A string as a modeline 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 26.3.3 [%-Constructs], page 376.

symbol A symbol as a modeline construct stands for its value. The value of *symbol* is used as a modeline construct, in place of *symbol*. However, the symbols `t` and `nil` are ignored; so is any symbol whose value is void.

There is one exception: if the value of *symbol* is a string, it is displayed verbatim: the `%`-constructs are not recognized.

`(`*string rest*`...)` or `(`*list rest*`...)`
A list whose first element is a string or list means to process all the elements recursively and concatenate the results. This is the most common form of mode line construct.

`(`*symbol then else*`)`
A list whose first element is a symbol is a conditional. Its meaning depends on the value of *symbol*. If the value is non-`nil`, the second element, *then*, is processed recursively as a modeline element. But if the value of *symbol* is `nil`, the third element, *else*, is processed recursively. You may omit *else*; then the mode line element displays nothing if the value of *symbol* is `nil`.

(*width rest*...)
: A list whose first element is an integer specifies truncation or padding of the results of *rest*. The remaining elements *rest* are processed recursively as modeline 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 `modeline-format` itself, the new value should use the same variables that appear in the default value (see Section 26.3.2 [Modeline Variables], page 374), rather than duplicating their contents or displaying the information in another fashion. This way, customizations made by the user or by Lisp programs (such as `display-time` and major modes) via changes to those variables remain effective.

Here is an example of a `modeline-format` that might be useful for `shell-mode`, since it contains the hostname and default directory.

```
(setq modeline-format
  (list ""
   'modeline-modified
   "%b--"
   (getenv "HOST")        ; One element is not constant.
   ":"
   'default-directory
   "   "
   'global-mode-string
   "   %[("
   'mode-name
   'modeline-process
   'minor-mode-alist
   "%n"
   ")%]----"
   '(line-number-mode "L%l--")
   '(-3 . "%p")
   "-%-"))
```

26.3.2 Variables Used in the Modeline

This section describes variables incorporated by the standard value of `modeline-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 modeline if `modeline-format` were changed to use them.

modeline-modified Variable

This variable holds the value of the modeline construct that displays whether the current buffer is modified.

The default value of `modeline-modified` is `("--%1*%1+-")`. This means that the modeline displays '`--**-`' if the buffer is modified, '`-----`' if the buffer is not modified, '--%%-' if the buffer is read only, and '--%*--' if the buffer is read only and modified.

Changing this variable does not force an update of the modeline.

modeline-buffer-identification Variable

This variable identifies the buffer being displayed in the window. Its default value is `("%F: %17b")`, which means that it usually displays '`Emacs:`' followed by seventeen characters of the buffer name. (In a terminal frame, it displays the frame name instead of '`Emacs`'; this has the effect of showing the frame number.) You may want to change this in modes such as Rmail that do not behave like a "normal" XEmacs.

global-mode-string Variable

This variable holds a modeline spec that appears in the mode line by default, just after the buffer name. The command `display-time` sets `global-mode-string` to refer to the variable `display-time-string`, which holds a string containing the time and load information.

The '`%M`' construct substitutes the value of `global-mode-string`, but this is obsolete, since the variable is included directly in the modeline.

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 modeline.

minor-mode-alist Variable

This variable holds an association list whose elements specify how the modeline should indicate that a minor mode is active. Each element of the `minor-mode-alist` should be a two-element list:

(*minor-mode-variable modeline-string*)

More generally, *modeline-string* can be any mode line spec. It appears in the mode line when the value of *minor-mode-variable* is non-`nil`, and not otherwise. These strings should begin with spaces so that they don't run together. Conventionally, the *minor-mode-variable* for a specific mode is set to a non-`nil` value when that minor mode is activated.

The default value of `minor-mode-alist` is:

```
minor-mode-alist
⇒ ((vc-mode vc-mode)
    (abbrev-mode " Abbrev")
    (overwrite-mode overwrite-mode)
    (auto-fill-function " Fill")
    (defining-kbd-macro " Def")
    (isearch-mode isearch-mode))
```

`minor-mode-alist` 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.

modeline-process Variable

This buffer-local variable contains the modeline 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-modeline-format Variable

This variable holds the default `modeline-format` for buffers that do not override it. This is the same as `(default-value 'modeline-format)`.

The default value of `default-modeline-format` is:

```
(""
 modeline-modified
 modeline-buffer-identification
 "   "
 global-mode-string
 "   %[("
 mode-name
 modeline-process
 minor-mode-alist
 "%n"
 ")%]----"
 (line-number-mode "L%l--")
 (-3 . "%p")
 "-%-")
```

vc-mode Variable

The variable `vc-mode`, local in each buffer, records whether the buffer's visited file is maintained with version control, and, if so, which kind. Its value is `nil` for no version control, or a string that appears in the mode line.

26.3.3 %-Constructs in the ModeLine

The following table lists the recognized `%`-constructs and what they mean. In any construct except '%%', you can add a decimal integer after the '%' to specify how many characters to display.

`%b` The current buffer name, obtained with the `buffer-name` function. See Section 30.3 [Buffer Names], page 433.

`%f` The visited file name, obtained with the `buffer-file-name` function. See Section 30.4 [Buffer File Name], page 434.

`%F` The name of the selected frame.

`%c` The current column number of point.

`%l` The current line number of point.

%* '%' if the buffer is read only (see buffer-read-only);
'*' if the buffer is modified (see buffer-modified-p);
'-' otherwise. See Section 30.5 [Buffer Modification], page 436.

%+ '*' if the buffer is modified (see buffer-modified-p);
'%' if the buffer is read only (see buffer-read-only);
'-' otherwise. This differs from '%*' only for a modified read-only buffer. See Section 30.5 [Buffer Modification], page 436.

%& '*' if the buffer is modified, and '-' otherwise.

%s The status of the subprocess belonging to the current buffer, obtained with process-status. See Section 49.6 [Process Information], page 679.

%l the current line number.

%S the name of the selected frame; this is only meaningful under the X Window System. See Section 32.2.5 [Frame Name], page 475.

%t Whether the visited file is a text file or a binary file. (This is a meaningful distinction only on certain operating systems.)

%p The percentage of the buffer text above the **top** of window, or 'Top', 'Bottom' or 'All'.

%P The percentage of the buffer text that is above the **bottom** of the window (which includes the text visible in the window, as well as the text above the top), plus 'Top' if the top of the buffer is visible on screen; or 'Bottom' or 'All'.

%n 'Narrow' when narrowing is in effect; nothing otherwise (see narrow-to-region in Section 34.4 [Narrowing], page 496).

%[An indication of the depth of recursive editing levels (not counting minibuffer levels): one '[' for each editing level. See Section 19.10 [Recursive Editing], page 310.

%] 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 modeline.

The following two %-constructs are still supported, but they are obsolete, since you can get the same results with the variables mode-name and global-mode-string.

%m The value of mode-name.

%M The value of global-mode-string. Currently, only display-time modifies the value of global-mode-string.

26.4 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. XEmacs provides hooks for the sake of customization. Most often, hooks are set up in the '.emacs' file, but Lisp programs can set them also. See Appendix F [Standard Hooks], page 785, for a list of standard hook variables.

Most of the hooks in XEmacs 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 usually 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 163). 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 that is a single function. Those whose names end in '-hooks' have a value that is a list of functions. Any hook that 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 that you should look at its documentation string to see how to use it properly.

Major mode functions are supposed to run a hook called the *mode hook* as the last step of initialization. This makes it easy for a user to customize the behavior of the mode, by overriding the local variable assignments already made by the mode. But hooks are used in other contexts too. For example, the hook `suspend-hook` runs just before XEmacs suspends itself (see Section 50.2.2 [Suspending XEmacs], page 696).

Here's an expression that uses a mode hook to turn on Auto Fill mode when in Lisp Interaction mode:

```
(add-hook 'lisp-interaction-mode-hook 'turn-on-auto-fill)
```

The next example shows how to use a hook to customize the way XEmacs formats C code. (People often have strong personal preferences for one format or 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)
```

The final example shows how the appearance of the modeline can be modified for a particular class of buffers only.

```
(add-hook 'text-mode-hook
  (function (lambda ()
              (setq modeline-format
                    '(modeline-modified
                      "Emacs: %14b"
                      "  "
                      default-directory
                      " "
                      global-mode-string
                      "%[("
                      mode-name
                      minor-mode-alist
                      "%n"
                      modeline-process
                      ") %]---"
                      (-3 . "%p")
                      "-%-")))))
```

At the appropriate time, XEmacs 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 variable names as arguments, and runs each hook in turn. Each *hookvar* argument should be a symbol that is a hook variable. These arguments are processed in the order specified.

If a hook variable has a non-`nil` value, that value may be a function or a list of functions. If the value is a function (either a lambda expression or a symbol with a function definition), it is called. If it is a list, the elements are called, in order. The hook functions are called with no arguments.

For example, here's how `emacs-lisp-mode` runs its mode hook:

```
(run-hooks 'emacs-lisp-mode-hook)
```

add-hook *hook function* &optional *append local* Function

This function is the handy way to add function *function* to hook variable *hook*. The argument *function* may be any valid Lisp function with the proper number of arguments. For example,

```
(add-hook 'text-mode-hook 'my-text-hook-function)
```

adds `my-text-hook-function` to the hook called `text-mode-hook`.

You can use `add-hook` for abnormal hooks as well as for normal hooks.

It is best to design your hook functions so that the order in which they are executed does not matter. Any dependence on the order is "asking for trouble." However, the order is predictable: normally, *function* goes at the front of the hook list, so it will be executed first (barring another `add-hook` call).

If the optional argument *append* is non-`nil`, the new hook function goes at the end of the hook list and will be executed last.

If *local* is non-`nil`, that says to make the new hook function local to the current buffer. Before you can do this, you must make the hook itself buffer-local by calling `make-local-hook` (**not** `make-local-variable`). If the hook itself is not buffer-local, then the value of *local* makes no difference—the hook function is always global.

remove-hook *hook function* &optional *local* Function

This function removes *function* from the hook variable *hook*.

If *local* is non-`nil`, that says to remove *function* from the local hook list instead of from the global hook list. If the hook itself is not buffer-local, then the value of *local* makes no difference.

make-local-hook *hook* Function

This function makes the hook variable `hook` local to the current buffer. When a hook variable is local, it can have local and global hook functions, and `run-hooks` runs all of them.

This function works by making `t` an element of the buffer-local value. That serves as a flag to use the hook functions in the default value of the hook variable as well as those in the local value. Since `run-hooks` understands this flag, `make-local-hook` works with all normal hooks. It works for only some non-normal hooks—those whose callers have been updated to understand this meaning of `t`.

Do not use `make-local-variable` directly for hook variables; it is not sufficient.

27 Documentation

XEmacs 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 XEmacs are not the same thing as the XEmacs 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.

27.1 Documentation Basics

A documentation string is written using the Lisp syntax for strings, with double-quote characters surrounding the text of the string. This is because it really is a Lisp string object. The string serves as documentation when it is written in the proper place in the definition of a function or variable. In a function definition, the documentation string follows the argument list. In a variable definition, the documentation string follows the initial value of the variable.

When you write a documentation string, make the first line a complete sentence (or two complete sentences) since some commands, such as `apropos`, show only the first line of a multi-line documentation string. Also, you should not indent the second line of a documentation string, if you have one, because that looks odd when you use *C-h f* (`describe-function`) or *C-h v* (`describe-variable`). See Section A.3 [Documentation Tips], page 760.

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 27.2 [Accessing Documentation], page 382.)

Within the Lisp world, a documentation string is accessible through the function or variable that it describes:

- The documentation for a function is stored in the function definition itself (see Section 11.2 [Lambda Expressions], page 164). The function `documentation` knows how to extract it.
- The documentation for a variable is stored in the variable's property list under the property name `variable-documentation`. The function `documentation-property` knows how to extract it.

To save space, the documentation for preloaded functions and variables (including primitive functions and autoloaded functions) is stored in the *internal doc file* '`DOC`'. The documentation for functions and variables loaded during the XEmacs session from byte-compiled files is stored in those very same byte-compiled files (see Section 15.3 [Docs and Compilation], page 208).

XEmacs does not keep documentation strings in memory unless necessary. Instead, XEmacs maintains, for preloaded symbols, an integer offset into the internal doc file, and for symbols loaded from byte-compiled files, a list containing the filename of the byte-compiled file and an integer offset, in place of the documentation string. The functions `documentation` and `documentation-property` use that information to read the documentation from the appropriate file; this is transparent to the user.

For information on the uses of documentation strings, see section "Help" in *The XEmacs Reference Manual*.

The '`emacs/lib-src`' directory contains two utilities that you can use to print nice-looking hardcopy for the file '`emacs/etc/DOC-`*version*'. These are '`sorted-doc.c`' and '`digest-doc.c`'.

27.2 Access to Documentation Strings

documentation-property *symbol property* &optional *verbatim* Function

This function returns the documentation string that is recorded in *symbol*'s property list under property *property*. It retrieves the text from a file if necessary, and runs `substitute-command-keys` to substitute actual key bindings. (This substitution is not done if *verbatim* is non-`nil`; 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*. It reads the text from a file if necessary. Then (unless *verbatim* is non-`nil`) it calls `substitute-command-keys`, to return a value containing the actual (current) key bindings.

The function `documentation` signals a `void-function` error if *function* has no function definition. However, it is ok if the function definition has no documentation string. In that case, `documentation` returns `nil`.

Here is an example of using the two functions, `documentation` and `documentation-property`, to display the documentation strings for several symbols in a '`*Help*`' buffer.

```
(defun describe-symbols (pattern)
  "Describe the XEmacs 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 XEmacs initialization, just before the runnable XEmacs 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 XEmacs], page 765.

XEmacs finds the file *filename* in the ‘lib-src’ directory. When the dumped XEmacs is later executed, the same file is found in the directory `doc-directory`. The usual value for *filename* is ‘DOC’, but this can be changed by modifying the variable `internal-doc-file-name`.

internal-doc-file-name Variable

This variable holds the name of the file containing documentation strings of built-in symbols, usually ‘DOC’. The full pathname of the internal doc file is ‘(concat doc-directory internal-doc-file-name)’.

doc-directory Variable

This variable holds the name of the directory which contains the *internal doc file* that contains documentation strings for built-in and preloaded functions and variables.

In most cases, this is the same as `exec-directory`. They may be different when you run XEmacs from the directory where you built it, without actually installing it. See `exec-directory` in Section 27.5 [Help Functions], page 387.

In older Emacs versions, `exec-directory` was used for this.

data-directory Variable

This variable holds the name of the directory in which XEmacs finds certain system independent documentation and text files that come with XEmacs. In older Emacs versions, `exec-directory` was used for this.

27.3 Substituting Key Bindings in Documentation

When documentation strings refer to key sequences, they should use the current, actual key bindings. They can do so using certain special text sequences described below. Accessing documentation strings in the usual way substitutes current key binding information for these special sequences. This works by calling `substitute-command-keys`. You can also call that function yourself.

Here is a list of the special sequences and what they mean:

`\[`*command*`]`
stands for a key sequence that will invoke *command*, or '`M-x` *command*' if *command* has no key bindings.

`\{`*mapvar*`}`
stands for a summary of the value of *mapvar*, which should be a keymap. The summary is made by `describe-bindings`.

`\<`*mapvar*`>`
stands for no text itself. It is used for a side effect: it specifies *mapvar* as the keymap for any following '`\[`*command*`]`' sequences in this documentation string.

`\=` quotes the following character and is discarded; this '`\=\=`' puts '`\=`' into the output, and '`\=\[`' puts '`\[`' into the output.

Please note: Each '`\`' must be doubled when written in a string in XEmacs Lisp.

substitute-command-keys *string* Function

This function scans *string* for the above special sequences and replaces them by what they stand for, returning the result as a string. This permits display of documentation that refers accurately to the user's own customized key bindings.

Here are examples of the special sequences:

```
(substitute-command-keys
   "To abort recursive edit, type: \\[abort-recursive-edit]")
⇒ "To abort recursive edit, type: C-]"

(substitute-command-keys
   "The keys that are defined for the minibuffer here are:
  \\{minibuffer-local-must-match-map}")
⇒ "The keys that are defined for the minibuffer here are:

?               minibuffer-completion-help
SPC             minibuffer-complete-word
TAB             minibuffer-complete
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."
```

```
(substitute-command-keys
  "Substrings of the form \\=\\{MAPVAR} are replaced by summaries
\(made by describe-bindings) of the value of MAPVAR, taken as a keymap.
Substrings of the form \\=\\<MAPVAR> specify to use the value of MAPVAR
as the keymap for future \\=\\[COMMAND] substrings.
\\=\\= quotes the following character and is discarded;
thus, \\=\\=\\=\\= puts \\=\\= into the output,
and \\=\\=\\=\\[ puts \\=\\[ into the output.")
⇒ "Substrings of the form \{MAPVAR} are replaced by summaries
(made by describe-bindings) of the value of MAPVAR, taken as a keymap.
Substrings of the form \<MAPVAR> specify to use the value of MAPVAR
as the keymap for future \[COMMAND] substrings.
\= quotes the following character and is discarded;
thus, \=\= puts \= into the output,
and \=\[ puts \[ into the output."
```

27.4 Describing Characters for Help Messages

These functions convert events, key sequences or characters to textual descriptions. These descriptions are useful for including arbitrary text characters or key sequences in messages, because they convert non-printing and whitespace characters to sequences of printing characters. The description of a non-whitespace printing character is the character itself.

key-description *sequence* Function

This function returns a string containing the XEmacs standard notation for the input events in *sequence*. The argument *sequence* may be a string, vector or list. See Section 19.5 [Events], page 290, for more information about valid events. See also the examples for `single-key-description`, below.

single-key-description *key* Function

This function returns a string describing *key* in the standard XEmacs notation for keyboard input. A normal printing character appears as itself, but a control character turns into a string starting with '`C-`', a meta character turns into a string starting with '`M-`', and space, linefeed, etc. appear as '`SPC`', '`LFD`', etc. A symbol appears as the name of the symbol. An event that is a list appears as the name of the symbol in the CAR of the list.

```
(single-key-description ?\C-x)
     ⇒ "C-x"
(key-description "\C-x \M-y \n \t \r \f123")
     ⇒ "C-x SPC M-y SPC LFD SPC TAB SPC RET SPC C-l 1 2 3"
(single-key-description 'kp_next)
     ⇒ "kp_next"
(single-key-description '(shift button1))
     ⇒ "Sh-button1"
```

text-char-description *character* Function

This function returns a string describing *character* in the standard XEmacs 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 XEmacs 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"
```

27.5 Help Functions

XEmacs provides a variety of on-line help functions, all accessible to the user as subcommands of the prefix `C-h`, or on some keyboards, `help`. For more information about them, see section "Help" in *The XEmacs Reference 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 (see Section 37.2 [Regular Expressions], page 550). 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)
```

`apropos` is used by various user-level commands, such as `C-h a` (`hyper-apropos`), a graphical front-end to `apropos`; and `C-h A` (`command-apropos`), which does an apropos over only those functions which are user commands. `command-apropos` calls `apropos`, specifying a *predicate* to restrict the output to symbols that are commands. The call to `apropos` looks like this:

```
(apropos string t 'commandp)
```

help-map — Variable

The value of this variable is a local keymap for characters following the Help key, `C-h`.

help-command Prefix Command

This symbol is not a function; its function definition is actually the keymap known as `help-map`. It is defined in '`help.el`' as follows:

```
(define-key global-map "\C-h" 'help-command)
(fset 'help-command help-map)
```

print-help-return-message &optional *function* Function

This function builds a string that explains how to restore the previous state of the windows after a help command. After building the message, it applies *function* to it if *function* is non-`nil`. Otherwise it calls `message` to display it in the echo area.

This function expects to be called inside a `with-output-to-temp-buffer` special form, and expects `standard-output` to have the value bound by that special form. For an example of its use, see the long example in Section 27.2 [Accessing Documentation], page 382.

help-char Variable

The value of this variable is the help character—the character that XEmacs recognizes as meaning Help. By default, it is the character '`?\^H`' (ASCII 8), which is *`C-h`*. When XEmacs reads this character, if `help-form` is non-`nil` Lisp expression, it evaluates that expression, and displays the result in a window if it is a string.

`help-char` can be a character or a key description such as `help` or `(meta h)`.

Usually the value of `help-form`'s value is `nil`. Then the help character has no special meaning at the level of command input, and it becomes part of a key sequence in the normal way. The standard key binding of *`C-h`* is a prefix key for several general-purpose help features.

The help character is special after prefix keys, too. If it has no binding as a subcommand of the prefix key, it runs `describe-prefix-bindings`, which displays a list of all the subcommands of the prefix key.

help-form Variable

If this variable is non-`nil`, its value is a form to evaluate whenever the character `help-char` is read. If evaluating the form produces a string, that string is displayed.

A command that calls `next-command-event` or `next-event` probably should bind `help-form` to a non-`nil` expression while it does input. (The exception is when *`C-h`* is meaningful input.) Evaluating this expression should result in a string that explains what the input is for and how to enter it properly.

Entry to the minibuffer binds this variable to the value of `minibuffer-help-form` (see Section 18.8 [Minibuffer Misc], page 278).

prefix-help-command Variable

This variable holds a function to print help for a prefix character. The function is called when the user types a prefix key followed by the help character, and the help character has no binding after that prefix. The variable's default value is `describe-prefix-bindings`.

describe-prefix-bindings Function

This function calls `describe-bindings` to display a list of all the subcommands of the prefix key of the most recent key sequence. The prefix described consists of all but the last event of that key sequence. (The last event is, presumably, the help character.)

The following two functions are 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`.

27.6 Obsoleteness

As you add functionality to a package, you may at times want to replace an older function with a new one. To preserve compatibility with existing code, the older function needs to still exist; but users of that function should be told to use the newer one instead. XEmacs Lisp lets you mark a function or variable as *obsolete*, and indicate what should be used instead.

make-obsolete *function new* Function

This function indicates that *function* is an obsolete function, and the function *new* should be used instead. The byte compiler will issue a warning to this effect when it encounters a usage of the older function, and the help system will also note this in the function's documentation. *new* can also be a string (if there is not a single function with the same functionality any more), and should be a descriptive statement, such as "use *foo* or *bar* instead" or "this function is unnecessary".

make-obsolete-variable *variable new* Function

This is like `make-obsolete` but is for variables instead of functions.

define-obsolete-function-alias *oldfun newfun* Function

This function combines `make-obsolete` and `define-function`, declaring *oldfun* to be an obsolete variant of *newfun* and defining *oldfun* as an alias for *newfun*.

define-obsolete-variable-alias *oldvar newvar* Function

This is like `define-obsolete-function-alias` but for variables.

Note that you should not normally put obsoleteness information explicitly in a function or variable's doc string. The obsoleteness information that you specify using the above functions will be displayed whenever the doc string is displayed, and by adding it explicitly the result is redundancy.

Also, if an obsolete function is substantially the same as a newer one but is not actually an alias, you should consider omitting the doc string entirely (use a null string '`""`' as the doc string). That way, the user is told about the obsoleteness and is forced to look at the documentation of the new function, making it more likely that he will use the new function.

function-obsoleteness-doc *function* Function

If *function* is obsolete, this function returns a string describing this. This is the message that is printed out during byte compilation or in the function's documentation. If *function* is not obsolete, `nil` is returned.

variable-obsoleteness-doc *variable* Function

This is like `function-obsoleteness-doc` but for variables.

The obsoleteness information is stored internally by putting a property `byte-obsolete-info` (for functions) or `byte-obsolete-variable` (for variables) on the symbol that specifies the obsolete function or variable. For more information, see the implementation of `make-obsolete` and `make-obsolete-variable` in '`lisp/bytecomp/bytecomp-runtime.el`'.

28 Files

In XEmacs, you can find, create, view, save, and otherwise work with files and file directories. This chapter describes most of the file-related functions of XEmacs Lisp, but a few others are described in Chapter 30 [Buffers], page 431, and those related to backups and auto-saving are described in Chapter 29 [Backups and Auto-Saving], page 421.

Many of the file functions take one or more arguments that are file names. A file name is actually a string. Most of these functions expand file name arguments using `expand-file-name`, so that '`~`' is handled correctly, as are relative file names (including '`../`'). These functions don't recognize environment variable substitutions such as '`$HOME`'. See Section 28.8.4 [File Name Expansion], page 409.

28.1 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 XEmacs that will vanish at the end of the editing session (or when you kill the buffer). Usually, a buffer contains information that you have copied from a file; then we say the buffer is visiting that file. The copy in the buffer is what you modify with editing commands. Such changes to the buffer do not change the file; therefore, to make the changes permanent, you must *save* the buffer, which means copying the altered buffer contents back into the file.

In spite of the distinction between files and buffers, people often refer to a file when they mean a buffer and vice-versa. Indeed, we say, "I am editing a file," rather than, "I am editing a buffer that I will soon save as a file of the same name." Humans do not usually need to make the distinction explicit. When dealing with a computer program, however, it is good to keep the distinction in mind.

28.1.1 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 30.4 [Buffer File Name], page 434, for functions and variables that access the visited file name of a buffer or that find an existing buffer by its visited file name.

In a Lisp program, if you want to look at the contents of a file but not alter it, the fastest way is to use `insert-file-contents` in a temporary buffer. Visiting the file is not necessary and takes longer. See Section 28.3 [Reading from Files], page 396.

find-file *filename* Command

This command selects a buffer visiting the file *filename*, using an existing buffer if there is one, and otherwise creating a new buffer and reading the file into it. It also returns that buffer.

The body of the `find-file` function is very simple and looks like this:

```
(switch-to-buffer (find-file-noselect filename))
```

(See `switch-to-buffer` in Section 31.7 [Displaying Buffers], page 453.)

When `find-file` is called interactively, it prompts for *filename* in the minibuffer.

find-file-noselect *filename* &optional *nowarn* Function

This function is the guts of all the file-visiting functions. It finds or creates a buffer visiting the file *filename*, and returns it. It uses an existing buffer if there is one, and otherwise creates a new buffer and reads the file into it. You may make the buffer current or display it in a window if you wish, but this function does not do so.

When `find-file-noselect` uses an existing buffer, it first verifies that the file has not changed since it was last visited or saved in that buffer. If the file has changed, then this function asks the user whether to reread the changed file. If the user says '`yes`', any changes previously made in the buffer are lost.

If `find-file-noselect` needs to create a buffer, and there is no file named *filename*, it displays the message '`New file`' in the echo area, and leaves the buffer empty.

If *no-warn* is non-`nil`, various warnings that XEmacs normally gives (e.g. if another buffer is already visiting *filename* but *filename* has been removed from disk since that buffer was created) are suppressed.

The `find-file-noselect` function calls `after-find-file` after reading the file (see Section 28.1.2 [Subroutines of Visiting], page 393). That function sets the buffer major mode, parses local variables, warns the user if there exists an auto-save file more recent than the file just visited, and finishes by running the functions in `find-file-hooks`.

The `find-file-noselect` function returns the buffer that is visiting the file *filename*.

```
(find-file-noselect "/etc/fstab")
     ⇒ #<buffer fstab>
```

find-file-other-window *filename* Command

This command selects a buffer visiting the file *filename*, but does so in a window other than the selected window. It may use another existing window or split a window; see Section 31.7 [Displaying Buffers], page 453.

When this command is called interactively, it prompts for *filename*.

find-file-read-only *filename* Command

This command selects a buffer visiting the file *filename*, like `find-file`, but it marks the buffer as read-only. See Section 30.7 [Read Only Buffers], page 438, for related functions and variables.

When this command is called interactively, it prompts for *filename*.

view-file *filename* Command

This command visits *filename* in View mode, and displays it in a recursive edit, 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. Entering View mode runs the normal hook `view-mode-hook`. See Section 26.4 [Hooks], page 378.

When `view-file` is called interactively, it prompts for *filename*.

find-file-hooks Variable

The value of this variable is a list of functions to be called after a file is visited. The file's local-variables specification (if any) will have been processed before the hooks are run. The buffer visiting the file is current when the hook functions are run.

This variable works just like a normal hook, but we think that renaming it would not be advisable.

find-file-not-found-hooks Variable

The value of this variable is a list of functions to be called when `find-file` or `find-file-noselect` is passed a nonexistent file name. `find-file-noselect` calls these functions as soon as it detects a nonexistent file. It calls them in the order of the list, until one of them returns non-`nil`. `buffer-file-name` is already set up.

This is not a normal hook because the values of the functions are used and they may not all be called.

28.1.2 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. It uses *filename* (sans directory) as the name if that name is free; otherwise, it appends a string such as '`<2>`' to get an unused name. See also Section 30.9 [Creating Buffers], page 440.

Please note: `create-file-buffer` does *not* associate the new buffer with a file and does not select the buffer. It also does not use the default major mode.

```
(create-file-buffer "foo")
     ⇒ #<buffer foo>
(create-file-buffer "foo")
     ⇒ #<buffer foo<2>>
(create-file-buffer "foo")
     ⇒ #<buffer foo<3>>
```

This function is used by `find-file-noselect`. It uses `generate-new-buffer` (see Section 30.9 [Creating Buffers], page 440).

after-find-file &optional *error warn noauto* Function

This function sets the buffer major mode, and parses local variables (see Section 26.1.3 [Auto Major Mode], page 367). It is called by `find-file-noselect` and by the default revert function (see Section 29.3 [Reverting], page 428).

If reading the file got an error because the file does not exist, but its directory does exist, the caller should pass a non-`nil` value for *error*. In that case, `after-find-file` issues a warning: '`(New File)`'. For more serious errors, the caller should usually not call `after-find-file`.

If *warn* is non-`nil`, then this function issues a warning if an auto-save file exists and is more recent than the visited file.

If *noauto* is non-`nil`, then this function does not turn on auto-save mode; otherwise, it does.

The last thing `after-find-file` does is call all the functions in `find-file-hooks`.

28.2 Saving Buffers

When you edit a file in XEmacs, you are actually working on a buffer that is visiting that file—that is, the contents of the file are copied into the buffer and the copy is what you edit. Changes to the buffer do not change the file until you *save* the buffer, which means copying the contents of the buffer into the file.

save-buffer &optional *backup-option* Command

This function saves the contents of the current buffer in its visited file if the buffer has been modified since it was last visited or saved. Otherwise it does nothing.

`save-buffer` is responsible for making backup files. Normally, *backup-option* is `nil`, and `save-buffer` makes a backup file only if this is the first save since visiting the file. Other values for *backup-option* request the making of backup files in other circumstances:

- With an argument of 4 or 64, reflecting 1 or 3 *C-u*'s, the `save-buffer` function marks this version of the file to be backed up when the buffer is next saved.
- With an argument of 16 or 64, reflecting 2 or 3 *C-u*'s, the `save-buffer` function unconditionally backs up the previous version of the file before saving it.

save-some-buffers &optional *save-silently-p exiting* Command

This command saves some modified file-visiting buffers. Normally it asks the user about each buffer. But if *save-silently-p* is non-`nil`, it saves all the file-visiting buffers without querying the user.

The optional *exiting* argument, if non-`nil`, requests this function to offer also to save certain other buffers that are not visiting files. These are buffers that have a non-`nil` 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, XEmacs 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. Then it renames the buffer based on *filename*, appending a string like '`<2>`' if necessary to make a unique buffer name. It does most of this work by calling `set-visited-file-name` and `save-buffer`.

write-file-hooks Variable

The value of this variable is a list of functions to be called before writing out a buffer to its visited file. If one of them returns non-`nil`, the file is considered already written and the rest of the functions are not called, nor is the usual code for writing the file executed.

If a function in `write-file-hooks` returns non-`nil`, it is responsible for making a backup file (if that is appropriate). To do so, execute the following code:

```
(or buffer-backed-up (backup-buffer))
```

You might wish to save the file modes value returned by `backup-buffer` and use that to set the mode bits of the file that you write. This is what `save-buffer` normally does.

Even though this is not a normal hook, you can use `add-hook` and `remove-hook` to manipulate the list. See Section 26.4 [Hooks], page 378.

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 for hooks that pertain to the contents of the file, as opposed to hooks that pertain to where the file came from. Such hooks are usually set up by major modes, as buffer-local bindings for this variable. Switching to a new major mode always resets this variable.

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.

As a side effect, backups are necessarily made by copying. See Section 29.1.2 [Rename or Copy], page 422. Yet, at the same time, saving a precious file always breaks all hard links between the file you save and other file names.

Some modes set this variable 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 `save-buffer` silently adds a newline at the end of the file whenever the buffer being saved does not already end in one. If the value of the variable is non-`nil`, but not `t`, then `save-buffer` asks the user whether to add a newline each time the case arises.

If the value of the variable is `nil`, then `save-buffer` doesn't add newlines at all. `nil` is the default value, but a few major modes set it to `t` in particular buffers.

28.3 Reading from Files

You can copy a file from the disk and insert it into a buffer using the `insert-file-contents` function. Don't use the user-level command `insert-file` in a Lisp program, as that sets the mark.

insert-file-contents *filename* &optional *visit beg end replace* Function

This function inserts the contents of file *filename* into the current buffer after point. It returns a list of the absolute file name and the length of the data inserted. An error is signaled if *filename* is not the name of a file that can be read.

The function `insert-file-contents` checks the file contents against the defined file formats, and converts the file contents if appropriate. See Section 28.13 [Format Conversion], page 417. It also calls the functions in the list `after-insert-file-functions`; see Section 36.18.5 [Saving Properties], page 543.

If *visit* is non-`nil`, this function additionally marks the buffer as unmodified and sets up various fields in the buffer so that it is visiting the file *filename*: these include the buffer's visited file name and its last save file modtime. This feature is used by `find-file-noselect` and you probably should not use it yourself.

If *beg* and *end* are non-`nil`, they should be integers specifying the portion of the file to insert. In this case, *visit* must be `nil`. For example,

```
(insert-file-contents filename nil 0 500)
```

inserts the first 500 characters of a file.

If the argument *replace* is non-`nil`, it means to replace the contents of the buffer (actually, just the accessible portion) with the contents of the file. This is better than simply deleting the buffer contents and inserting the whole file, because (1) it preserves some marker positions and (2) it puts less data in the undo list.

If you want to pass a file name to another process so that another program can read the file, use the function `file-local-copy`; see Section 28.11 [Magic File Names], page 414.

28.4 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. If that file exists it is overwritten. 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 delimited by *start* and *end* in the current buffer into the file specified by *filename*.

If *start* is a string, then `write-region` writes or appends that string, rather than text from the buffer.

If *append* is non-`nil`, then the specified text is appended to the existing file contents (if any).

If *visit* is `t`, then XEmacs establishes an association between the buffer and the file: the buffer is then visiting that file. It also sets the last file modification time for the current buffer to *filename*'s modtime, and marks the buffer as not modified. This feature is used by `save-buffer`, but you probably should not use it yourself.

If *visit* is a string, it specifies the file name to visit. This way, you can write the data to one file (*filename*) while recording the buffer as visiting another file (*visit*). The argument *visit* is used in the echo area message and also for file locking; *visit* is stored in `buffer-file-name`. This feature is used to implement `file-precious-flag`; don't use it yourself unless you really know what you're doing.

The function `write-region` converts the data which it writes to the appropriate file formats specified by `buffer-file-format`. See Section 28.13 [Format Conversion], page 417. It also calls the functions in the list `write-region-annotate-functions`; see Section 36.18.5 [Saving Properties], page 543.

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 that the user does not need to know about.

28.5 File Locks

When two users edit the same file at the same time, they are likely to interfere with each other. XEmacs tries to prevent this situation from arising by recording a *file lock* when a file is being modified. XEmacs can then detect the first attempt to modify a buffer visiting a file that is locked by another XEmacs process, 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 XEmacs 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 30.6 [Modification Time], page 437.

file-locked-p &optional *filename* Function

This function returns `nil` if the file *filename* is not locked by this XEmacs process. It returns `t` if it is locked by this XEmacs, 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 named *other-user*. The value it returns determines what happens next:

- A value of `t` says to grab the lock on the file. Then this user may edit the file and *other-user* loses the lock.
- A value of `nil` says to ignore the lock and let this user edit the file anyway.
- This function may instead signal a `file-locked` error, in which case the change that the user was about to make does not take place.

  The error message for this error looks like this:

  ```
  error  File is locked: file other-user
  ```

  where `file` is the name of the file and *other-user* is the name of the user who has locked the file.

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'.

28.6 Information about Files

The functions described in this section all operate on strings that designate file names. All the functions have names that begin with the word 'file'. These functions all return information about actual files or directories, so their arguments must all exist as actual files or directories unless otherwise noted.

28.6.1 Testing Accessibility

These functions test for permission to access a file in specific ways.

file-exists-p *filename* Function

This function returns `t` if a file named *filename* appears to exist. This does not mean you can necessarily read the file, only that you can find out its attributes. (On Unix, this is true if the file exists and you have execute permission on the containing directories, regardless of the protection of the file itself.)

If the file does not exist, or if fascist access control policies prevent you from finding the attributes of the file, this function returns `nil`.

file-readable-p *filename* Function

This function returns `t` if a file named *filename* exists and you can read it. It returns `nil` otherwise.

```
(file-readable-p "files.texi")
     ⇒ t
(file-exists-p "/usr/spool/mqueue")
     ⇒ t
(file-readable-p "/usr/spool/mqueue")
     ⇒ nil
```

file-executable-p *filename* Function

This function returns `t` if a file named *filename* exists and you can execute it. It returns `nil` otherwise. If the file is a directory, execute permission means you can check the existence and attributes of files inside the directory, and open those files if their modes permit.

file-writable-p *filename* Function

This function returns `t` if the file *filename* can be written or created by you, and `nil` otherwise. A file is writable if the file exists and you can write it. It is creatable if it does not exist, but the specified directory does exist and you can write in that directory.

In the third example below, 'foo' is not writable because the parent directory does not exist, even though the user could create such a directory.

```
(file-writable-p "~/foo")
     ⇒ t
```

```
(file-writable-p "/foo")
     ⇒ nil
(file-writable-p "~/no-such-dir/foo")
     ⇒ nil
```

file-accessible-directory-p *dirname* Function

This function returns `t` if you have permission to open existing files in the directory whose name as a file is *dirname*; otherwise (or if there is no such directory), it returns `nil`. The value of *dirname* may be either a directory name or the file name of a 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-ownership-preserved-p *filename* Function

This function returns `t` if deleting the file *filename* and then creating it anew would keep the file's owner unchanged.

file-newer-than-file-p *filename1 filename2* Function

This function returns `t` if the file *filename1* is newer than file *filename2*. If *filename1* does not exist, it returns `nil`. If *filename2* does not exist, it returns `t`.

In the following example, assume that the file '`aug-19`' was written on the 19th, '`aug-20`' was written on the 20th, and the file '`no-file`' doesn't exist at all.

```
(file-newer-than-file-p "aug-19" "aug-20")
     ⇒ nil
(file-newer-than-file-p "aug-20" "aug-19")
     ⇒ t
(file-newer-than-file-p "aug-19" "no-file")
     ⇒ t
(file-newer-than-file-p "no-file" "aug-19")
     ⇒ nil
```

You can use `file-attributes` to get a file's last modification time as a list of two numbers. See Section 28.6.4 [File Attributes], page 401.

28.6.2 Distinguishing Kinds of Files

This section describes how to distinguish various kinds of files, such as directories, symbolic links, and ordinary files.

file-symlink-p *filename* Function

If the file *filename* is a symbolic link, the `file-symlink-p` function returns the file name to which it is linked. This may be the name of a text file, a directory, or even another symbolic link, or it may be a nonexistent file name.

If the file *filename* is not a symbolic link (or there is no such file), `file-symlink-p` returns `nil`.

```
(file-symlink-p "foo")
     ⇒ nil
(file-symlink-p "sym-link")
     ⇒ "foo"
(file-symlink-p "sym-link2")
     ⇒ "sym-link"
(file-symlink-p "/bin")
     ⇒ "/pub/bin"
```

file-directory-p *filename* Function

This function returns `t` if *filename* is the name of an existing directory, `nil` otherwise.

```
(file-directory-p "~rms")
     ⇒ t
(file-directory-p "~rms/lewis/files.texi")
     ⇒ nil
(file-directory-p "~rms/lewis/no-such-file")
     ⇒ nil
(file-directory-p "$HOME")
     ⇒ nil
(file-directory-p
 (substitute-in-file-name "$HOME"))
     ⇒ t
```

file-regular-p *filename* Function

This function returns `t` if the file *filename* exists and is a regular file (not a directory, symbolic link, named pipe, terminal, or other I/O device).

28.6.3 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* &optional *default* 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.

If the filename is relative, *default* is the directory to start with. If *default* is `nil` or missing, the current buffer's value of `default-directory` is used.

See Section 30.4 [Buffer File Name], page 434, for related information.

28.6.4 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 28.7 [Changing File Attributes], page 404).
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 50.5 [Time of Day], page 702.)
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 and the file's inode number together give enough information to distinguish any two files on the system—no two files can have the same values for both of these numbers.

For example, here are the file attributes for '`files.texi`':

```
(file-attributes "files.texi")
     ⇒  (nil
         1
         2235
         75
         (8489 20284)
         (8489 20284)
         (8489 20285)
         14906
         "-rw-rw-rw-"
         nil
         129500
         -32252)
```

and here is how the result is interpreted:

`nil` is neither a directory nor a symbolic link.

`1` has only one name (the name '`files.texi`' in the current default directory).

`2235` is owned by the user with UID 2235.

`75` is in the group with GID 75.

`(8489 20284)`
was last accessed on Aug 19 00:09. Use `format-time-string` to ! convert this number into a time string. See Section 50.6 [Time Conversion], page 703.

`(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.

28.7 Changing File Names and Attributes

The functions in this section rename, copy, delete, link, and set the modes of files.

In the functions that have an argument *newname*, if a file by the name of *newname* already exists, the actions taken depend on the value of the argument *ok-if-already-exists*:

- Signal a `file-already-exists` error if *ok-if-already-exists* is `nil`.
- Request confirmation if *ok-if-already-exists* is a number.
- Replace the old file without confirmation if *ok-if-already-exists* is any other value.

add-name-to-file *oldname newname* &optional *ok-if-already-exists* — Command

This function gives the file named *oldname* the additional name *newname*. This means that *newname* becomes a new "hard link" to *oldname*.

In the first part of the following example, we list two files, '`foo`' and '`foo3`'.

```
% ls -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 28.6.4 [File Attributes], page 401.

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. (On Unix, a file is deletable if its directory is writable.)

See also `delete-directory` in Section 28.10 [Create/Delete Dirs], page 413.

make-symbolic-link *filename newname* &optional *ok-if-exists* Command

This command makes a symbolic link to *filename*, named *newname*. This is like the shell command '`ln -s` *filename newname*'.

In an interactive call, this function prompts for *filename* and *newname* in the minibuffer; also, it requests confirmation if *newname* already exists.

define-logical-name *varname string* Function

This function defines the logical name *name* to have the value *string*. It is available only on VMS.

set-file-modes *filename mode* Function

This function sets mode bits of *filename* to *mode* (which must be an integer). Only the low 12 bits of *mode* are used.

set-default-file-modes *mode* Function

This function sets the default file protection for new files created by XEmacs and its subprocesses. Every file created with XEmacs 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 low 9 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.

On MS-DOS, there is no such thing as an "executable" file mode bit. So Emacs considers a file executable if its name ends in '`.com`', '`.bat`' or '`.exe`'. This is reflected in the values returned by `file-modes` and `file-attributes`.

28.8 File Names

Files are generally referred to by their names, in XEmacs as elsewhere. File names in XEmacs are represented as strings. The functions that operate on a file all expect a file name argument.

In addition to operating on files themselves, XEmacs 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. On MS-DOS, these functions understand MS-DOS file-name syntax as well as Unix syntax.

28.8.1 File Name Components

The operating system groups files into directories. To specify a file, you must specify the directory and the file's name within that directory. Therefore, XEmacs considers a file name as having two main parts: the *directory name* part, and the *nondirectory* part (or *file name within the directory*). Either part may be empty. Concatenating these two parts reproduces the original file name.

On Unix, the directory part is everything up to and including the last slash; the nondirectory part is the rest. The rules in VMS syntax are complicated.

For some purposes, the nondirectory part is further subdivided into the name proper and the *version number*. On Unix, only backup files have version numbers in their names; on VMS, every file has a version number, but most of the time the file name actually used in XEmacs 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* &optional *keep-backup-version* Function

This function returns *filename* without any file version numbers, backup version numbers, or trailing tildes.

If *keep-backup-version* is non-`nil`, we do not remove backup version numbers, only true file version numbers.

```
(file-name-sans-versions "~rms/foo.~1~")
     ⇒ "~rms/foo"
(file-name-sans-versions "~rms/foo~")
     ⇒ "~rms/foo"
(file-name-sans-versions "~rms/foo")
     ⇒ "~rms/foo"
;; The following example applies to VMS only.
(file-name-sans-versions "foo;23")
     ⇒ "foo"
```

file-name-sans-extension *filename* Function

This function returns *filename* minus its “extension,” if any. The extension, in a file name, is the part that starts with the last ‘`.`’ in the last name component. For example,

```
(file-name-sans-extension "foo.lose.c")
     ⇒ "foo.lose"
(file-name-sans-extension "big.hack/foo")
     ⇒ "big.hack/foo"
```

28.8.2 Directory Names

A *directory name* is the name of a directory. A directory is a kind of file, and it has a file name, which is related to the directory name but not identical to it. (This is not quite the same as the usual Unix terminology.) These two different names for the same entity are related by a syntactic transformation. On Unix, this is simple: a directory name ends in a slash, whereas the directory's name as a file lacks that slash. On VMS, the relationship is more complicated.

The difference between a directory name and its name as a file is subtle but crucial. When an XEmacs variable or function argument is described as being a directory name, a file name of a directory is not acceptable.

The following two functions convert between directory names and file names. They do nothing special with environment variable substitutions such as '`$HOME`', and the constructs '`~`', and '`..`'.

file-name-as-directory *filename* Function

This function returns a string representing *filename* in a form that the operating system will interpret as the name of a directory. In Unix, this means appending a slash to the string. 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, XEmacs shows users the abbreviation instead.

If you wish to convert a directory name to its abbreviation, use this function:

abbreviate-file-name *dirname* &optional *hack-homedir* Function

This function applies abbreviations from `directory-abbrev-alist` to its argument, and substitutes '`~`' for the user's home directory.

If *hack-homedir* is non-`nil`, then this also 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"))
```

28.8.3 Absolute and Relative File Names

All the directories in the file system form a tree starting at the root directory. A file name can specify all the directory names starting from the root of the tree; then it is called an *absolute* file name. Or it can specify the position of the file in the tree relative to a default directory; then it is called a *relative* file name. On Unix, an absolute file name starts with a slash or a tilde ('~'), and a relative one does not. The rules on VMS are complicated.

file-name-absolute-p *filename* Function

This function returns `t` if file *filename* is an absolute file name, `nil` otherwise. On VMS, this function understands both Unix syntax and VMS syntax.

```
(file-name-absolute-p "~rms/foo")
     ⇒ t
(file-name-absolute-p "rms/foo")
     ⇒ nil
(file-name-absolute-p "/user/rms/foo")
     ⇒ t
```

28.8.4 Functions that Expand Filenames

Expansion of a file name means converting a relative file name to an absolute one. Since this is done relative to a default directory, you must specify the default directory name as well as the file name to be expanded. Expansion also simplifies file names by eliminating redundancies such as '`./`' and '*name*`/../`'.

expand-file-name *filename* &optional *directory* Function

This function converts *filename* to an absolute file name. If *directory* is supplied, it is the directory to start with if *filename* is relative. (The value of *directory* should itself be an absolute directory name; it may start with '~'.) Otherwise, the current buffer's value of `default-directory` is used. For example:

```
(expand-file-name "foo")
     ⇒ "/xcssun/users/rms/lewis/foo"
(expand-file-name "../foo")
     ⇒ "/xcssun/users/rms/foo"
```

```
(expand-file-name "foo" "/usr/spool/")
     ⇒ "/usr/spool/foo"
(expand-file-name "$HOME/foo")
     ⇒ "/xcssun/users/rms/lewis/$HOME/foo"
```

Filenames containing '`.`' or '`..`' are simplified to their canonical form:

```
(expand-file-name "bar/../foo")
     ⇒ "/xcssun/users/rms/lewis/foo"
```

'`~/`' at the beginning is expanded into the user's home directory. A '`/`' or '`~`' following a '`/`'.

Note that `expand-file-name` does *not* expand environment variables; only `substitute-in-file-name` does that.

file-relative-name *filename* &optional *directory* Function

This function does the inverse of expansion—it tries to return a relative name that is equivalent to *filename* when interpreted relative to *directory*.

If *directory* is `nil` or omitted, the value of `default-directory` is used.

```
(file-relative-name "/foo/bar" "/foo/")
     ⇒ "bar")
(file-relative-name "/foo/bar" "/hack/")
     ⇒ "../foo/bar")
```

default-directory Variable

The value of this buffer-local variable is the default directory for the current buffer. It should be an absolute directory name; it may start with '`~`'. This variable is 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 variable references in *filename* with the environment variable values. Following standard Unix shell syntax, '`$`' is the prefix to substitute an environment variable value.

The environment variable name is the series of alphanumeric characters (including underscores) that follow the '`$`'. If the character following the '`$`' is a '`{`', then the variable name is everything up to the matching '`}`'.

Here we assume that the environment variable `HOME`, which holds the user's home directory name, has value '`/xcssun/users/rms`'.

```
(substitute-in-file-name "$HOME/foo")
     ⇒ "/xcssun/users/rms/foo"
```

After substitution, 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:

```
(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.

28.8.5 Generating Unique File Names

Some programs need to write temporary files. Here is the usual way to construct a name for such a file:

```
(make-temp-name (expand-file-name name-of-application (temp-directory)))
```

Here we use `(temp-directory)` to specify a directory for temporary files—under Unix, it will normally evaluate to '"/tmp/"'. The job of `make-temp-name` is to prevent two different users or two different processes from trying to use the same name.

temp-directory Function

This function returns the name of the directory to use for temporary files. Under Unix, this will be the value of `TMPDIR`, defaulting to '/tmp'. On Windows, this will be obtained from the `TEMP` or `TMP` environment variables, defaulting to '/'.

Note that the `temp-directory` function does not exist under FSF Emacs.

make-temp-name *prefix* Function

This function generates a temporary file name starting with *prefix*. The Emacs process number forms part of the result, so there is no danger of generating a name being used by another process.

```
(make-temp-name "/tmp/foo")
     ⇒ "/tmp/fooGaAQjC"
```

In addition, this function makes an attempt to choose a name that does not specify an existing file. To make this work, *prefix* should be an absolute file name.

To avoid confusion, each Lisp application should preferably use a unique *prefix* to `make-temp-name`.

28.8.6 File Name Completion

This section describes low-level subroutines for completing a file name. For other completion functions, see Section 18.5 [Completion], page 266.

file-name-all-completions *partial-filename directory* Function

This function returns a list of all possible completions for a file whose name starts with *partial-filename* in directory *directory*. The order of the completions is the order of the files in the directory, which is unpredictable and conveys no useful information.

The argument *partial-filename* must be a file name containing no directory part and no slash. The current buffer's default directory is prepended to *directory*, if *directory* is not absolute.

In the following example, suppose that 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")
```

28.9 Contents of Directories

A directory is a kind of file that contains other files entered under various names. Directories are a feature of the file system.

XEmacs 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 files-only* 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`, `directory-files` does not sort the list, so you get the file names in no particular order. Use this if you want the utmost possible speed and don't care what order the files are processed in. If the order of processing is visible to the user, then the user will probably be happier if you do sort the names.

If *files-only* is the symbol `t`, then only the "files" in the directory will be returned; subdirectories will be excluded. If *files-only* is not `nil` and not `t`, then only the subdirectories will be returned. Otherwise, if *files-only* is `nil` (the default) then both files and subdirectories will be returned.

```
(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.

insert-directory *file switches* &optional *wildcard full-directory-p* Function

This function inserts (in the current buffer) a directory listing for directory *file*, formatted with `ls` according to *switches*. It leaves point after the inserted text.

The argument *file* may be either a directory name or a file specification including wildcard characters. If *wildcard* is non-`nil`, that means treat *file* as a file specification with wildcards.

If *full-directory-p* is non-`nil`, that means *file* is a directory and switches do not contain '`-d`', so that the listing should show the full contents of the directory. (The '`-d`' option to `ls` says to describe a directory itself rather than its contents.)

This function works by running a directory listing program whose name is in the variable `insert-directory-program`. If *wildcard* is non-`nil`, it also runs the shell specified by `shell-file-name`, to expand the wildcards.

insert-directory-program Variable

This variable's value is the program to run to generate a directory listing for the function `insert-directory`.

28.10 Creating and Deleting Directories

Most XEmacs Lisp file-manipulation functions get errors when used on files that are directories. For example, you cannot delete a directory with `delete-file`. These special functions exist to create and delete directories.

make-directory *dirname* &optional *parents* Command

This function creates a directory named *dirname*. Interactively, the default choice of directory to create is the current default directory for file names. That is useful when you have visited a file in a nonexistent directory.

Non-interactively, optional argument *parents* says whether to create parent directories if they don't exist. (Interactively, this always happens.)

delete-directory *dirname* Command

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.

28.11 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 that match the regular expression), plus a handler that implements all the primitive XEmacs file operations for file names that do match.

The variable `file-name-handler-alist` holds a list of handlers, together with regular expressions that determine when to apply each handler. Each element has this form:

```
(regexp . handler)
```

All the XEmacs primitives for file access and file name transformation check the given file name against `file-name-handler-alist`. If the file name matches *regexp*, the primitives handle that file by calling *handler*.

The first argument given to *handler* is the name of the primitive; the remaining arguments are the arguments that were passed to that operation. (The first of these arguments is typically the file name itself.) For example, if you do this:

```
(file-exists-p filename)
```

and *filename* has handler *handler*, then *handler* is called like this:

```
(funcall handler 'file-exists-p filename)
```

Here are the operations that a magic file name handler gets to handle:

`add-name-to-file`, `copy-file`, `delete-directory`, `delete-file`, `diff-latest-backup-file`, `directory-file-name`, `directory-files`, `dired-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-regular-p`, `file-symlink-p`, `file-truename`, `file-writable-p`, `get-file-buffer`, `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`.

Handlers for `insert-file-contents` typically need to clear the buffer's modified flag, with `(set-buffer-modified-p nil)`, if the *visit* argument is non-`nil`. This also has the effect of unlocking the buffer if it is locked.

The handler function must handle all of the above operations, and possibly others to be added in the future. It need not implement all these operations itself—when it has nothing special to do for a certain operation, it can reinvoke the primitive, to handle the operation "in the usual way". It should always reinvoke the primitive for an operation it does not recognize. Here's one way to do this:

```
(defun my-file-handler (operation &rest args)
  ;; First check for the specific operations
  ;; that we have special handling for.
  (cond ((eq operation 'insert-file-contents) ...)
        ((eq operation 'write-region) ...)
        ...
        ;; Handle any operation we don't know about.
        (t (let ((inhibit-file-name-handlers
                  (cons 'my-file-handler
                        (and (eq inhibit-file-name-operation operation)
                             inhibit-file-name-handlers)))
                 (inhibit-file-name-operation operation))
             (apply operation args)))))
```

When a handler function decides to call the ordinary Emacs primitive for the operation at hand, it needs to prevent the primitive from calling the same handler once again, thus leading to an infinite recursion. The example above shows how to do this, with the variables `inhibit-file-name-handlers` and `inhibit-file-name-operation`. Be careful to use them exactly as shown above; the details are crucial for proper behavior in the case of multiple handlers, and for operations that have two file names that may each have handlers.

inhibit-file-name-handlers Variable

This variable holds a list of handlers whose use is presently inhibited for a certain operation.

inhibit-file-name-operation Variable

The operation for which certain handlers are presently inhibited.

find-file-name-handler *file operation* Function

This function returns the handler function for file name *file*, or `nil` if there is none. The argument *operation* should be the operation to be performed on the file—the value you will pass to the handler as its first argument when you call it. The operation is needed for comparison with `inhibit-file-name-operation`.

file-local-copy *filename* Function

This function copies file *filename* to an ordinary non-magic file, if it isn't one already.

If *filename* specifies a "magic" file name, which programs outside Emacs cannot directly read or write, this copies the contents to an ordinary file and returns that file's name.

If *filename* is an ordinary file name, not magic, then this function does nothing and returns `nil`.

unhandled-file-name-directory *filename* Function

This function returns the name of a directory that is not magic. It uses the directory part of *filename* if that is not magic. Otherwise, it asks the handler what to do.

This is useful for running a subprocess; every subprocess must have a non-magic directory to serve as its current directory, and this function is a good way to come up with one.

28.12 Partial Files

28.12.1 Intro to Partial Files

A *partial file* is a section of a buffer (called the *master buffer*) that is placed in its own buffer and treated as its own file. Changes made to the partial file are not reflected in the master buffer until the partial file is "saved" using the standard buffer save commands. Partial files can be "reverted" (from the master buffer) just like normal files. When a file part is active on a master buffer, that section of the master buffer is marked as read-only. Two file parts on the same master buffer are not allowed to overlap. Partial file buffers are indicated by the words '`File Part`' in the modeline.

The master buffer knows about all the partial files that are active on it, and thus killing or reverting the master buffer will be handled properly. When the master buffer is saved, if there are any unsaved partial files active on it then the user will be given the opportunity to first save these files.

When a partial file buffer is first modified, the master buffer is automatically marked as modified so that saving the master buffer will work correctly.

28.12.2 Creating a Partial File

make-file-part &optional *start end name buffer* Function

Make a file part on buffer *buffer* out of the region. Call it *name*. This command creates a new buffer containing the contents of the region and marks the buffer as referring to the specified buffer, called the *master buffer*. When the file-part buffer is saved, its changes are integrated back into the master buffer. When the master buffer is deleted, all file parts are deleted with it.

When called from a function, expects four arguments, *start*, *end*, *name*, and *buffer*, all of which are optional and default to the beginning of *buffer*, the end of *buffer*, a name generated from *buffer* name, and the current buffer, respectively.

28.12.3 Detached Partial Files

Every partial file has an extent in the master buffer associated with it (called the *master extent*), marking where in the master buffer the partial file begins and ends. If the text in master buffer that is contained by the extent is deleted, then the extent becomes "detached", meaning that it no longer refers to a specific region of the master buffer. This can happen either when the text is deleted directly or when the master buffer is reverted. Neither of these should happen in normal usage because the master buffer should generally not be edited directly.

Before doing any operation that references a partial file's master extent, XEmacs checks to make sure that the extent is not detached. If this is the case, XEmacs warns the user of this and the master extent is deleted out of the master buffer, disconnecting the file part. The file part's filename is cleared and thus must be explicitly specified if the detached file part is to be saved.

28.13 File Format Conversion

The variable `format-alist` defines a list of *file formats*, which describe textual representations used in files for the data (text, text-properties, and possibly other information) in an Emacs buffer. Emacs performs format conversion if appropriate when reading and writing files.

format-alist Variable

This list contains one format definition for each defined file format.

Each format definition is a list of this form:

(*name doc-string regexp from-fn to-fn modify mode-fn*)

Here is what the elements in a format definition mean:

name The name of this format.

doc-string A documentation string for the format.

regexp A regular expression which is used to recognize files represented in this format.

from-fn A function to call to decode data in this format (to convert file data into the usual Emacs data representation).

The *from-fn* is called with two args, *begin* and *end*, which specify the part of the buffer it should convert. It should convert the text by editing it in place. Since this can change the length of the text, *from-fn* should return the modified end position.

One responsibility of *from-fn* is to make sure that the beginning of the file no longer matches *regexp*. Otherwise it is likely to get called again.

to-fn A function to call to encode data in this format (to convert the usual Emacs data representation into this format).

The *to-fn* is called with two args, *begin* and *end*, which specify the part of the buffer it should convert. There are two ways it can do the conversion:

- By editing the buffer in place. In this case, *to-fn* should return the end-position of the range of text, as modified.
- By returning a list of annotations. This is a list of elements of the form (*position* . *string*), where *position* is an integer specifying the relative position in the text to be written, and *string* is the annotation to add there. The list must be sorted in order of position when *to-fn* returns it.

When `write-region` actually writes the text from the buffer to the file, it intermixes the specified annotations at the corresponding positions. All this takes place without modifying the buffer.

modify A flag, `t` if the encoding function modifies the buffer, and `nil` if it works by returning a list of annotations.

mode A mode function to call after visiting a file converted from this format.

The function `insert-file-contents` automatically recognizes file formats when it reads the specified file. It checks the text of the beginning of the file against the regular expressions of the format definitions, and if it finds a match, it calls the decoding function for that format. Then it checks all the known formats over again. It keeps checking them until none of them is applicable.

Visiting a file, with `find-file-noselect` or the commands that use it, performs conversion likewise (because it calls `insert-file-contents`); it also calls the mode function for each format that it decodes. It stores a list of the format names in the buffer-local variable `buffer-file-format`.

buffer-file-format Variable

This variable states the format of the visited file. More precisely, this is a list of the file format names that were decoded in the course of visiting the current buffer's file. It is always local in all buffers.

When `write-region` writes data into a file, it first calls the encoding functions for the formats listed in `buffer-file-format`, in the order of appearance in the list.

format-write-file *file format* Function

This command writes the current buffer contents into the file *file* in format *format*, and makes that format the default for future saves of the buffer. The argument *format* is a list of format names.

format-find-file *file format* Function

This command finds the file *file*, converting it according to format *format*. It also makes *format* the default if the buffer is saved later.

The argument *format* is a list of format names. If *format* is `nil`, no conversion takes place. Interactively, typing just ⟨RET⟩ for *format* specifies `nil`.

format-insert-file *file format* &optional *beg end* Function

This command inserts the contents of file *file*, converting it according to format *format*. If *beg* and *end* are non-`nil`, they specify which part of the file to read, as in `insert-file-contents` (see Section 28.3 [Reading from Files], page 396).

The return value is like what `insert-file-contents` returns: a list of the absolute file name and the length of the data inserted (after conversion).

The argument *format* is a list of format names. If *format* is `nil`, no conversion takes place. Interactively, typing just ⟨RET⟩ for *format* specifies `nil`.

format-find-file *file format* Function

This command finds the file *file*, converting it according to format *format*. It also makes *format* the default if the buffer is saved later.

The argument *format* is a list of format names. If *format* is `nil`, no conversion takes place. Interactively, typing just ⟨RET⟩ for *format* specifies `nil`.

format-insert-file *file format* &optional *beg end* Function

This command inserts the contents of file *file*, converting it according to format *format*. If *beg* and *end* are non-`nil`, they specify which part of the file to read, as in `insert-file-contents` (see Section 28.3 [Reading from Files], page 396).

The return value is like what `insert-file-contents` returns: a list of the absolute file name and the length of the data inserted (after conversion).

The argument *format* is a list of format names. If *format* is `nil`, no conversion takes place. Interactively, typing just ⟨RET⟩ for *format* specifies `nil`.

auto-save-file-format Variable

This variable specifies the format to use for auto-saving. Its value is a list of format names, just like the value of `buffer-file-format`; but it is used instead of `buffer-file-format` for writing auto-save files. This variable is always local in all buffers.

28.14 Files and MS-DOS

Emacs on MS-DOS makes a distinction between text files and binary files. This is necessary because ordinary text files on MS-DOS use a two character sequence between lines: carriage-return and linefeed (CRLF). Emacs expects just a newline character (a linefeed) between lines. When Emacs reads or writes a text file on MS-DOS, it needs to convert the line separators. This means it needs to know which files are text files and which are binary. It makes this decision when visiting a file, and records the decision in the variable `buffer-file-type` for use when the file is saved.

See Section 49.3 [MS-DOS Subprocesses], page 677, for a related feature for subprocesses.

buffer-file-type Variable

This variable, automatically local in each buffer, records the file type of the buffer's visited file. The value is `nil` for text, `t` for binary.

find-buffer-file-type *filename* Function

This function determines whether file *filename* is a text file or a binary file. It returns `nil` for text, `t` for binary.

file-name-buffer-file-type-alist User Option

This variable holds an alist for distinguishing text files from binary files. Each element has the form (*regexp* . *type*), where *regexp* is matched against the file name, and *type* may be is `nil` for text, `t` for binary, or a function to call to compute which. If it is a function, then it is called with a single argument (the file name) and should return `t` or `nil`.

default-buffer-file-type User Option

This variable specifies the default file type for files whose names don't indicate anything in particular. Its value should be `nil` for text, or `t` for binary.

find-file-text *filename* Command

Like `find-file`, but treat the file as text regardless of its name.

find-file-binary *filename* Command

Like `find-file`, but treat the file as binary regardless of its name.

29 Backups and Auto-Saving

Backup files and auto-save files are two methods by which XEmacs 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.

29.1 Backup Files

A *backup file* is a copy of the old contents of a file you are editing. XEmacs 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, XEmacs 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 XEmacs can delete them automatically.

29.1.1 Making Backup Files

backup-buffer Function

This function makes a backup of the file visited by the current buffer, if appropriate. It is called by `save-buffer` before saving the buffer the first time.

buffer-backed-up Variable

This buffer-local variable indicates whether this buffer's file has been backed up on account of this buffer. If it is non-`nil`, then the backup file has been written. Otherwise, the file should be backed up when it is next saved (if backups are enabled). This is a permanent local; `kill-local-variables` does not alter it.

make-backup-files User Option

This variable determines whether or not to make backup files. If it is non-`nil`, then XEmacs creates a backup of each file when it is saved for the first time—provided that `backup-inhibited` is `nil` (see below).

The following example shows how to change the `make-backup-files` variable only in the '`RMAIL`' buffer and not elsewhere. Setting it `nil` stops XEmacs 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 Variable

This variable's value is a function to be called on certain occasions to decide whether a file should have backup files. The function receives one argument, a file name to consider. If the function returns `nil`, backups are disabled for that file. Otherwise, the other variables in this section say whether and how to make backups.

The default value is this:

```
(lambda (name)
  (or (< (length name) 5)
      (not (string-equal "/tmp/"
                         (substring name 0 5)))))
```

backup-inhibited Variable

If this variable is non-`nil`, backups are inhibited. It records the result of testing `backup-enable-predicate` on the visited file name. It can also coherently be used by other mechanisms that inhibit backups based on which file is visited. For example, VC sets this variable non-`nil` to prevent making backups for files managed with a version control system.

This is a permanent local, so that changing the major mode does not lose its value. Major modes should not set this variable—they should set `make-backup-files` instead.

29.1.2 Backup by Renaming or by Copying?

There are two ways that XEmacs can make a backup file:

- XEmacs 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.
- XEmacs 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 28.2 [Saving Buffers], page 394.

backup-by-copying Variable

If this variable is non-`nil`, XEmacs always makes backup files by copying.

The following two variables, when non-`nil`, cause the second method to be used in certain special cases. They have no effect on the treatment of files that don't fall into the special cases.

backup-by-copying-when-linked Variable

If this variable is non-`nil`, XEmacs makes backups by copying for files with multiple names (hard links).

This variable is significant only if `backup-by-copying` is `nil`, since copying is always used when that variable is non-`nil`.

backup-by-copying-when-mismatch Variable

If this variable is non-`nil`, XEmacs makes backups by copying in cases where renaming would change either the owner or the group of the file.

The value has no effect when renaming would not alter the owner or group of the file; that is, for files which are owned by the user and whose group matches the default for a new file created there by the user.

This variable is significant only if `backup-by-copying` is `nil`, since copying is always used when that variable is non-`nil`.

29.1.3 Making and Deleting Numbered Backup Files

If a file's name is '`foo`', the names of its numbered backup versions are '`foo.~`*v*`~`', for various integers *v*, like this: '`foo.~1~`', '`foo.~2~`', '`foo.~3~`', ..., '`foo.~259~`', and so on.

version-control User Option

This variable controls whether to make a single non-numbered backup file or multiple numbered backups.

`nil` Make numbered backups if the visited file already has numbered backups; otherwise, do not.

`never` Do not make numbered backups.

anything else
Make numbered backups.

The use of numbered backups ultimately leads to a large number of backup versions, which must then be deleted. XEmacs can do this automatically or it can ask the user whether to delete them.

kept-new-versions User Option

The value of this variable is the number of newest versions to keep when a new numbered backup is made. The newly made backup is included in the count. The default value is 2.

kept-old-versions User Option

The value of this variable is the number of oldest versions to keep when a new numbered backup is made. The default value is 2.

If there are backups numbered 1, 2, 3, 5, and 7, and both of these variables have the value 2, then the backups numbered 1 and 2 are kept as old versions and those numbered 5 and 7 are kept as new versions; backup version 3 is excess. The function `find-backup-file-name` (see Section 29.1.4 [Backup Names], page 424) 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.

dired-kept-versions User Option

This variable specifies how many of the newest backup versions to keep in the Dired command . (`dired-clean-directory`). That's the same thing `kept-new-versions` specifies when you make a new backup file. The default value is 2.

29.1.4 Naming Backup Files

The functions in this section are documented mainly because you can customize the naming conventions for backup files by redefining them. If you change one, you probably need to change the rest.

backup-file-name-p *filename* Function

This function returns a non-`nil` value if *filename* is a possible name for a backup file. A file with the name *filename* need not exist; the function just checks the name.

```
(backup-file-name-p "foo")
     ⇒ nil
(backup-file-name-p "foo~")
     ⇒ 3
```

The standard definition of this function is as follows:

```
(defun backup-file-name-p (file)
  "Return non-nil if FILE is a backup file \
name (numeric or not)..."
  (string-match "~$" file))
```

Thus, the function returns a non-`nil` value if the file name ends with a '~'. (We use a backslash to split the documentation string's first line into two lines in the text, but produce just one line in the string itself.)

This simple expression is placed in a separate function to make it easy to redefine for customization.

make-backup-file-name *filename* Function

This function returns a string that is the name to use for a non-numbered backup file for file *filename*. On Unix, this is just *filename* with a tilde appended.

The standard definition of this function is as follows:

```
(defun make-backup-file-name (file)
  "Create the non-numeric backup file name for FILE.
...
"
  (concat file "~"))
```

You can change the backup-file naming convention by redefining this function. The following example redefines `make-backup-file-name` to prepend a '.' in addition to appending a tilde:

```
(defun make-backup-file-name (filename)
  (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 backup versions should be kept. This function keeps those versions by excluding them from the CDR of the value. See Section 29.1.3 [Numbered Backups], page 423.

In this example, the value says that '`~rms/foo.~5~`' is the name to use for the new backup file, and '`~rms/foo.~3~`' is an "excess" version that the caller should consider deleting now.

```
(find-backup-file-name "~rms/foo")
     ⇒ ("~rms/foo.~5~" "~rms/foo.~3~")
```

file-newest-backup *filename* Function

This function returns the name of the most recent backup file for *filename*, or `nil` if that file has no backup files.

Some file comparison commands use this function so that they can automatically compare a file with its most recent backup.

29.2 Auto-Saving

XEmacs 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 XEmacs Reference Manual*, for information on auto-save for users. Here we describe the functions used to implement auto-saving and the variables that control them.

buffer-auto-save-file-name Variable

This buffer-local variable is the name of the file used for auto-saving the current buffer. It is `nil` if the buffer should not be auto-saved.

```
buffer-auto-save-file-name
=> "/xcssun/users/rms/lewis/#files.texi#"
```

auto-save-mode *arg* Command

When used interactively without an argument, this command is a toggle switch: it turns on auto-saving of the current buffer if it is off, and vice-versa. With an argument *arg*, the command turns auto-saving on if the value of *arg* is `t`, a nonempty list, or a positive integer. Otherwise, it turns auto-saving off.

auto-save-file-name-p *filename* Function

This function returns a non-`nil` value if *filename* is a string that could be the name of an auto-save file. It works based on knowledge of the naming convention for auto-save files: a name that begins and ends with hash marks ('`#`') is a possible auto-save file name. The argument *filename* should not contain a directory part.

```
(make-auto-save-file-name)
     ⇒ "/xcssun/users/rms/lewis/#files.texi#"
(auto-save-file-name-p "#files.texi#")
     ⇒ 0
(auto-save-file-name-p "files.texi")
     ⇒ nil
```

The standard definition of this function is as follows:

```
(defun auto-save-file-name-p (filename)
  "Return non-nil if FILENAME can be yielded by..."
  (string-match "^#.*#$" filename))
```

This function exists so that you can customize it if you wish to change the naming convention for auto-save files. If you redefine it, be sure to redefine the function `make-auto-save-file-name` correspondingly.

make-auto-save-file-name Function

This function returns the file name to use for auto-saving the current buffer. This is just the file name with hash marks ('`#`') appended and prepended to it. This function does not look at the variable `auto-save-visited-file-name` (described below); you should check that before calling this function.

```
(make-auto-save-file-name)
     ⇒ "/xcssun/users/rms/lewis/#backup.texi#"
```

The standard definition of this function is as follows:

```
(defun make-auto-save-file-name ()
  "Return file name to use for auto-saves \
of current buffer.
...
"
  (if buffer-file-name
      (concat
       (file-name-directory buffer-file-name)
       "#"
       (file-name-nondirectory buffer-file-name)
       "#")
    (expand-file-name
     (concat "#%" (buffer-name) "#"))))
```

This exists as a separate function so that you can redefine it to customize the naming convention for auto-save files. Be sure to change `auto-save-file-name-p` in a corresponding way.

auto-save-visited-file-name Variable

If this variable is non-`nil`, XEmacs auto-saves buffers in the files they are visiting. That is, the auto-save is done in the same file that you are editing.

Normally, this variable is `nil`, so auto-save files have distinct names that are created by `make-auto-save-file-name`.

When you change the value of this variable, the value does not take effect until the next time auto-save mode is reenabled in any given buffer. If auto-save mode is already enabled, auto-saves continue to go in the same file name until `auto-save-mode` is called again.

recent-auto-save-p Function

This function returns `t` if the current buffer has been auto-saved since the last time it was read in or saved.

set-buffer-auto-saved Function

This function marks the current buffer as auto-saved. The buffer will not be auto-saved again until the buffer text is changed again. The function returns `nil`.

auto-save-interval User Option

The value of this variable is the number of characters that XEmacs 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, XEmacs auto-saves any buffers that need it. (Actually, the specified timeout is multiplied by a factor depending on the size of the current buffer.)

auto-save-hook Variable

This normal hook is run whenever an auto-save is about to happen.

auto-save-default User Option

If this variable is non-`nil`, buffers that are visiting files have auto-saving enabled by default. Otherwise, they do not.

do-auto-save &optional *no-message current-only* Command

This function auto-saves all buffers that need to be auto-saved. It saves all buffers for which auto-saving is enabled and that have been changed since the previous auto-save.

Normally, if any buffers are auto-saved, a message that says '`Auto-saving...`' is displayed in the echo area while auto-saving is going on. However, if *no-message* is non-`nil`, the message is inhibited.

If *current-only* is non-`nil`, only the current buffer is auto-saved.

delete-auto-save-file-if-necessary Function

This function deletes the current buffer's auto-save file if `delete-auto-save-files` is non-`nil`. It is called every time a buffer is saved.

delete-auto-save-files Variable

This variable is used by the function `delete-auto-save-file-if-necessary`. If it is non-`nil`, Emacs deletes auto-save files when a true save is done (in the visited file). This saves disk space and unclutters your directory.

rename-auto-save-file Function

This function adjusts the current buffer's auto-save file name if the visited file name has changed. It also renames an existing auto-save file. If the visited file name has not changed, this function does nothing.

buffer-saved-size Variable

The value of this buffer-local variable is the length of the current buffer as of the last time it was read in, saved, or auto-saved. This is used to detect a substantial decrease in size, and turn off auto-saving in response.

If it is -1, that means auto-saving is temporarily shut off in this buffer due to a substantial deletion. Explicitly saving the buffer stores a positive value in this variable, thus reenabling auto-saving. Turning auto-save mode off or on also alters this variable.

auto-save-list-file-name Variable

This variable (if non-`nil`) specifies a file for recording the names of all the auto-save files. Each time XEmacs does auto-saving, it writes two lines into this file for each buffer that has auto-saving enabled. The first line gives the name of the visited file (it's empty if the buffer has none), and the second gives the name of the auto-save file.

If XEmacs exits normally, it deletes this file. If XEmacs crashes, you can look in the file to find all the auto-save files that might contain work that was otherwise lost. The `recover-session` command uses these files.

The default name for this file is in your home directory and starts with '`.saves-`'. It also contains the XEmacs process ID and the host name.

29.3 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 XEmacs Reference 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.

Normally, `revert-buffer` asks for confirmation before it changes the buffer; but if the argument *noconfirm* is non-`nil`, `revert-buffer` does not ask for confirmation.

Reverting tries to preserve marker positions in the buffer by using the replacement feature of `insert-file-contents`. If the buffer contents and the file contents are identical before the revert operation, reverting preserves all the markers. If they are not identical, reverting does change the buffer; then it preserves the markers in the unchanged text (if any) at the beginning and end of the buffer. Preserving any additional markers would be problematical.

You can customize how `revert-buffer` does its work by setting these variables—typically, as buffer-local variables.

revert-buffer-function Variable

The value of this variable is the function to use to revert this buffer. If non-`nil`, it is called as a function with no arguments to do the work of reverting. If the value is `nil`, reverting works the usual way.

Modes such as Dired mode, in which the text being edited does not consist of a file's contents but can be regenerated in some other fashion, give this variable a buffer-local value that is a function to regenerate the contents.

revert-buffer-insert-file-contents-function Variable

The value of this variable, if non-`nil`, is the function to use to insert the updated contents when reverting this buffer. The function receives two arguments: first the file name to use; second, `t` if the user has asked to read the auto-save file.

before-revert-hook Variable

This normal hook is run by `revert-buffer` before actually inserting the modified contents—but only if `revert-buffer-function` is `nil`.

Font Lock mode uses this hook to record that the buffer contents are no longer fontified.

after-revert-hook Variable

This normal hook is run by `revert-buffer` after actually inserting the modified contents—but only if `revert-buffer-function` is `nil`.

Font Lock mode uses this hook to recompute the fonts for the updated buffer contents.

30 Buffers

A *buffer* is a Lisp object containing text to be edited. Buffers are used to hold the contents of files that are being visited; there may also be buffers that are not visiting files. While several buffers may exist at one time, exactly one buffer is designated the *current buffer* at any time. Most editing commands act on the contents of the current buffer. Each buffer, including the current buffer, may or may not be displayed in any windows.

30.1 Buffer Basics

Buffers in Emacs editing are objects that have distinct names and hold text that can be edited. Buffers appear to Lisp programs as a special data type. You can think of the contents of a buffer as an extendable string; insertions and deletions may occur in any part of the buffer. See Chapter 36 [Text], page 511.

A Lisp buffer object contains numerous pieces of information. Some of this information is directly accessible to the programmer through variables, while other information is accessible only through special-purpose functions. For example, the visited file name is directly accessible through a variable, while the value of point is accessible only through a primitive function.

Buffer-specific information that is directly accessible is stored in *buffer-local* variable bindings, which are variable values that are effective only in a particular buffer. This feature allows each buffer to override the values of certain variables. Most major modes override variables such as `fill-column` or `comment-column` in this way. For more information about buffer-local variables and functions related to them, see Section 10.9 [Buffer-Local Variables], page 157.

For functions and variables related to visiting files in buffers, see Section 28.1 [Visiting Files], page 391 and Section 28.2 [Saving Buffers], page 394. For functions and variables related to the display of buffers in windows, see Section 31.6 [Buffers and Windows], page 453.

bufferp *object* Function

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

30.2 The Current Buffer

There are, in general, many buffers in an Emacs session. At any time, one of them is designated as the *current buffer*. This is the buffer in which most editing takes place, because most of the primitives for examining or changing text in a buffer operate implicitly on the current buffer (see Chapter 36 [Text], page 511). 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 buffer that the command will apply to. (See Chapter 19 [Command Loop], page 281.) Therefore, `set-buffer` is not the way to switch visibly to a different buffer so that the user can edit it. For this, you must use the functions described in Section 31.7 [Displaying Buffers], page 453.

However, Lisp functions that change to a different current buffer should not depend on the command loop to set it back afterwards. Editing commands written in XEmacs 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 function is done (see Section 34.3 [Excursions], page 495). 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, the next redisplay will show how its text has changed. Otherwise, you will not see the change immediately on the screen. The buffer becomes current temporarily during the execution of the command, but this does not cause it to be displayed.

If you make local bindings (with `let` or function arguments) for a variable that may also have buffer-local bindings, make sure that the same buffer is current at the beginning and at the end of the local binding's scope. Otherwise you might bind it in one buffer and unbind it in another! There are two ways to do this. In simple cases, you may see that nothing ever changes the current buffer within the scope of the binding. Otherwise, use `save-excursion` to make sure that the buffer current at the beginning is current again whenever the variable is unbound.

It is not reliable to change the current buffer back with `set-buffer`, because that won't do the job if a quit happens while the wrong buffer is current. Here is what *not* to do:

```
     (let (buffer-read-only
           (obuf (current-buffer)))
       (set-buffer ...)
       ...
       (set-buffer obuf))
```

Using `save-excursion`, as shown below, handles quitting, errors, and `throw`, as well as ordinary evaluation.

```
(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. It does not display the buffer in the currently selected window or in any other window, so the user cannot necessarily see the buffer. But Lisp programs can in any case work on it.

This function returns the buffer identified by *buffer-or-name*. An error is signaled if *buffer-or-name* does not identify an existing buffer.

30.3 Buffer Names

Each buffer has a unique name, which is a string. Many of the functions that work on buffers accept either a buffer or a buffer name as an argument. Any argument called *buffer-or-name* is of this sort, and an error is signaled if it is neither a string nor a buffer. Any argument called *buffer* must be an actual buffer object, not a name.

Buffers that are ephemeral and generally uninteresting to the user have names starting with a space, so that the `list-buffers` and `buffer-menu` commands don't mention them. A name starting with space also initially disables recording undo information; see Section 36.9 [Undo], page 523.

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 30.10 [Killing Buffers], page 441.

```
(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 30.9 [Creating Buffers], page 440.

generate-new-buffer-name *starting-name* &optional *ignore* Function

This function returns a name that would be unique for a new buffer—but does not create the buffer. It starts with *starting-name*, and produces a name not currently in use for any buffer by appending a number inside of '`<...>`'.

If *ignore* is given, it specifies a name that is okay to use (if it is in the sequence to be tried), even if a buffer with that name exists.

See the related function `generate-new-buffer` in Section 30.9 [Creating Buffers], page 440.

30.4 Buffer File Name

The *buffer file name* is the name of the file that is visited in that buffer. When a buffer is not visiting a file, its buffer file name is `nil`. Most of the time, the buffer name is the same as the nondirectory part of the buffer file name, but the buffer file name and the buffer name are distinct and can be set independently. See Section 28.1 [Visiting Files], page 391.

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 XEmacs.

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 28.6.3 [Truenames], page 401.

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 28.6.3 [Truenames], page 401.

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 28.6.4 [File Attributes], page 401, 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 28.8.4 [File Name Expansion], page 409), 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 XEmacs 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 30.5 [Buffer Modification], page 436.

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.

30.5 Buffer Modification

XEmacs 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 modeline (see Section 26.3.2 [Modeline Variables], page 374), and controls saving (see Section 28.2 [Saving Buffers], page 394) and auto-saving (see Section 29.2 [Auto-Saving], page 425).

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 36 [Text], page 511.

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 modeline for the current buffer. In fact, the function `redraw-modeline` works by doing this:

```
(set-buffer-modified-p (buffer-modified-p))
```

not-modified &optional *arg* Command

This command marks the current buffer as unmodified, and not needing to be saved. (If *arg* is non-`nil`, the buffer is instead marked as modified.) 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.

30.6 Comparison of Modification Time

Suppose that you visit a file and make changes in its buffer, and meanwhile the file itself is changed on disk. At this point, saving the buffer would overwrite the changes in the file. Occasionally this may be what you want, but usually it would lose valuable information. XEmacs therefore checks the file's modification time using the functions described below before saving the file.

verify-visited-file-modtime *buffer* Function

This function compares what *buffer* has recorded for the modification time of its visited file against the actual modification time of the file as recorded by the operating system. The two should be the same unless some other process has written the file since XEmacs visited or saved it.

The function returns `t` if the last actual modification time and XEmacs's recorded modification time are the same, `nil` otherwise.

clear-visited-file-modtime Function

This function clears out the record of the last modification time of the file being visited by the current buffer. As a result, the next attempt to save this buffer will not complain of a discrepancy in file modification times.

This function is called in `set-visited-file-name` and other exceptional places where the usual test to avoid overwriting a changed file should not be done.

visited-file-modtime Function

This function returns the buffer's recorded last file modification time, as a list of the form `(`*high* `.` *low*`)`. (This is the same format that `file-attributes` uses to return time values; see Section 28.6.4 [File Attributes], page 401.)

set-visited-file-modtime &optional *time* Function

This function updates the buffer's record of the last modification time of the visited file, to the value specified by *time* if *time* is not `nil`, and otherwise to the last modification time of the visited file.

If *time* is not `nil`, it should have the form `(`*high* `.` *low*`)` or `(`*high low*`)`, in either case containing two integers, each of which holds 16 bits of the time.

This function is useful if the buffer was not read from the file normally, or if the file itself has been changed for some known benign reason.

ask-user-about-supersession-threat *filename* Function

This function is used to ask a user how to proceed after an attempt to modify an obsolete buffer visiting file *filename*. An *obsolete buffer* is an unmodified buffer for which the associated file on disk is newer than the last save-time of the buffer. This means some other program has probably altered the file.

Depending on the user's answer, the function may return normally, in which case the modification of the buffer proceeds, or it may signal a `file-supersession` error with data (*filename*), in which case the proposed buffer modification is not allowed.

This function is called automatically by XEmacs on the proper occasions. It exists so you can customize XEmacs by redefining it. See the file '`userlock.el`' for the standard definition.

See also the file locking mechanism in Section 28.5 [File Locks], page 398.

30.7 Read-Only Buffers

If a buffer is *read-only*, then you cannot change its contents, although you may change your view of the contents by scrolling and narrowing.

Read-only buffers are used in two kinds of situations:

- A buffer visiting a write-protected file is normally read-only.

  Here, the purpose is to 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 *C-x C-q*.
- Modes such as Dired and Rmail make buffers read-only when altering the contents with the usual editing commands is probably a mistake.

  The special commands of these modes bind `buffer-read-only` to `nil` (with `let`) or bind `inhibit-read-only` to `t` around the places where they change the text.

buffer-read-only Variable

This buffer-local variable specifies whether the buffer is read-only. The buffer is read-only if this variable is non-`nil`.

inhibit-read-only Variable

If this variable is non-`nil`, then read-only buffers and read-only characters may be modified. Read-only characters in a buffer are those that have non-`nil` `read-only` properties (either text properties or extent properties). See Section 40.6 [Extent Properties], page 591, for more information about text properties and extent properties.

If `inhibit-read-only` is `t`, all `read-only` character properties have no effect. If `inhibit-read-only` is a list, then `read-only` character properties have no effect if they are members of the list (comparison is done with `eq`).

toggle-read-only Command

This command changes whether the current buffer is read-only. It is intended for interactive use; don't use it in programs. At any given point in a program, you should know whether you want the read-only flag on or off; so you can set `buffer-read-only` explicitly to the proper value, `t` or `nil`.

barf-if-buffer-read-only Function

This function signals a `buffer-read-only` error if the current buffer is read-only. See Section 19.3 [Interactive Call], page 286, for another way to signal an error if the current buffer is read-only.

30.8 The Buffer List

The *buffer list* is a list of all live buffers. Creating a buffer adds it to this list, and killing a buffer deletes it. The order of the buffers in the list is based primarily on how recently each buffer has been displayed in the selected window. Buffers move to the front of the list when they are selected and to the end when they are buried. Several functions, notably `other-buffer`, use this ordering. A buffer list displayed for the user also follows this order.

Every frame has its own order for the buffer list. Switching to a new buffer inside of a particular frame changes the buffer list order for that frame, but does not affect the buffer list order of any other frames. In addition, there is a global, non-frame buffer list order that is independent of the buffer list orders for any particular frame.

Note that the different buffer lists all contain the same elements. It is only the order of those elements that is different.

buffer-list &optional *frame* 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. The order of the list is specific to *frame*, which defaults to the current frame. If *frame* is `t`, the global, non-frame ordering is returned instead.

```
(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 XEmacs; modifying it has no effect on the buffers.

other-buffer &optional *buffer-or-name frame visible-ok* Function

This function returns the first buffer in the buffer list other than *buffer-or-name*, in *frame*'s ordering for the buffer list. (*frame* defaults to the current frame. If *frame* is `t`, then the global, non-frame ordering is used.) 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 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.

If the selected frame has a non-`nil` `buffer-predicate` property, then `other-buffer` uses that predicate to decide which buffers to consider. It calls the predicate once for each buffer, and if the value is `nil`, that buffer is ignored. See Section 32.2.3 [X Frame Properties], page 473.

If *visible-ok* is `nil`, `other-buffer` avoids returning a buffer visible in any window on any visible frame, except as a last resort. If *visible-ok* is non-`nil`, then it does not matter whether a buffer is displayed somewhere or not.

If no suitable buffer exists, the buffer '`*scratch*`' is returned (and created, if necessary).

Note that in FSF Emacs 19, there is no *frame* argument, and *visible-ok* is the second argument instead of the third. FSF Emacs 19.

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 XEmacs Reference 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.

If *buffer-or-name* is `nil` or omitted, this means to bury the current buffer. In addition, if the buffer is displayed in the selected window, this switches to some other buffer (obtained using `other-buffer`) in the selected window. But if the buffer is displayed in some other window, it remains displayed there.

If you wish to replace a buffer in all the windows that display it, use `replace-buffer-in-windows`. See Section 31.6 [Buffers and Windows], page 453.

30.9 Creating Buffers

This section describes the two primitives for creating buffers. `get-buffer-create` creates a buffer if it finds no existing buffer with the specified name; `generate-new-buffer` always creates a new buffer and gives it a unique name.

Other functions you can use to create buffers include `with-output-to-temp-buffer` (see Section 45.8 [Temporary Displays], page 656) and `create-file-buffer` (see Section 28.1 [Visiting Files], page 391). Starting a subprocess can also create a buffer (see Chapter 49 [Processes], page 673).

get-buffer-create *name* Function

This function returns a buffer named *name*. It returns an existing buffer with that name, if one exists; otherwise, it creates a new buffer. The buffer does not become the current buffer—this function does not change which buffer is current.

An error is signaled if *name* is not a string.

```
(get-buffer-create "foo")
     ⇒ #<buffer foo>
```

The major mode for the new buffer is set to Fundamental mode. The variable `default-major-mode` is handled at a higher level. See Section 26.1.3 [Auto Major Mode], page 367.

generate-new-buffer *name* Function

This function returns a newly created, empty buffer, but does not make it current. If there is no buffer named *name*, then that is the name of the new buffer. If that name is in use, this function adds suffixes of the form '`<`*n*`>`' to *name*, where *n* is an integer. It tries successive integers starting with 2 until it finds an available name.

An error is signaled if *name* is not a string.

```
(generate-new-buffer "bar")
     ⇒ #<buffer bar>
(generate-new-buffer "bar")
     ⇒ #<buffer bar<2>>
(generate-new-buffer "bar")
     ⇒ #<buffer bar<3>>
```

The major mode for the new buffer is set to Fundamental mode. The variable `default-major-mode` is handled at a higher level. See Section 26.1.3 [Auto Major Mode], page 367.

See the related function `generate-new-buffer-name` in Section 30.3 [Buffer Names], page 433.

30.10 Killing Buffers

Killing a buffer makes its name unknown to XEmacs and makes its text space available for other use.

The buffer object for the buffer that has been killed remains in existence as long as anything refers to it, but it is specially marked so that you cannot make it current or display it. Killed buffers retain their identity, however; two distinct buffers, when killed, remain distinct according to `eq`.

If you kill a buffer that is current or displayed in a window, XEmacs 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 30.2 [Current Buffer], page 431.

If you kill a buffer that is the base buffer of one or more indirect buffers, the indirect buffers are automatically killed as well.

The `buffer-name` of a killed buffer is `nil`. To test whether a buffer has been killed, you can either use this feature or the function `buffer-live-p`.

buffer-live-p *buffer* Function

This function returns `nil` if *buffer* is deleted, and `t` otherwise.

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 basic meaning of `SIGHUP` is that a dialup line has been disconnected.) See Section 49.5 [Deleting Processes], page 678.

If the buffer is visiting a file and contains unsaved changes, `kill-buffer` asks the user to confirm before the buffer is killed. It does this even if not called interactively. To prevent the request for confirmation, clear the modified flag before calling `kill-buffer`. See Section 30.5 [Buffer Modification], page 436.

Killing a buffer that is already dead has no effect.

```
(kill-buffer "foo.unchanged")
     ⇒ nil
(kill-buffer "foo.changed")

---------- Buffer: Minibuffer ----------
Buffer foo.changed modified; kill anyway? (yes or no) yes
---------- Buffer: Minibuffer ----------

     ⇒ nil
```

kill-buffer-query-functions Variable

After confirming unsaved changes, `kill-buffer` calls the functions in the list `kill-buffer-query-functions`, in order of appearance, with no arguments. The buffer being killed is the current buffer when they are called. The idea is that these functions ask for confirmation from the user for various nonstandard reasons. If any of them returns `nil`, `kill-buffer` spares the buffer's life.

kill-buffer-hook Variable

This is a normal hook run by `kill-buffer` after asking all the questions it is going to ask, just before actually killing the buffer. The buffer to be killed is current when the hook functions run. See Section 26.4 [Hooks], page 378.

buffer-offer-save Variable

This variable, if non-`nil` in a particular buffer, tells `save-buffers-kill-emacs` and `save-some-buffers` to offer to save that buffer, just as they offer to save file-visiting buffers. The variable `buffer-offer-save` automatically becomes buffer-local when set for any reason. See Section 10.9 [Buffer-Local Variables], page 157.

30.11 Indirect Buffers

An *indirect buffer* shares the text of some other buffer, which is called the *base buffer* of the indirect buffer. In some ways it is the analogue, for buffers, of a symbolic link among files. The base buffer may not itself be an indirect buffer.

The text of the indirect buffer is always identical to the text of its base buffer; changes made by editing either one are visible immediately in the other. This includes the text properties as well as the characters themselves.

But in all other respects, the indirect buffer and its base buffer are completely separate. They have different names, different values of point, different narrowing, different markers and overlays (though inserting or deleting text in either buffer relocates the markers and overlays for both), different major modes, and different local variables.

An indirect buffer cannot visit a file, but its base buffer can. If you try to save the indirect buffer, that actually works by saving the base buffer.

Killing an indirect buffer has no effect on its base buffer. Killing the base buffer effectively kills the indirect buffer in that it cannot ever again be the current buffer.

make-indirect-buffer *base-buffer name* Command

This creates an indirect buffer named *name* whose base buffer is *base-buffer*. The argument *base-buffer* may be a buffer or a string.

If *base-buffer* is an indirect buffer, its base buffer is used as the base for the new buffer.

buffer-base-buffer *buffer* Function

This function returns the base buffer of *buffer*. If *buffer* is not indirect, the value is `nil`. Otherwise, the value is another buffer, which is never an indirect buffer.

31 Windows

This chapter describes most of the functions and variables related to Emacs windows. See Chapter 45 [Display], page 647, for information on how text is displayed in windows.

31.1 Basic Concepts of Emacs Windows

A *window* in XEmacs is the physical area of the screen in which a buffer is displayed. The term is also used to refer to a Lisp object that represents that screen area in XEmacs Lisp. It should be clear from the context which is meant.

XEmacs groups windows into frames. A frame represents an area of screen available for XEmacs to use. Each frame always contains at least one window, but you can subdivide it vertically or horizontally into multiple nonoverlapping Emacs windows.

In each frame, at any time, one and only one window is designated as *selected within the frame*. The frame's cursor appears in that window. At ant time, one frame is the selected frame; and the window selected within that frame is *the selected window*. The selected window's buffer is usually the current buffer (except when `set-buffer` has been used). See Section 30.2 [Current Buffer], page 431.

For practical purposes, a window exists only while it is displayed in a frame. Once removed from the frame, the window is effectively deleted and should not be used, *even though there may still be references to it* from other Lisp objects. Restoring a saved window configuration is the only way for a window no longer on the screen to come back to life. (See Section 31.3 [Deleting Windows], page 449.)

Each window has the following attributes:

- containing frame
- window height
- window width
- window edges with respect to the frame or screen
- the buffer it displays
- position within the buffer at the upper left of the window
- amount of horizontal scrolling, in columns
- point
- the mark
- how recently the window was selected

Users create multiple windows so they can look at several buffers at once. Lisp libraries use multiple windows for a variety of reasons, but most often to display related information. In Rmail, for example, you can move through a summary buffer in one window while the other window shows messages one at a time as they are reached.

The meaning of "window" in XEmacs is similar to what it means in the context of general-purpose window systems such as X, but not identical. The X Window System places X windows on the screen; XEmacs uses one or more X windows as frames, and subdivides them into Emacs windows. When you use XEmacs on a character-only terminal, XEmacs treats the whole terminal screen as one frame.

Most window systems support arbitrarily located overlapping windows. In contrast, Emacs windows are *tiled*; they never overlap, and together they fill the whole screen or frame. Because of the way in which XEmacs creates new windows and resizes them, you can't create every conceivable tiling of windows on an Emacs frame. See Section 31.2 [Splitting Windows], page 446, and Section 31.13 [Size of Window], page 464.

See Chapter 45 [Display], page 647, 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.

31.2 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 31.7 [Displaying Buffers], page 453).

The functions described here do not accept a buffer as an argument. The two "halves" of the split window initially display the same buffer previously visible in the window that was split.

one-window-p &optional *no-mini all-frames* Function

This function returns non-`nil` if there is only one window. The argument *no-mini*, if non-`nil`, means don't count the minibuffer even if it is active; otherwise, the minibuffer window is included, if active, in the total number of windows which is compared against one.

The argument *all-frame* controls which set of windows are counted.

- If it is `nil` or omitted, then count only the selected frame, plus the minibuffer it uses (which may be on another frame).
- If it is `t`, then windows on all frames that currently exist (including invisible and iconified frames) are counted.
- If it is the symbol `visible`, then windows on all visible frames are counted.
- If it is the number 0, then windows on all visible and iconified frames are counted.
- If it is any other value, then precisely the windows in *window*'s frame are counted, excluding the minibuffer in use if it lies in some other frame.

split-window &optional *window size horizontal* Command

This function splits *window* into two windows. The original window *window* remains the selected window, but occupies only part of its former screen area. The rest is occupied by a newly created window which is returned as the value of this function.

If *horizontal* is non-`nil`, then *window* splits into two side by side windows. The original window *window* keeps the leftmost *size* columns, and gives the rest of the columns to the new window. Otherwise, it splits into windows one above the other, and *window* keeps the upper *size* lines and gives the rest of the lines to the new window. The original window is therefore the left-hand or upper of the two, and the new window is the right-hand or lower.

If *window* is omitted or `nil`, then the selected window is split. If *size* is omitted or `nil`, then *window* is divided evenly into two parts. (If there is an odd line, it is allocated to the new window.) When `split-window` is called interactively, all its arguments are `nil`.

The following example starts with one window on a frame 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 frame 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
```

Normally, Emacs indicates the border between two side-by-side windows with a scroll bar (see Section 32.2.3 [X Frame Properties], page 473) or '`|`' characters. The display table can specify alternative border characters; see Section 45.11 [Display Tables], page 659.

split-window-vertically &optional *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 current window into two windows, one above the other."
  (interactive "P")
  (split-window nil (and arg (prefix-numeric-value arg))))
```

split-window-horizontally &optional *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))
```

one-window-p &optional *no-mini all-frames* — Function

This function returns non-`nil` if there is only one window. The argument *no-mini*, if non-`nil`, means don't count the minibuffer even if it is active; otherwise, the minibuffer window is included, if active, in the total number of windows, which is compared against one.

The argument *all-frames* specifies which frames to consider. Here are the possible values and their meanings:

`nil` Count the windows in the selected frame, plus the minibuffer used by that frame even if it lies in some other frame.

`t` Count all windows in all existing frames.

`visible` Count all windows in all visible frames.

0 Count all windows in all visible or iconified frames.

anything else
Count precisely the windows in the selected frame, and no others.

31.3 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 31.16 [Window Configurations], page 468). 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 the other windows in that frame. 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*, it does nothing.

`delete-windows-on` operates frame by frame. If a frame has several windows showing different buffers, then those showing *buffer* are removed, and the others expand to fill the space. If all windows in some frame are showing *buffer* (including the case where there is only one window), then the frame reverts to having a single window showing another buffer chosen with `other-buffer`. See Section 30.8 [The Buffer List], page 439.

The argument *frame* controls which frames to operate on:

- If it is `nil`, operate on the selected frame.
- If it is `t`, operate on all frames.
- If it is `visible`, operate on all visible frames.
- 0 If it is 0, operate on all visible or iconified frames.
- If it is a frame, operate on that frame.

This function always returns `nil`.

31.4 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 &optional *device* — Function

This function returns the selected window. This is the window in which the cursor appears and to which many commands apply. Each separate device can have its own selected window, which is remembered as focus changes from device to device. Optional argument *device* specifies which device to return the selected window for, and defaults to the selected device.

select-window *window* &optional *norecord* — 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.

If optional argument *norecord* is non-`nil` then the global and per-frame buffer orderings are not modified, as by the function `record-buffer`.

The return value is *window*.

```
(setq w (next-window))
(select-window w)
     ⇒ #<window 65 on windows.texi>
```

save-selected-window *forms...* — Macro

This macro records the selected window, executes *forms* in sequence, then restores the earlier selected window. It does not save or restore anything about the sizes, arrangement or contents of windows; therefore, if the *forms* change them, the changes are permanent.

The following functions choose one of the windows on the screen, offering various criteria for the choice.

get-lru-window &optional *frame* — Function

This function returns the window least recently "used" (that is, selected). The selected window is always the most recently used window.

The selected window can be the least recently used window if it is the only window. A newly created window becomes the least recently used window until it is selected. A minibuffer window is never a candidate.

The argument *frame* controls which windows are considered.

- If it is `nil`, consider windows on the selected frame.
- If it is `t`, consider windows on all frames.
- If it is `visible`, consider windows on all visible frames.
- If it is 0, consider windows on all visible or iconified frames.
- If it is a frame, consider windows on that frame.

get-largest-window &optional *frame* Function

This function returns the window with the largest area (height times width). If there are no side-by-side windows, then this is the window with the most lines. A minibuffer window is never a candidate.

If there are two windows of the same size, then the function returns the window that is first in the cyclic ordering of windows (see following section), starting from the selected window.

The argument *frame* controls which set of windows are considered. See `get-lru-window`, above.

31.5 Cyclic Ordering of Windows

When you use the command *`C-x o`* (`other-window`) to select the next window, it moves through all the windows on the screen in a specific cyclic order. For any given configuration of windows, this order never varies. It is called the *cyclic ordering of windows*.

This ordering generally goes from top to bottom, and from left to right. But it may go down first or go right first, depending on the order in which the windows were split.

If the first split was vertical (into windows one above each other), and then the subwindows were split horizontally, then the ordering is left to right in the top of the frame, and then left to right in the next lower part of the frame, and so on. If the first split was horizontal, the ordering is top to bottom in the left part, and so on. In general, within each set of siblings at any level in the window tree, the order is left to right, or top to bottom.

next-window &optional *window minibuf all-frames* Function

This function returns the window following *window* in the cyclic ordering of windows. This is the window that *`C-x o`* would select if typed when *window* is selected. If *window* is the only window visible, then this function returns *window*. If omitted, *window* defaults to the selected window.

The value of the argument *minibuf* determines whether the minibuffer is included in the window order. Normally, when *minibuf* is `nil`, the minibuffer is included if it is currently active; this is the behavior of *`C-x o`*. (The minibuffer window is active while the minibuffer is in use. See Chapter 18 [Minibuffers], page 261.)

If *minibuf* is `t`, then the cyclic ordering includes the minibuffer window even if it is not active.

If *minibuf* is neither `t` nor `nil`, then the minibuffer window is not included even if it is active.

The argument *all-frames* specifies which frames to consider. Here are the possible values and their meanings:

`nil` Consider all the windows in *window*'s frame, plus the minibuffer used by that frame even if it lies in some other frame.

`t` Consider all windows in all existing frames.

`visible` Consider all windows in all visible frames. (To get useful results, you must ensure *window* is in a visible frame.)

0 Consider all windows in all visible or iconified frames.

anything else
Consider precisely the windows in *window*'s frame, and no others.

This example assumes there are two windows, both displaying the buffer '`windows.texi`':

```
(selected-window)
     ⇒ #<window 56 on windows.texi>
(next-window (selected-window))
     ⇒ #<window 52 on windows.texi>
(next-window (next-window (selected-window)))
     ⇒ #<window 56 on windows.texi>
```

previous-window &optional *window minibuf all-frames* Function

This function returns the window preceding *window* in the cyclic ordering of windows. The other arguments specify which windows to include in the cycle, as in `next-window`.

other-window *count* &optional *frame* Command

This function selects the *count*th following 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.

The argument *frame* controls which set of windows are considered.

- If it is `nil` or omitted, 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 frames that currently exist (including invisible and iconified frames) are considered.
- If it is the symbol `visible`, then windows on all visible frames are considered.
- If it is the number 0, then windows on all visible and iconified frames are considered.
- If it is any other value, then the behavior is undefined.

walk-windows *proc* &optional *minibuf all-frames* Function

This function cycles through all windows, calling `proc` once for each window with the window as its sole argument.

The optional arguments *minibuf* and *all-frames* specify the set of windows to include in the scan. See `next-window`, above, for details.

31.6 Buffers and Windows

This section describes low-level functions to examine windows or to display buffers in windows in a precisely controlled fashion. See the following section for related functions that find a window to use and specify a buffer for it. The functions described there are easier to use than these, but they employ heuristics in choosing or creating a window; use these functions when you need complete control.

set-window-buffer *window buffer-or-name* Function

This function makes *window* display *buffer-or-name* as its contents. It returns `nil`.

```
(set-window-buffer (selected-window) "foo")
     ⇒ nil
```

window-buffer &optional *window* Function

This function returns the buffer that *window* is displaying. If *window* is omitted, this function returns the buffer for the selected window.

```
(window-buffer)
     ⇒ #<buffer windows.texi>
```

get-buffer-window *buffer-or-name* &optional *frame* 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 31.5 [Cyclic Window Ordering], page 451.

The argument *all-frames* controls which windows to consider.

- If it is `nil`, consider windows on the selected frame.
- If it is `t`, consider windows on all frames.
- If it is `visible`, consider windows on all visible frames.
- If it is 0, consider windows on all visible or iconified frames.
- If it is a frame, consider windows on that frame.

31.7 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 30.2 [Current Buffer], page 431) and `save-excursion` (see Section 34.3 [Excursions], page 495), 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 30.2 [Current Buffer], page 431.

If *buffer-or-name* does not identify an existing buffer, then a new buffer by that name is created. The major mode for the new buffer is set according to the variable `default-major-mode`. See Section 26.1.3 [Auto Major Mode], page 367.

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 30.8 [The Buffer List], page 439.

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 currently selected window is absolutely never used to do the job. If it is the only window, then it is split to make a distinct window for this purpose. If the selected window is already displaying the buffer, then it continues to do so, but another window is nonetheless found to display it in as well.

pop-to-buffer *buffer-or-name* &optional *other-window on-frame* Function

This function makes *buffer-or-name* the current buffer and switches to it in some window, preferably not the window previously selected. The "popped-to" window becomes the selected window within its frame.

If the variable `pop-up-frames` is non-`nil`, `pop-to-buffer` looks for a window in any visible frame already displaying the buffer; if there is one, it returns that window and makes it be selected within its frame. If there is none, it creates a new frame and displays the buffer in it.

If `pop-up-frames` is `nil`, then `pop-to-buffer` operates entirely within the selected frame. (If the selected frame has just a minibuffer, `pop-to-buffer` operates within the most recently selected frame that was not just a minibuffer.)

If the variable `pop-up-windows` is non-`nil`, windows may be split to create a new window that is different from the original window. For details, see Section 31.8 [Choosing Window], page 455.

If *other-window* is non-`nil`, `pop-to-buffer` finds or creates another window even if *buffer-or-name* is already visible in the selected window. Thus *buffer-or-name* could end up displayed in two windows. On the other hand, if *buffer-or-name* is already displayed in the selected window and *other-window* is `nil`, then the selected window is considered sufficient display for *buffer-or-name*, so that nothing needs to be done.

All the variables that affect `display-buffer` affect `pop-to-buffer` as well. See Section 31.8 [Choosing Window], page 455.

If *buffer-or-name* is a string that does not name an existing buffer, a buffer by that name is created. The major mode for the new buffer is set according to the variable `default-major-mode`. See Section 26.1.3 [Auto Major Mode], page 367.

If *on-frame* is non-`nil`, it is the frame to pop to this buffer on.

An example use of this function is found at the end of Section 49.9.2 [Filter Functions], page 684.

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`.

31.8 Choosing a Window for Display

This section describes the basic facility that chooses a window to display a buffer in—`display-buffer`. All the higher-level functions and commands use this subroutine. Here we describe how to use `display-buffer` and how to customize it.

display-buffer *buffer-or-name* &optional *not-this-window* Command

This command makes *buffer-or-name* appear in some window, like `pop-to-buffer`, but it does not select that window and does not make the buffer current. The identity of the selected window is unaltered by this function.

If *not-this-window* is non-`nil`, it means to display the specified buffer in a window other than the selected one, even if it is already on display in the selected window. This can cause the buffer to appear in two windows at once. Otherwise, if *buffer-or-name* is already being displayed in any window, that is good enough, so this function does nothing.

`display-buffer` returns the window chosen to display *buffer-or-name*.

Precisely how `display-buffer` finds or creates a window depends on the variables described below.

A window can be marked as "dedicated" to a particular buffer. Then XEmacs will not automatically change which buffer appears in the window, such as `display-buffer` might normally do.

window-dedicated-p *window* Function

This function returns *window*'s dedicated object, usually `t` or `nil`.

set-window-buffer-dedicated *window buffer* Function

This function makes *window* display *buffer* and be dedicated to that buffer. Then XEmacs will not automatically change which buffer appears in *window*. If *buffer* is `nil`, this function makes *window* not be dedicated (but doesn't change which buffer appears in it currently).

pop-up-windows User Option

This variable controls whether `display-buffer` makes new windows. If it is non-`nil` and there is only one window, then that window is split. If it is `nil`, then `display-buffer` does not split the single window, but uses it whole.

split-height-threshold User Option

This variable determines when `display-buffer` may split a window, if there are multiple windows. `display-buffer` always splits the largest window if it has at least this many lines. If the largest window is not this tall, it is split only if it is the sole window and `pop-up-windows` is non-`nil`.

pop-up-frames User Option

This variable controls whether `display-buffer` makes new frames. If it is non-`nil`, `display-buffer` looks for an existing window already displaying the desired buffer, on any visible frame. If it finds one, it returns that window. Otherwise it makes a new frame. The variables `pop-up-windows` and `split-height-threshold` do not matter if `pop-up-frames` is non-`nil`.

If `pop-up-frames` is `nil`, then `display-buffer` either splits a window or reuses one.

See Chapter 32 [Frames], page 471, for more information.

pop-up-frame-function Variable

This variable specifies how to make a new frame if `pop-up-frames` is non-`nil`.

Its value should be a function of no arguments. When `display-buffer` makes a new frame, it does so by calling that function, which should return a frame. The default value of the variable is a function that creates a frame using properties from `pop-up-frame-plist`.

pop-up-frame-plist Variable

This variable holds a plist specifying frame properties used when `display-buffer` makes a new frame. See Section 32.2 [Frame Properties], page 471, for more information about frame properties.

special-display-buffer-names Variable

A list of buffer names for buffers that should be displayed specially. If the buffer's name is in this list, `display-buffer` handles the buffer specially.

By default, special display means to give the buffer a dedicated frame.

If an element is a list, instead of a string, then the CAR of the list is the buffer name, and the rest of the list says how to create the frame. There are two possibilities for the rest of the list. It can be a plist, specifying frame properties, or it can contain a function and arguments to give to it. (The function's first argument is always the buffer to be displayed; the arguments from the list come after that.)

special-display-regexps Variable

A list of regular expressions that specify buffers that should be displayed specially. If the buffer's name matches any of the regular expressions in this list, `display-buffer` handles the buffer specially.

By default, special display means to give the buffer a dedicated frame.

If an element is a list, instead of a string, then the CAR of the list is the regular expression, and the rest of the list says how to create the frame. See above, under `special-display-buffer-names`.

special-display-function Variable

This variable holds the function to call to display a buffer specially. It receives the buffer as an argument, and should return the window in which it is displayed.

The default value of this variable is `special-display-popup-frame`.

special-display-popup-frame *buffer* Function

This function makes *buffer* visible in a frame of its own. If *buffer* is already displayed in a window in some frame, it makes the frame visible and raises it, to use that window. Otherwise, it creates a frame that will be dedicated to *buffer*.

This function uses an existing window displaying *buffer* whether or not it is in a frame of its own; but if you set up the above variables in your init file, before *buffer* was created, then presumably the window was previously made by this function.

special-display-frame-plist User Option

This variable holds frame properties for `special-display-popup-frame` to use when it creates a frame.

same-window-buffer-names Variable

A list of buffer names for buffers that should be displayed in the selected window. If the buffer's name is in this list, `display-buffer` handles the buffer by switching to it in the selected window.

same-window-regexps Variable

A list of regular expressions that specify buffers that should be displayed in the selected window. If the buffer's name matches any of the regular expressions in this list, `display-buffer` handles the buffer by switching to it in the selected window.

display-buffer-function Variable

This variable is the most flexible way to customize the behavior of `display-buffer`. If it is non-`nil`, it should be a function that `display-buffer` calls to do the work. The function should accept two arguments, the same two arguments that `display-buffer` received. It should choose or create a window, display the specified buffer, and then return the window.

This hook takes precedence over all the other options and hooks described above.

A window can be marked as "dedicated" to its buffer. Then `display-buffer` 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 *flag* is non-`nil`, and nondedicated otherwise.

31.9 Windows and Point

Each window has its own value of point, independent of the value of point in other windows displaying the same buffer. This makes it useful to have multiple windows showing one buffer.

- The window point is established when a window is first created; it is initialized from the buffer's point, or from the window point of another window opened on the buffer if such a window exists.
- Selecting a window sets the value of point in its buffer 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 34 [Positions], page 487, for more details on buffer positions.

As far as the user is concerned, point is where the cursor is, and when the user switches to another buffer, the cursor jumps to the position of point in that buffer.

window-point *window* Function

This function returns the current position of point in *window*. For a nonselected window, this is the value point would have (in that window's buffer) if that window were selected.

When *window* is the selected window and its buffer is also the current buffer, the value returned is the same as point in that buffer.

Strictly speaking, it would be more correct to return the "top-level" value of point, outside of any `save-excursion` forms. But that value is hard to find.

set-window-point *window position* Function

This function positions point in *window* at position *position* in *window*'s buffer.

31.10 The Window Start Position

Each window contains a marker used to keep track of a buffer position that specifies where in the buffer display should start. This position is called the *display-start* position of the window (or just the *start*). The character after this position is the one that appears at the upper left corner of the window. It is usually, but not inevitably, at the beginning of a text line.

window-start &optional *window* Function

This function returns the display-start position of window *window*. If *window* is `nil`, the selected window is used. For example,

```
(window-start)
     ⇒ 7058
```

When you create a window, or display a different buffer in it, the display-start position is set to a display-start position recently used for the same buffer, or 1 if the buffer doesn't have any.

For a realistic example, see the description of `count-lines` in Section 34.2.4 [Text Lines], page 490.

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.

Simply changing the buffer text or moving point does not update the value that `window-end` returns. The value is updated only when Emacs redisplays and redisplay actually finishes.

If the last redisplay of *window* was preempted, and did not finish, Emacs does not know the position of the end of display in that window. In that case, this function returns a value that is not correct. In a future version, `window-end` will return `nil` in that case.

set-window-start *window position* &optional *noforce* — Function

This function sets the display-start position of *window* to *position* in *window*'s buffer. It returns *position*.

The display routines insist that the position of point be visible when a buffer is displayed. Normally, they change the display-start position (that is, scroll the window) whenever necessary to make point visible. However, if you specify the start position with this function using `nil` for *noforce*, it means you want display to start at *position* even if that would put the location of point off the screen. If this does place point off screen, the display routines move point to the left margin on the middle line in the window.

For example, if point is 1 and you set the start of the window to 2, then point would be "above" the top of the window. The display routines will automatically move point if it is still 1 when redisplay occurs. Here is an example:

```
;; Here is what ‘foo’ looks like before executing
;;   the set-window-start expression.

---------- Buffer: foo ----------
⋆This is the contents of buffer foo.
2
3
4
5
6
---------- Buffer: foo ----------

(set-window-start
 (selected-window)
 (1+ (window-start)))
⇒ 2
```

```
;; Here is what 'foo' looks like after executing
;;   the set-window-start expression.
---------- Buffer: foo ----------
his is the contents of buffer foo.
2
3
⋆4
5
6
---------- Buffer: foo ----------
```

If *noforce* is non-`nil`, and *position* would place point off screen at the next redisplay, then redisplay computes a new window-start position that works well with point, and thus *position* is not used.

pos-visible-in-window-p &optional *position window* Function

This function returns `t` if *position* is within the range of text currently visible on the screen in *window*. It returns `nil` if *position* is scrolled vertically out of view. The argument *position* defaults to the current position of point; *window*, to the selected window. Here is an example:

```
(or (pos-visible-in-window-p
     (point) (selected-window))
    (recenter 0))
```

The `pos-visible-in-window-p` function considers only vertical scrolling. If *position* is out of view only because *window* has been scrolled horizontally, `pos-visible-in-window-p` returns `t`. See Section 31.12 [Horizontal Scrolling], page 462.

31.11 Vertical Scrolling

Vertical scrolling means moving the text up or down in a window. It works by changing the value of the window's display-start location. It may also change the value of `window-point` to keep it on the screen.

In the commands `scroll-up` and `scroll-down`, the directions "up" and "down" refer to the motion of the text in the buffer at which you are looking through the window. Imagine that the text is written on a long roll of paper and that the scrolling commands move the paper up and down. Thus, if you are looking at text in the middle of a buffer and repeatedly call `scroll-down`, you will eventually see the beginning of the buffer.

Some people have urged that the opposite convention be used: they imagine that the window moves over text that remains in place. Then "down" commands would take you to the end of the buffer. This view is more consistent with the actual relationship between windows and the text in the buffer, but it is less like what the user sees. The position of a window on the terminal does not move, and short scrolling commands clearly move the text up or down on the screen. We have chosen names that fit the user's point of view.

The scrolling functions (aside from `scroll-other-window`) have unpredictable results if the current buffer is different from the buffer that is displayed in the selected window. See Section 30.2 [Current Buffer], page 431.

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 modeline).

`scroll-up` returns `nil`.

scroll-down &optional *count* Command

This function scrolls the text in the selected window downward *count* lines. If *count* is negative, scrolling is actually upward.

If *count* is omitted or `nil`, then the length of the scroll is `next-screen-context-lines` lines less than the usable height of the window (not counting its mode line).

`scroll-down` returns `nil`.

scroll-other-window &optional *count* Command

This function scrolls the text in another window upward *count* lines. Negative values of *count*, or `nil`, are handled as in `scroll-up`.

You can specify a buffer to scroll with the variable `other-window-scroll-buffer`. When the selected window is the minibuffer, the next window is normally the one at the top left corner. You can specify a different window to scroll with the variable `minibuffer-scroll-window`. This variable has no effect when any other window is selected. See Section 18.8 [Minibuffer Misc], page 278.

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 redisplay scrolls the text to center point vertically in the window. If the value is a positive integer n, then redisplay brings point back on screen by scrolling n lines in either direction, if possible; otherwise, it centers point. The default value is zero.

scroll-conservatively User Option

This variable controls how many lines Emacs tries to scroll before recentering. If you set it to a small number, then when you move point a short distance off the screen, XEmacs will scroll the screen just far enough to bring point back on screen, provided that does not exceed `scroll-conservatively` lines. This variable overrides the redisplay preemption.

next-screen-context-lines User Option

The value of this variable is the number of lines of continuity to retain when scrolling by full screens. For example, `scroll-up` with an argument of `nil` scrolls so that this many lines at the bottom of the window appear instead at the top. The default value is `2`.

recenter &optional *count* Command

This function scrolls the selected window to put the text where point is located at a specified vertical position within the window.

If *count* is a nonnegative number, it puts the line containing point *count* lines down from the top of the window. If *count* is a negative number, then it counts upward from the bottom of the window, so that −1 stands for the last usable line in the window. If *count* is a non-`nil` list, then it stands for the line in the middle of the window.

If *count* is `nil`, `recenter` puts the line containing point in the middle of the window, then clears and redisplays the entire selected frame.

When `recenter` is called interactively, *count* is the raw prefix argument. Thus, typing *C-u* as the prefix sets the *count* to a non-`nil` list, while typing *C-u 4* sets *count* to 4, which positions the current line four lines from the top.

With an argument of zero, `recenter` positions the current line at the top of the window. This action is so handy that some people make a separate key binding to do this. For example,

```
(defun line-to-top-of-window ()
  "Scroll current line to top of window.
Replaces three keystroke sequence C-u 0 C-l."
  (interactive)
  (recenter 0))

(global-set-key [kp-multiply] 'line-to-top-of-window)
```

31.12 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 this is not allowed. Scrolling to the left is allowed; it scrolls the first columns of text off the edge of the window and can reveal additional columns on the right that were truncated before. Once a window has a nonzero amount of leftward horizontal scrolling, you can scroll it back to the right, but only so far as to reduce the net horizontal scroll to zero. There is no limit to how far left you can scroll, but eventually all the text will disappear off the left edge.

scroll-left *count* Command

This function scrolls the selected window *count* columns to the left (or to the right if *count* is negative). The return value is the total amount of leftward horizontal scrolling in effect after the change—just like the value returned by `window-hscroll` (below).

scroll-right *count* Command

This function scrolls the selected window *count* columns to the right (or to the left if *count* is negative). The return value is the total amount of leftward horizontal scrolling in effect after the change—just like the value returned by `window-hscroll` (below).

Once you scroll a window as far right as it can go, back to its normal position where the total leftward scrolling is zero, attempts to scroll any farther right have no effect.

window-hscroll &optional *window* Function

This function returns the total leftward horizontal scrolling of *window*—the number of columns by which the text in *window* is scrolled left past the left margin.

The value is never negative. It is zero when no horizontal scrolling has been done in *window* (which is usually the case).

If *window* is `nil`, the selected window is used.

```
(window-hscroll)
     ⇒ 0
(scroll-left 5)
     ⇒ 5
(window-hscroll)
     ⇒ 5
```

set-window-hscroll *window columns* Function

This function sets the number of columns from the left margin that *window* is scrolled 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:

```
(defun hscroll-on-screen (window position)
  (save-excursion
    (goto-char position)
    (and
     (>= (- (current-column) (window-hscroll window)) 0)
     (< (- (current-column) (window-hscroll window))
        (window-width window)))))
```

31.13 The Size of a Window

An Emacs window is rectangular, and its size information consists of the height (in lines or pixels) and the width (in character positions or pixels). The modeline is included in the height. The pixel width and height values include scrollbars and margins, while the line/character-position values do not.

Note that the height in lines, and the width in characters, are determined by dividing the corresponding pixel value by the height or width of the default font in that window (if this is a variable-width font, the average width is used). The resulting values may or may not represent the actual number of lines in the window, or the actual number of character positions in any particular line, esp. if there are pixmaps or various different fonts in the window.

The following functions return size information about a window:

window-height &optional *window* Function

This function returns the number of lines in *window*, including its modeline but not including the horizontal scrollbar, if any (this is different from `window-pixel-height`). If *window* is `nil`, the function uses the selected window.

```
(window-height)
     ⇒ 40
(split-window-vertically)
     ⇒ #<window on "windows.texi" 0x679b>
(window-height)
     ⇒ 20
```

window-width &optional *window* Function

This function returns the number of columns in *window*, not including any left margin, right margin, or vertical scrollbar (this is different from `window-pixel-width`). If *window* is `nil`, the function uses the selected window.

```
(window-width)
     ⇒ 80
(window-height)
     ⇒ 40
(split-window-horizontally)
     ⇒ #<window on "windows.texi" 0x7538>
(window-width)
     ⇒ 39
```

Note that after splitting the window into two side-by-side windows, the width of each window is less the half the width of the original window because a vertical scrollbar appeared between the windows, occupying two columns worth of space. Also, the height shrunk by one because horizontal scrollbars appeared that weren't there before. (Horizontal scrollbars appear only when lines are truncated, not when they wrap. This is usually the case for horizontally split windows but not for full-frame windows. You can change this using the variables `truncate-lines` and `truncate-partial-width-windows`.)

window-pixel-height &optional *window* Function

This function returns the height of *window* in pixels, including its modeline and horizontal scrollbar, if any. If *window* is `nil`, the function uses the selected window.

```
(window-pixel-height)
     ⇒ 600
(split-window-vertically)
     ⇒ #<window on "windows.texi" 0x68a6>
(window-pixel-height)
     ⇒ 300
```

window-pixel-width &optional *window* Function

This function returns the width of *window* in pixels, including any left margin, right margin, or vertical scrollbar that may be displayed alongside it. If *window* is `nil`, the function uses the selected window.

```
(window-pixel-width)
     ⇒ 735
(window-pixel-height)
     ⇒ 600
(split-window-horizontally)
     ⇒ #<window on "windows.texi" 0x7538>
(window-pixel-width)
     ⇒ 367
(window-pixel-height)
     ⇒ 600
```

window-text-area-pixel-height &optional *window* Function

This function returns the height in pixels of the text displaying portion of *window*, which defaults to the selected window. Unlike `window-pixel-height`, the space occupied by the modeline and horizontal scrollbar, if any, is not counted.

window-text-area-pixel-width &optional *window* Function

This function returns the width in pixels of the text displaying portion of *window*, which defaults to the selected window. Unlike `window-pixel-width`, the space occupied by the vertical scrollbar and divider, if any, is not counted.

window-displayed-text-pixel-height &optional *window* Function
noclipped

This function returns the height in pixels of the text displayed in *window*, which defaults to the selected window. Unlike `window-text-area-pixel-height`, any blank space below the end of the buffer is not included. If optional argument *noclipped* is non-`nil`, any space occupied by clipped lines will not be included.

31.14 The Position of a Window

XEmacs provides functions to determine the absolute location of windows within a frame, and the relative location of a window in comparison to other windows in the same frame.

window-pixel-edges &optional *window* Function

This function returns a list of the pixel 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 pixel used by *window* (including any left margin, right margin, or vertical scrollbar displayed alongside it), and *bottom* is one more than the bottommost pixel used by *window* (including any modeline or horizontal scrollbar displayed above or below it). The frame area does not include any frame menubars or toolbars that may be displayed; thus, for example, if there is only one window on the frame, the values for *left* and *top* will always be 0.

If *window* is at the upper left corner of its frame, *right* and *bottom* are the same as the values returned by `(window-pixel-width)` and `(window-pixel-height)` respectively, and *top* and *bottom* are zero.

There is no longer a function `window-edges` because it does not make sense in a world with variable-width and variable-height lines, as are allowed in XEmacs.

window-highest-p *window* Function

This function returns non-`nil` if *window* is along the top of its frame.

window-lowest-p *window* Function

This function returns non-`nil` if *window* is along the bottom of its frame.

window-text-area-pixel-edges &optional *window* Function

This function allows one to determine the location of the text-displaying portion of *window*, which defaults to the selected window, with respect to the top left corner of the window. It returns a list of integer pixel positions `(left top right bottom)`, all relative to `(0,0)` at the top left corner of the window.

31.15 Changing the Size of a Window

The window size functions fall into two classes: high-level commands that change the size of windows and low-level functions that access window size. XEmacs does not permit overlapping windows or gaps between windows, so resizing one window affects other windows.

enlarge-window *size* &optional *horizontal window* Command

This function makes the selected window *size* lines taller, stealing lines from neighboring windows. It takes the lines from one window at a time until that window is used up, then takes from another. If a window from which lines are stolen shrinks below `window-min-height` lines, that window disappears.

If *horizontal* is non-`nil`, this function makes *window* wider by *size* columns, stealing columns instead of lines. If a window from which columns are stolen shrinks below `window-min-width` columns, that window disappears.

If the requested size would exceed that of the window's frame, then the function makes the window occupy the entire height (or width) of the frame.

If *size* is negative, this function shrinks the window by $-$*size* lines or columns. If that makes the window smaller than the minimum size (`window-min-height` and `window-min-width`), `enlarge-window` deletes the window.

If *window* is non-`nil`, it specifies a window to change instead of the selected window.

`enlarge-window` returns `nil`.

enlarge-window-horizontally *columns* — Command

This function makes the selected window *columns* wider. It could be defined as follows:

```
(defun enlarge-window-horizontally (columns)
  (enlarge-window columns t))
```

enlarge-window-pixels *count* &optional *side window* — Command

This function makes the selected window *count* pixels larger. When called from Lisp, optional second argument *side* non-`nil` means to grow sideways *count* pixels, and optional third argument *window* specifies the window to change instead of the selected window.

shrink-window *size* &optional *horizontal window* — Command

This function is like `enlarge-window` but negates the argument *size*, making the selected window smaller by giving lines (or columns) to the other windows. If the window shrinks below `window-min-height` or `window-min-width`, then it disappears.

If *size* is negative, the window is enlarged by $-$*size* lines or columns.

If *window* is non-`nil`, it specifies a window to change instead of the selected window.

shrink-window-horizontally *columns* — Command

This function makes the selected window *columns* narrower. It could be defined as follows:

```
(defun shrink-window-horizontally (columns)
  (shrink-window columns t))
```

shrink-window-pixels *count* &optional *side window* — Command

This function makes the selected window *count* pixels smaller. When called from Lisp, optional second argument *side* non-`nil` means to shrink sideways *count* pixels, and optional third argument *window* specifies the window to change instead of the selected window.

The following two variables constrain the window-size-changing functions to a minimum height and width.

window-min-height — User Option

The value of this variable determines how short a window may become before it is automatically deleted. Making a window smaller than `window-min-height` automatically deletes it, and no window may be created shorter than this. The absolute minimum height is two (allowing one line for the mode line, and one line for the buffer display). Actions that change window sizes reset this variable to two if it is less than two. The default value is 4.

window-min-width User Option

The value of this variable determines how narrow a window may become before it automatically deleted. Making a window smaller than `window-min-width` automatically deletes it, and no window may be created narrower than this. The absolute minimum width is one; any value below that is ignored. The default value is 10.

window-size-change-functions Variable

This variable holds a list of functions to be called if the size of any window changes for any reason. The functions are called just once per redisplay, and just once for each frame on which size changes have occurred.

Each function receives the frame as its sole argument. There is no direct way to find out which windows changed size, or precisely how; however, if your size-change function keeps track, after each change, of the windows that interest you, you can figure out what has changed by comparing the old size data with the new.

Creating or deleting windows counts as a size change, and therefore causes these functions to be called. Changing the frame size also counts, because it changes the sizes of the existing windows.

It is not a good idea to use `save-window-excursion` in these functions, because that always counts as a size change, and it would cause these functions to be called over and over. In most cases, `save-selected-window` is what you need here.

31.16 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 32.11 [Frame Configurations], page 481.

current-window-configuration Function

This function returns a new object representing XEmacs'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 XEmacs'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`.

This function always counts as a window size change and triggers execution of the `window-size-change-functions`. (It doesn't know how to tell whether the new configuration actually differs from the old one.)

Here is a way of using this function to get the same effect as `save-window-excursion`:

```
(let ((config (current-window-configuration)))
  (unwind-protect
      (progn (split-window-vertically nil)
             ...)
    (set-window-configuration config)))
```

save-window-excursion *forms...* Special Form

This special form records the window configuration, executes *forms* in sequence, then restores the earlier window configuration. The window configuration includes the value of point and the portion of the buffer that is visible. It also includes the choice of selected window. However, it does not include the value of point in the current buffer; use `save-excursion` if you wish to preserve that.

Don't use this construct when `save-selected-window` is all you need.

Exit from `save-window-excursion` always triggers execution of the `window-size-change-functions`. (It doesn't know how to tell whether the restored configuration actually differs from the one in effect at the end of the *forms*.)

The return value is the value of the final form in *forms*. For example:

```
(split-window)
     ⇒ #<window 25 on control.texi>
(setq w (selected-window))
     ⇒ #<window 19 on control.texi>
(save-window-excursion
  (delete-other-windows w)
  (switch-to-buffer "foo")
  'do-something)
     ⇒ do-something
     ;; The frame 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.

32 Frames

A *frame* is a rectangle on the screen that contains one or more XEmacs 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 XEmacs runs on a text-only terminal, it starts with one *TTY frame*. If you create additional ones, XEmacs displays one and only one at any given time—on the terminal screen, of course.

When XEmacs communicates directly with an X server, it does not have a TTY frame; instead, it starts with a single *X window frame*. It can display multiple X window frames at the same time, each in its own X window.

framep *object* Function

This predicate returns `t` if *object* is a frame, and `nil` otherwise.

See Chapter 45 [Display], page 647, for related information.

32.1 Creating Frames

To create a new frame, call the function `make-frame`.

make-frame &optional *props device* Function

This function creates a new frame on *device*, if *device* permits creation of frames. (An X server does; an ordinary terminal does not (yet).) *device* defaults to the selected device if omitted. See Chapter 33 [Consoles and Devices], page 483.

The argument *props* is a property list (a list of alternating keyword-value specifications) of properties for the new frame. (An alist is accepted for backward compatibility but should not be passed in.) Any properties not mentioned in *props* default according to the value of the variable `default-frame-plist`. For X devices, properties not specified in `default-frame-plist` default in turn from `default-x-frame-plist` and, if not specified there, from the X resources. For TTY devices, `default-tty-frame-plist` is consulted as well as `default-frame-plist`.

The set of possible properties depends in principle on what kind of window system XEmacs uses to display its frames. See Section 32.2.3 [X Frame Properties], page 473, for documentation of individual properties you can specify when creating an X window frame.

32.2 Frame Properties

A frame has many properties that control its appearance and behavior. Just what properties a frame has depends on which display mechanism it uses.

Frame properties exist for the sake of window systems. A terminal frame has few properties, mostly for compatibility's sake; only the height, width and `buffer-predicate` properties really do something.

32.2.1 Access to Frame Properties

These functions let you read and change the properties of a frame.

frame-properties &optional *frame* Function

This function returns a plist listing all the properties of *frame* and their values.

frame-property *frame property* &optional *default* Function

This function returns *frame*'s value for the property *property*.

set-frame-properties *frame plist* Function

This function alters the properties of frame *frame* based on the elements of property list *plist*. If you don't mention a property in *plist*, its value doesn't change.

set-frame-property *frame prop val* Function

This function sets the property *prop* of frame *frame* to the value *val*.

32.2.2 Initial Frame Properties

You can specify the properties for the initial startup frame by setting `initial-frame-plist` in your '`.emacs`' file.

initial-frame-plist Variable

This variable's value is a plist of alternating property-value pairs used when creating the initial X window frame.

XEmacs creates the initial frame before it reads your '`~/.emacs`' file. After reading that file, XEmacs checks `initial-frame-plist`, and applies the property settings in the altered value to the already created initial frame.

If these settings affect the frame geometry and appearance, you'll see the frame appear with the wrong ones and then change to the specified ones. If that bothers you, you can specify the same geometry and appearance with X resources; those do take affect before the frame is created. See section "X Resources" in *The XEmacs User's Manual*.

X resource settings typically apply to all frames. If you want to specify some X resources solely for the sake of the initial frame, and you don't want them to apply to subsequent frames, here's how to achieve this: specify properties in `default-frame-plist` to override the X resources for subsequent frames; then, to prevent these from affecting the initial frame, specify the same properties in `initial-frame-plist` with values that match the X resources.

If these properties specify a separate minibuffer-only frame via a `minibuffer` property of `nil`, and you have not yet created one, XEmacs creates one for you.

minibuffer-frame-plist Variable

This variable's value is a plist of properties used when creating an initial minibuffer-only frame—if such a frame is needed, according to the properties for the main initial frame.

default-frame-plist Variable

This is a plist specifying default values of frame properties for subsequent XEmacs frames (not the initial ones).

See also `special-display-frame-plist`, in Section 31.8 [Choosing Window], page 455.

If you use options that specify window appearance when you invoke XEmacs, they take effect by adding elements to `default-frame-plist`. One exception is '`-geometry`', which adds the specified position to `initial-frame-plist` instead. See section "Command Arguments" in *The XEmacs User's Manual*.

32.2.3 X Window Frame Properties

Just what properties a frame has depends on what display mechanism it uses. Here is a table of the properties of an X window frame; of these, `name`, `height`, `width`, and `buffer-predicate` provide meaningful information in non-X frames.

`name` The name of the frame. Most window managers display the frame's name in the frame's border, at the top of the frame. If you don't specify a name, and you have more than one frame, XEmacs sets the frame name based on the buffer displayed in the frame's selected window.

If you specify the frame name explicitly when you create the frame, the name is also used (instead of the name of the XEmacs executable) when looking up X resources for the frame.

`display` The display on which to open this frame. It should be a string of the form `"`*host*`:`*dpy*`.`*screen*`"`, just like the `DISPLAY` environment variable.

`left` The screen position of the left edge, in pixels, with respect to the left edge of the screen. The value may be a positive number *pos*, or a list of the form `(+` *pos*`)` which permits specifying a negative *pos* value.

A negative number $-pos$, or a list of the form `(-` *pos*`)`, actually specifies the position of the right edge of the window with respect to the right edge of the screen. A positive value of *pos* counts toward the left. If the property is a negative integer $-pos$ then *pos* is positive!

`top` The screen position of the top edge, in pixels, with respect to the top edge of the screen. The value may be a positive number *pos*, or a list of the form `(+` *pos*`)` which permits specifying a negative *pos* value.

A negative number $-pos$, or a list of the form `(-` *pos*`)`, actually specifies the position of the bottom edge of the window with respect to the bottom edge of the screen. A positive value of *pos* counts toward the top. If the property is a negative integer $-pos$ then *pos* is positive!

`icon-left`

The screen position of the left edge *of the frame's icon*, in pixels, counting from the left edge of the screen. This takes effect if and when the frame is iconified.

`icon-top` The screen position of the top edge *of the frame's icon*, in pixels, counting from the top edge of the screen. This takes effect if and when the frame is iconified.

`user-position`
: Non-`nil` if the screen position of the frame was explicitly requested by the user (for example, with the '`-geometry`' option). Nothing automatically makes this property non-`nil`; it is up to Lisp programs that call `make-frame` to specify this property as well as specifying the `left` and `top` properties.

`height`
: The height of the frame contents, in characters. (To get the height in pixels, call `frame-pixel-height`; see Section 32.2.4 [Size and Position], page 475.)

`width`
: The width of the frame contents, in characters. (To get the height in pixels, call `frame-pixel-width`; see Section 32.2.4 [Size and Position], page 475.)

`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. If the value is a minibuffer window (in some other frame), the new frame uses that minibuffer. (Minibuffer-only and minibuffer-less frames are not yet implemented in XEmacs.)

`buffer-predicate`
: The buffer-predicate function for this frame. The function `other-buffer` uses this predicate (from the selected frame) to decide which buffers it should consider, if the predicate is not `nil`. It calls the predicate with one arg, a buffer, once for each buffer; if the predicate returns a non-`nil` value, it considers that buffer.

`scroll-bar-width`
: The width of the vertical scroll bar, in pixels.

`cursor-color`
: The color for the cursor that shows point.

`border-color`
: The color for the border of the frame.

`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.

`inter-line-space`
: The space in pixels between adjacent lines of text. (Not currently implemented.)

`modeline`
: Whether the frame has a modeline.

32.2.4 Frame Size And Position

You can read or change the size and position of a frame using the frame properties `left`, `top`, `height`, and `width`. Whatever geometry properties you don't specify are chosen by the window manager in its usual fashion.

Here are some special features for working with sizes and positions:

set-frame-position *frame left top* Function

This function sets the position of the top left corner of *frame* to *left* and *top*. These arguments are measured in pixels, and count from the top left corner of the screen. Negative property values count up or rightward 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 lines and columns. 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.

set-frame-size *frame cols rows* &optional *pretend* Function

This function sets the size of *frame*, measured in characters; *cols* and *rows* specify the new width and height. (If *pretend* is non-nil, it means that redisplay should act as if the frame's size is *cols* by *rows*, but the actual size of the frame should not be changed. You should not normally use this option.)

You can also use the functions `set-frame-height` and `set-frame-width` to set the height and width individually. The frame is the first argument and the size (in rows or columns) is the second. (There is an optional third argument, *pretend*, which has the same purpose as the corresponding argument in `set-frame-size`.)

32.2.5 The Name of a Frame (As Opposed to Its Title)

Under X, every frame has a name, which is not the same as the title of the frame. A frame's name is used to look up its resources and does not normally change over the lifetime of a frame. It is perfectly allowable, and quite common, for multiple frames to have the same name.

frame-name &optional *frame* Function

This function returns the name of *frame*, which defaults to the selected frame if not specified. The name of a frame can also be obtained from the frame's properties. See Section 32.2 [Frame Properties], page 471.

default-frame-name Variable

This variable holds the default name to assign to newly-created frames. This can be overridden by arguments to `make-frame`. This must be a string.

32.3 Frame Titles

Every frame has a title; most window managers display the frame title at the top of the frame. You can specify an explicit title with the `name` frame property. But normally you don't specify this explicitly, and XEmacs computes the title automatically.

XEmacs computes the frame title based on a template stored in the variable `frame-title-format`.

frame-title-format Variable

This variable specifies how to compute a title for a frame when you have not explicitly specified one.

The variable's value is actually a modeline construct, just like `modeline-format`. See Section 26.3.1 [Modeline Data], page 373.

frame-icon-title-format Variable

This variable specifies how to compute the title for an iconified frame, when you have not explicitly specified the frame title. This title appears in the icon itself.

x-set-frame-icon-pixmap *frame pixmap* &optional *mask* Function

This function sets the icon of the given frame to the given image instance, which should be an image instance object (as returned by `make-image-instance`), a glyph object (as returned by `make-glyph`), or `nil`. If a glyph object is given, the glyph will be instantiated on the frame to produce an image instance object.

If the given image instance has a mask, that will be used as the icon mask; however, not all window managers support this.

The window manager is also not required to support color pixmaps, only bitmaps (one plane deep).

If the image instance does not have a mask, then the optional third argument may be the image instance to use as the mask (it must be one plane deep). See Chapter 43 [Glyphs], page 625.

32.4 Deleting Frames

Frames remain potentially visible until you explicitly *delete* them. A deleted frame cannot appear on the screen, but continues to exist as a Lisp object until there are no references to it.

delete-frame &optional *frame* 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.

32.5 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 XEmacs.

device-frame-list &optional *device* Function

This function returns a list of all frames on *device*. If *device* is `nil`, the selected device will be used.

visible-frame-list &optional *device* Function

This function returns a list of just the currently visible frames. If *device* is specified only frames on that device will be returned. See Section 32.9 [Visibility of Frames], page 480. (TTY frames always count as "visible", even though only the selected one is actually displayed.)

next-frame &optional *frame minibuf* Function

The function `next-frame` lets you cycle conveniently through all the frames from an arbitrary starting point. It returns the "next" frame after *frame* in the cycle. If *frame* is omitted or `nil`, it defaults to the selected frame.

The second argument, *minibuf*, says which frames to consider:

`nil` Exclude minibuffer-only frames.

`visible` Consider all visible frames.

0 Consider all visible or iconified frames.

a window Consider only the frames using that particular window as their minibuffer.

the symbol `visible`
Include all visible frames.

`0` Include all visible and iconified frames.

anything else
Consider all frames.

previous-frame &optional *frame minibuf* Function

Like `next-frame`, but cycles through all frames in the opposite direction.

See also `next-window` and `previous-window`, in Section 31.5 [Cyclic Window Ordering], page 451.

32.6 Frames and Windows

Each window is part of one and only one frame; you can get the frame with `window-frame`.

frame-root-window &optional *frame* Function

This returns the root window of frame *frame*. *frame* defaults to the selected frame if not specified.

window-frame &optional *window* Function

This function returns the frame that *window* is on. *window* defaults to the selected window if omitted.

All the non-minibuffer windows in a frame are arranged in a cyclic order. The order runs from the frame's top window, which is at the upper left corner, down and to the right, until it reaches the window at the lower right corner (always the minibuffer window, if the frame has one), and then it moves back to the top.

frame-top-window *frame* Function

This returns the topmost, leftmost window of frame *frame*.

At any time, exactly one window on any frame is *selected within the frame*. The significance of this designation is that selecting the frame also selects this window. You can get the frame's current selected window with `frame-selected-window`.

frame-selected-window &optional *frame* Function

This function returns the window on *frame* that is selected within *frame*. *frame* defaults to the selected frame if not specified.

Conversely, selecting a window for XEmacs with `select-window` also makes that window selected within its frame. See Section 31.4 [Selecting Windows], page 450.

Another function that (usually) returns one of the windows in a frame is `minibuffer-window`. See Section 18.8 [Minibuffer Misc], page 278.

32.7 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 18.8 [Minibuffer Misc], page 278).

However, you can also create a frame with no minibuffer. Such a frame must use the minibuffer window of some other frame. When you create the frame, you can specify explicitly the minibuffer window to use (in some other frame). If you don't, then the minibuffer is found in the frame which is the value of the variable `default-minibuffer-frame`. Its value should be a frame which does have a minibuffer.

default-minibuffer-frame Variable

This variable specifies the frame to use for the minibuffer window, by default.

32.8 Input Focus

At any time, one frame in XEmacs is the *selected frame*. The selected window always resides on the selected frame. As the focus moves from device to device, the selected frame on each device is remembered and restored when the focus moves back to that device.

selected-frame &optional *device* Function

This function returns the selected frame on *device*. If *device* is not specified, the selected device will be used. If no frames exist on the device, `nil` is returned.

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.

When using a text-only terminal, there is no window manager; therefore, `select-frame` is the only way to switch frames, and the effect lasts until overridden by a subsequent call to `select-frame`. Only the selected terminal frame is actually displayed on the terminal. Each terminal screen except for the initial one has a number, and the number of the selected frame appears in the mode line after the word '`XEmacs`' (see Section 26.3.2 [Modeline Variables], page 374).

select-frame *frame* Function

This function selects frame *frame*, temporarily disregarding the focus of the X server if any. The selection of *frame* lasts until the next time the user does something to select a different frame, or until the next time this function is called.

Note that `select-frame` does not actually cause the window-system focus to be set to this frame, or the `select-frame-hook` or `deselect-frame-hook` to be run, until the next time that XEmacs is waiting for an event.

Also note that when the variable `focus-follows-mouse` is non-`nil`, the frame selection is temporary and is reverted when the current command terminates, much like the buffer selected by `set-buffer`. In order to effect a permanent focus change use `focus-frame`.

focus-frame *frame* Function

This function selects *frame* and gives it the window system focus. The operation of `focus-frame` is not affected by the value of `focus-follows-mouse`.

save-selected-frame *forms...* Macro

This macro records the selected frame, executes *forms* in sequence, then restores the earlier selected frame. The value returned is the value of the last form.

with-selected-frame *frame forms...* Macro

This macro records the selected frame, then selects *frame* and executes *forms* in sequence. After the last form is finished, the earlier selected frame is restored. The value returned is the value of the last form.

32.9 Visibility of Frames

An X window frame may be *visible*, *invisible*, or *iconified*. If it is visible, you can see its contents. If it is iconified, the frame's contents do not appear on the screen, but an icon does. If the frame is invisible, it doesn't show on the screen, not even as an icon.

Visibility is meaningless for TTY frames, since only the selected one is actually displayed in any case.

make-frame-visible &optional *frame* Command

This function makes frame *frame* visible. If you omit *frame*, it makes the selected frame visible.

make-frame-invisible &optional *frame* Command

This function makes frame *frame* invisible.

iconify-frame &optional *frame* Command

This function iconifies frame *frame*.

deiconify-frame &optional *frame* Command

This function de-iconifies frame *frame*. Under X, this is equivalent to `make-frame-visible`.

frame-visible-p *frame* Function

This returns whether *frame* is currently "visible" (actually in use for display). A frame that is not visible is not updated, and, if it works through a window system, may not show at all.

frame-iconified-p *frame* Function

This returns whether *frame* is iconified. Not all window managers use icons; some merely unmap the window, so this function is not the inverse of `frame-visible-p`. It is possible for a frame to not be visible and not be iconified either. However, if the frame is iconified, it will not be visible. (Under FSF Emacs, the functionality of this function is obtained through `frame-visible-p`.)

frame-totally-visible-p *frame* Function

This returns whether *frame* is not obscured by any other X windows. On TTY frames, this is the same as `frame-visible-p`.

32.10 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 XEmacs's X windows with these functions:

raise-frame &optional *frame* Command

This function raises frame *frame*.

lower-frame &optional *frame* Command

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). Under X, most ICCCM-compliant window managers will have an option to do this for you, but the following variables are provided in case you're using a broken WM. (Under FSF Emacs, the same functionality is provided through the `auto-raise` and `auto-lower` frame properties.)

auto-raise-frame Variable

This variable's value is `t` if frames will be raised to the top when selected.

auto-lower-frame Variable

This variable's value is `t` if frames will be lowered to the bottom when no longer selected.

Auto-raising and auto-lowering is implemented through functions attached to `select-frame-hook` and `deselect-frame-hook` (see Section 32.12 [Frame Hooks], page 481). Under normal circumstances, you should not call these functions directly.

default-select-frame-hook Function

This hook function implements the `auto-raise-frame` variable; it is for use as the value of `select-frame-hook`.

default-deselect-frame-hook Function

This hook function implements the `auto-lower-frame` variable; it is for use as the value of `deselect-frame-hook`.

32.11 Frame Configurations

A *frame configuration* records the current arrangement of frames, all their properties, and the window configuration of each one.

current-frame-configuration Function

This function returns a frame configuration list that describes the current arrangement of frames and their contents.

set-frame-configuration *configuration* Function

This function restores the state of frames described in *configuration*.

32.12 Hooks for Customizing Frame Behavior

XEmacs provides many hooks that are called at various times during a frame's lifetime. See Section 26.4 [Hooks], page 378.

create-frame-hook Variable

This hook is called each time a frame is created. The functions are called with one argument, the newly-created frame.

delete-frame-hook Variable

This hook is called each time a frame is deleted. The functions are called with one argument, the about-to-be-deleted frame.

select-frame-hook Variable

This is a normal hook that is run just after a frame is selected. The function `default-select-frame-hook`, which implements auto-raising (see Section 32.10 [Raising and Lowering], page 480), is normally attached to this hook.

Note that calling `select-frame` does not necessarily set the focus: The actual window-system focus will not be changed until the next time that XEmacs is waiting for an event, and even then, the window manager may refuse the focus-change request.

deselect-frame-hook Variable

This is a normal hook that is run just before a frame is deselected (and another frame is selected). The function `default-deselect-frame-hook`, which implements auto-lowering (see Section 32.10 [Raising and Lowering], page 480), is normally attached to this hook.

map-frame-hook Variable

This hook is called each time a frame is mapped (i.e. made visible). The functions are called with one argument, the newly mapped frame.

unmap-frame-hook Variable

This hook is called each time a frame is unmapped (i.e. made invisible or iconified). The functions are called with one argument, the newly unmapped frame.

33 Consoles and Devices

A *console* is an object representing a single input connection to XEmacs, such as an X display or a TTY connection. It is possible for XEmacs to have frames on multiple consoles at once (even on heterogeneous types – you can simultaneously have a frame on an X display and a TTY connection). Normally, there is only one console in existence.

A *device* is an object representing a single output device, such as a particular screen on an X display. (Usually there is exactly one device per X console connection, but there may be more than one if you have a multi-headed X display. For TTY connections, there is always exactly one device per console.)

Each device has one or more *frames* in which text can be displayed. For X displays and the like, a frame corresponds to the normal window-system concept of a window. Frames can overlap, be displayed at various locations within the display, be resized, etc. For TTY, only one frame can be displayed at a time, and it occupies the entire TTY display area.

However, you can still define multiple frames and switch between them. Their contents are entirely separate from each other. These sorts of frames resemble the "virtual console" capability provided under Linux or the multiple screens provided by the multiplexing program '`screen`' under Unix.

When you start up XEmacs, an initial console and device are created to receive input and display frames on. This will either be an X display or a TTY connection, depending on what mode you started XEmacs in (this is determined by the '`DISPLAY`' environment variable, the '`-nw`', '`-t`' and '`-display`' command-line options, etc.).

You can connect to other X displays and TTY connections by creating new console objects, and to other X screens on an existing display by creating new device objects, as described below. Many functions (for example the frame-creation functions) take an optional device argument specifying which device the function pertains to. If the argument is omitted, it defaults to the selected device (see below).

consolep *object* Function

This returns non-`nil` if *object* is a console.

devicep *object* Function

This returns non-`nil` if *object* is a device.

33.1 Basic Console Functions

console-list Function

This function returns a list of all existing consoles.

console-device-list &optional *console* Function

This function returns a list of all devices on *console*. If *console* is `nil`, the selected console will be used.

33.2 Basic Device Functions

device-list Function

This function returns a list of all existing devices.

device-or-frame-p *object* Function

This function returns non-`nil` if *object* is a device or frame. This function is useful because devices and frames are similar in many respects and many functions can operate on either one.

device-frame-list *device* Function

This function returns a list of all frames on *device*.

frame-device *frame* Function

This function returns the device that *frame* is on.

33.3 Console Types and Device Classes

Every device is of a particular *type*, which describes how the connection to that device is made and how the device operates, and a particular *class*, which describes other characteristics of the device (currently, the color capabilities of the device).

The currently-defined device types are

`x` A connection to an X display (such as '`willow:0`').

`tty` A connection to a tty (such as '`/dev/ttyp3`').

`stream` A stdio connection. This describes a device for which input and output is only possible in a stream-like fashion, such as when XEmacs in running in batch mode. The very first device created by XEmacs is a terminal device and is used to print out messages of various sorts (for example, the help message when you use the '`-help`' command-line option).

The currently-defined device classes are

`color` A color device.

`grayscale`
A grayscale device (a device that can display multiple shades of gray, but no color).

`mono` A device that can only display two colors (e.g. black and white).

device-type *device* Function

This function returns the type of *device*. This is a symbol whose name is one of the device types mentioned above.

device-or-frame-type *device-or-frame* Function

This function returns the type of *device-or-frame*.

device-class *device* Function

This function returns the class (color behavior) of *device*. This is a symbol whose name is one of the device classes mentioned above.

valid-device-type-p *device-type* Function

This function returns whether *device-type* (which should be a symbol) species a valid device type.

valid-device-class-p *device-class* Function

This function returns whether *device-class* (which should be a symbol) species a valid device class.

terminal-device Variable

This variable holds the initial terminal device object, which represents XEmacs's stdout.

33.4 Connecting to a Console or Device

make-device &optional *type device-data* Function

This function creates a new device.

The following two functions create devices of specific types and are written in terms of `make-device`.

make-tty-device &optional *tty terminal-type* Function

This function creates a new tty device on *tty*. This also creates the tty's first frame. *tty* should be a string giving the name of a tty device file (e.g. '`/dev/ttyp3`' under SunOS et al.), as returned by the '`tty`' command issued from the Unix shell. A value of `nil` means use the stdin and stdout as passed to XEmacs from the shell. If *terminal-type* is non-`nil`, it should be a string specifying the type of the terminal attached to the specified tty. If it is `nil`, the terminal type will be inferred from the '`TERM`' environment variable.

make-x-device &optional *display argv-list* Function

This function creates a new device connected to *display*. Optional argument *argv-list* is a list of strings describing command line options.

delete-device *device* Function

This function deletes *device*, permanently eliminating it from use. This disconnects XEmacs's connection to the device.

create-device-hook Variable

This variable, if non-`nil`, should contain a list of functions, which are called when a device is created.

delete-device-hook Variable

This variable, if non-`nil`, should contain a list of functions, which are called when a device is deleted.

console-live-p *object* Function

This function returns non-`nil` if *object* is a console that has not been deleted.

device-live-p *object* Function

This function returns non-`nil` if *object* is a device that has not been deleted.

device-x-display *device* Function

This function returns the X display which *device* is connected to, if *device* is an X device.

33.5 The Selected Console and Device

select-console *console* Function

This function selects the console *console*. Subsequent editing commands apply to its selected device, selected frame, and selected window. The selection of *console* lasts until the next time the user does something to select a different console, or until the next time this function is called.

selected-console Function

This function returns the console which is currently active.

select-device *device* Function

This function selects the device *device*.

selected-device &optional *console* Function

This function returns the device which is currently active. If optional *console* is non-`nil`, this function returns the device that would be currently active if *console* were the selected console.

33.6 Console and Device I/O

console-disable-input *console* Function

This function disables input on console *console*.

console-enable-input *console* Function

This function enables input on console *console*.

Each device has a *baud rate* value associated with it. On most systems, changing this value will affect the amount of padding and other strategic decisions made during redisplay.

device-baud-rate &optional *device* Function

This function returns the output baud rate of *device*.

set-device-baud-rate *device rate* Function

This function sets the output baud rate of *device* to *rate*.

34 Positions

A *position* is the index of a character in the text of a buffer. More precisely, a position identifies the place between two characters (or before the first character, or after the last character), so we can speak of the character before or after a given position. However, we often speak of the character "at" a position, meaning the character after that position.

Positions are usually represented as integers starting from 1, but can also be represented as *markers*—special objects that relocate automatically when text is inserted or deleted so they stay with the surrounding characters. See Chapter 35 [Markers], page 499.

34.1 Point

Point is a special buffer position used by many editing commands, including the self-inserting typed characters and text insertion functions. Other commands move point through the text to allow editing and insertion at different places.

Like other positions, point designates a place between two characters (or before the first character, or after the last character), rather than a particular character. Usually terminals display the cursor over the character that immediately follows point; point is actually before the character on which the cursor sits.

The value of point is a number between 1 and the buffer size plus 1. If narrowing is in effect (see Section 34.4 [Narrowing], page 496), 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 31.9 [Window Point], page 458, for more details.

point &optional *buffer* Function

This function returns the value of point in *buffer*, as an integer. *buffer* defaults to the current buffer if omitted.

```
(point)
     ⇒ 175
```

point-min &optional *buffer* Function

This function returns the minimum accessible value of point in *buffer*. This is normally 1, but if narrowing is in effect, it is the position of the start of the region that you narrowed to. (See Section 34.4 [Narrowing], page 496.) *buffer* defaults to the current buffer if omitted.

point-max &optional *buffer* Function

This function returns the maximum accessible value of point in *buffer*. This is `(1+ (buffer-size buffer))`, unless narrowing is in effect, in which case it is the position of the end of the region that you narrowed to. (see Section 34.4 [Narrowing], page 496). *buffer* defaults to the current buffer if omitted.

buffer-end *flag* &optional *buffer* Function

This function returns `(point-min buffer)` if *flag* is less than 1, `(point-max buffer)` otherwise. The argument *flag* must be a number. *buffer* defaults to the current buffer if omitted.

buffer-size &optional *buffer* Function

This function returns the total number of characters in *buffer*. In the absence of any narrowing (see Section 34.4 [Narrowing], page 496), `point-max` returns a value one larger than this. *buffer* defaults to the current buffer if omitted.

```
(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.

34.2 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 34.1 [Point], page 487.

34.2.1 Motion by Characters

These functions move point based on a count of characters. `goto-char` is the fundamental primitive; the other functions use that.

goto-char *position* &optional *buffer* Command

This function sets point in `buffer` to the value *position*. If *position* is less than 1, it moves point to the beginning of the buffer. If *position* is greater than the length of the buffer, it moves point to the end. *buffer* defaults to the current buffer if omitted.

If narrowing is in effect, *position* still counts from the beginning of the buffer, but point cannot go outside the accessible portion. If *position* is out of range, `goto-char` moves point to the beginning or the end of the accessible portion.

When this function is called interactively, *position* is the numeric prefix argument, if provided; otherwise it is read from the minibuffer.

`goto-char` returns *position*.

forward-char &optional *count buffer* Command

This function moves point *count* characters forward, towards the end of the buffer (or backward, towards the beginning of the buffer, if *count* is negative). If the function attempts to move point past the beginning or end of the buffer (or the limits of the accessible portion, when narrowing is in effect), an error is signaled with error code `beginning-of-buffer` or `end-of-buffer`. *buffer* defaults to the current buffer if omitted.

In an interactive call, *count* is the numeric prefix argument.

backward-char &optional *count buffer* — Command

This function moves point *count* characters backward, towards the beginning of the buffer (or forward, towards the end of the buffer, if *count* is negative). If the function attempts to move point past the beginning or end of the buffer (or the limits of the accessible portion, when narrowing is in effect), an error is signaled with error code `beginning-of-buffer` or `end-of-buffer`. *buffer* defaults to the current buffer if omitted.

In an interactive call, *count* is the numeric prefix argument.

34.2.2 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 38 [Syntax Tables], page 569.

forward-word *count* &optional *buffer* — 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`. *buffer* defaults to the current buffer if omitted.

In an interactive call, *count* is set to the numeric prefix argument.

backward-word *count* &optional *buffer* — Command

This function is just like `forward-word`, except that it moves backward until encountering the front of a word, rather than forward. *buffer* defaults to the current buffer if omitted.

In an interactive call, *count* is set to the numeric prefix argument.

This function is rarely used in programs, as it is more efficient to call `forward-word` with a negative argument.

words-include-escapes — Variable

This variable affects the behavior of `forward-word` and everything that uses it. If it is non-`nil`, then characters in the "escape" and "character quote" syntax classes count as part of words. Otherwise, they do not.

34.2.3 Motion to an End of the Buffer

To move point to the beginning of the buffer, write:

```
(goto-char (point-min))
```

Likewise, to move to the end of the buffer, use:

```
(goto-char (point-max))
```

Here are two commands that users use to do these things. They are documented here to warn you not to use them in Lisp programs, because they set the mark and display messages in the echo area.

beginning-of-buffer &optional *n* Command

This function moves point to the beginning of the buffer (or the limits of the accessible portion, when narrowing is in effect), setting the mark at the previous position. If *n* is non-`nil`, then it puts point *n* tenths of the way from the beginning of the buffer.

In an interactive call, *n* is the numeric prefix argument, if provided; otherwise *n* defaults to `nil`.

Don't use this function in Lisp programs!

end-of-buffer &optional *n* Command

This function moves point to the end of the buffer (or the limits of the accessible portion, when narrowing is in effect), setting the mark at the previous position. If *n* is non-`nil`, then it puts point *n* tenths of the way from the end of the buffer.

In an interactive call, *n* is the numeric prefix argument, if provided; otherwise *n* defaults to `nil`.

Don't use this function in Lisp programs!

34.2.4 Motion by Text Lines

Text lines are portions of the buffer delimited by newline characters, which are regarded as part of the previous line. The first text line begins at the beginning of the buffer, and the last text line ends at the end of the buffer whether or not the last character is a newline. The division of the buffer into text lines is not affected by the width of the window, by line continuation in display, or by how tabs and control characters are displayed.

goto-line *line* Command

This function moves point to the front of the *line*th line, counting from line 1 at beginning of the buffer. If *line* is less than 1, it moves point to the beginning of the buffer. If *line* is greater than the number of lines in the buffer, it moves point to the end of the buffer—that is, the *end of the last line* of the buffer. This is the only case in which `goto-line` does not necessarily move to the beginning of a line.

If narrowing is in effect, then *line* still counts from the beginning of the buffer, but point cannot go outside the accessible portion. So `goto-line` moves point to the beginning or end of the accessible portion, if the line number specifies an inaccessible position.

The return value of `goto-line` is the difference between *line* and the line number of the line to which point actually was able to move (in the full buffer, before taking account of narrowing). Thus, the value is positive if the scan encounters the real end of the buffer. The value is zero if scan encounters the end of the accessible portion but not the real end of the buffer.

In an interactive call, *line* is the numeric prefix argument if one has been provided. Otherwise *line* is read in the minibuffer.

beginning-of-line &optional *count buffer* 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. *buffer* defaults to the current buffer if omitted.

If this function reaches the end of the buffer (or of the accessible portion, if narrowing is in effect), it positions point there. No error is signaled.

end-of-line &optional *count buffer* 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. *buffer* defaults to the current buffer if omitted.

If this function reaches the end of the buffer (or of the accessible portion, if narrowing is in effect), it positions point there. No error is signaled.

forward-line &optional *count buffer* Command

This function moves point forward *count* lines, to the beginning of the line. If *count* is negative, it moves point −*count* lines backward, to the beginning of a line. If *count* is zero, it moves point to the beginning of the current line. *buffer* defaults to the current buffer if omitted.

If `forward-line` encounters the beginning or end of the buffer (or of the accessible portion) before finding that many lines, it sets point there. No error is signaled.

`forward-line` returns the difference between *count* and the number of lines actually moved. If you attempt to move down five lines from the beginning of a buffer that has only three lines, point stops at the end of the last line, and the value will be 2.

In an interactive call, *count* is the numeric prefix argument.

count-lines *start end* Function

This function returns the number of lines between the positions *start* and *end* in the current buffer. If *start* and *end* are equal, then it returns 0. Otherwise it returns at least 1, even if *start* and *end* are on the same line. This is because the text between them, considered in isolation, must contain at least one line unless it is empty.

Here is an example of using `count-lines`:

```
(defun current-line ()
  "Return the vertical position of point..."
  (+ (count-lines (window-start) (point))
     (if (= (current-column) 0) 1 0)
     -1))
```

Also see the functions `bolp` and `eolp` in Section 36.1 [Near Point], page 511. These functions do not move point, but test whether it is already at the beginning or end of a line.

34.2.5 Motion by Screen Lines

The line functions in the previous section count text lines, delimited only by newline characters. By contrast, these functions count screen lines, which are defined by the way the text appears on the screen. A text line is a single screen line if it is short enough to fit the width of the selected window, but otherwise it may occupy several screen lines.

In some cases, text lines are truncated on the screen rather than continued onto additional screen lines. In these cases, `vertical-motion` moves point much like `forward-line`. See Section 45.2 [Truncation], page 648.

Because the width of a given string depends on the flags that control the appearance of certain characters, `vertical-motion` behaves differently, for a given piece of text, depending on the buffer it is in, and even on the selected window (because the width, the truncation flag, and display table may vary between windows). See Section 45.10 [Usual Display], page 658.

These functions scan text to determine where screen lines break, and thus take time proportional to the distance scanned. If you intend to use them heavily, Emacs provides caches which may improve the performance of your code. See Section 34.2.4 [Text Lines], page 490.

vertical-motion *count* &optional *window pixels* — Function

> This function moves point to the start of the frame line *count* frame lines down from the frame line containing point. If *count* is negative, it moves up instead. The optional second argument *window* may be used to specify a window other than the selected window in which to perform the motion.
>
> Normally, `vertical-motion` 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. If the optional third argument, *pixels* is non-`nil`, the vertical pixel height of the motion which took place is returned instead of the actual number of lines moved. A motion of zero lines returns the height of the current line.
>
> Note that `vertical-motion` sets *window*'s buffer's point, not *window*'s point. (This differs from FSF Emacs, which buggily always sets current buffer's point, regardless of *window*.)

vertical-motion-pixels *count* &optional *window how* — Function

> This function moves point to the start of the frame line *pixels* vertical pixels down from the frame line containing point, or up if *pixels* is negative. The optional second argument *window* is the window to move in, and defaults to the selected window. The optional third argument *how* specifies the stopping condition. A negative integer indicates that the motion should be no more than *pixels*. A positive value indicates that the motion should be at least *pixels*. Any other value indicates that the motion should be as close as possible to *pixels*.

move-to-window-line *count* &optional *window* — Command

> This function moves point with respect to the text currently displayed in *window*, which defaults to the selected window. It moves point to the beginning of the screen line *count* screen lines from the top of the window. If *count* is

negative, that specifies a position $-count$ lines from the bottom (or the last line of the buffer, if the buffer ends above the specified screen position).

If *count* is `nil`, then point moves to the beginning of the line in the middle of the window. If the absolute value of *count* is greater than the size of the window, then point moves to the place that would appear on that screen line if the window were tall enough. This will probably cause the next redisplay to scroll to bring that location onto the screen.

In an interactive call, *count* is the numeric prefix argument.

The value returned is the window line number point has moved to, with the top line in the window numbered 0.

34.2.6 Moving over Balanced Expressions

Here are several functions concerned with balanced-parenthesis expressions (also called *sexps* in connection with moving across them in XEmacs). The syntax table controls how these functions interpret various characters; see Chapter 38 [Syntax Tables], page 569. See Section 38.5 [Parsing Expressions], page 575, for lower-level primitives for scanning sexps or parts of sexps. For user-level commands, see section "Lists and Sexps" in *XEmacs Reference Manual*.

forward-list &optional *arg* Command

This function moves forward across *arg* balanced groups of parentheses. (Other syntactic entities such as words or paired string quotes are ignored.) *arg* defaults to 1 if omitted. If *arg* is negative, move backward across that many groups of parentheses.

backward-list &optional *arg* Command

This function moves backward across *arg* balanced groups of parentheses. (Other syntactic entities such as words or paired string quotes are ignored.) *arg* defaults to 1 if omitted. If *arg* is negative, move forward across that many groups of parentheses.

up-list *arg* Command

This function moves forward out of *arg* levels of parentheses. A negative argument means move backward but still to a less deep spot.

down-list *arg* Command

This function moves forward into *arg* levels of parentheses. A negative argument means move backward but still go deeper in parentheses ($-arg$ levels).

forward-sexp &optional *arg* Command

This function moves forward across *arg* balanced expressions. Balanced expressions include both those delimited by parentheses and other kinds, such as words and string constants. *arg* defaults to 1 if omitted. If *arg* is negative, move backward across that many balanced expressions. For example,

```
---------- Buffer: foo ----------
(concat⋆ "foo " (car x) y z)
---------- Buffer: foo ----------
```

```
(forward-sexp 3)
     ⇒ nil

---------- Buffer: foo ----------
(concat "foo " (car x) y⋆ z)
---------- Buffer: foo ----------
```

backward-sexp &optional *arg* — Command

This function moves backward across *arg* balanced expressions. *arg* defaults to 1 if omitted. If *arg* is negative, move forward across that many balanced expressions.

beginning-of-defun &optional *arg* — Command

This function moves back to the *arg*th beginning of a defun. If *arg* is negative, this actually moves forward, but it still moves to the beginning of a defun, not to the end of one. *arg* defaults to 1 if omitted.

end-of-defun &optional *arg* — Command

This function moves forward to the *arg*th end of a defun. If *arg* is negative, this actually moves backward, but it still moves to the end of a defun, not to the beginning of one. *arg* defaults to 1 if omitted.

defun-prompt-regexp — User Option

If non-`nil`, this variable holds a regular expression that specifies what text can appear before the open-parenthesis that starts a defun. That is to say, a defun begins on a line that starts with a match for this regular expression, followed by a character with open-parenthesis syntax.

34.2.7 Skipping Characters

The following two functions move point over a specified set of characters. For example, they are often used to skip whitespace. For related functions, see Section 38.4 [Motion and Syntax], page 575.

skip-chars-forward *character-set* &optional *limit buffer* — Function

This function moves point in *buffer* forward, skipping over a given set of characters. It examines the character following point, then advances point if the character matches *character-set*. This continues until it reaches a character that does not match. The function returns `nil`. *buffer* defaults to the current buffer if omitted.

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 non-letter, and `"^a-zA-Z"` skips non-letters stopping before the first letter. See Section 37.2 [Regular Expressions], page 550.

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 buffer* Function

This function moves point backward, skipping characters that match *character-set*, until *limit*. It just like `skip-chars-forward` except for the direction of motion.

34.3 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 31.16 [Window Configurations], page 468 and see Section 32.11 [Frame Configurations], page 481).

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 134).

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 XEmacs.

`save-excursion` does not save the values of point and the mark for other buffers, so changes in other buffers remain in effect after `save-excursion` exits.

Likewise, `save-excursion` does not restore window-buffer correspondences altered by functions such as `switch-to-buffer`. One way to restore these correspondences, and the selected window, is to use `save-window-excursion` inside `save-excursion` (see Section 31.16 [Window Configurations], page 468).

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)))
```

save-current-buffer *forms...* — Special Form

This special form is similar to `save-excursion` but it only saves and restores the current buffer. Beginning with XEmacs 20.3, `save-current-buffer` is a primitive.

with-current-buffer *buffer forms...* — Special Form

This special form evaluates *forms* with *buffer* as the current buffer. It returns the value of the last form.

with-temp-file *file forms...* — Special Form

This special form creates a new buffer, evaluates *forms* there, and writes the buffer to *file*. It returns the value of the last form evaluated.

save-selected-window *forms...* — Special Form

This special form is similar to `save-excursion` but it saves and restores the selected window and nothing else.

34.4 Narrowing

Narrowing means limiting the text addressable by XEmacs editing commands to a limited range of characters in a buffer. The text that remains addressable is called the *accessible portion* of the buffer.

Narrowing is specified with two buffer positions which become the beginning and end of the accessible portion. For most editing commands and most Emacs primitives, these positions replace the values of the beginning and end of the buffer. While narrowing is in effect, no text outside the accessible portion is displayed, and point cannot move outside the accessible portion.

Values such as positions or line numbers, which usually count from the beginning of the buffer, do so despite narrowing, but the functions which use them refuse to operate on text that is inaccessible.

The commands for saving buffers are unaffected by narrowing; they save the entire buffer regardless of any narrowing.

narrow-to-region *start end* &optional *buffer* Command

This function sets the accessible portion of *buffer* to start at *start* and end at *end*. Both arguments should be character positions. *buffer* defaults to the current buffer if omitted.

In an interactive call, *start* and *end* are set to the bounds of the current region (point and the mark, with the smallest first).

narrow-to-page &optional *move-count* Command

This function sets the accessible portion of the current buffer to include just the current page. An optional first argument *move-count* non-`nil` means to move forward or backward by *move-count* pages and then narrow. The variable `page-delimiter` specifies where pages start and end (see Section 37.8 [Standard Regexps], page 566).

In an interactive call, *move-count* is set to the numeric prefix argument.

widen &optional *buffer* Command

This function cancels any narrowing in *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)))
```

buffer defaults to the current buffer if omitted.

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 134). Therefore, this construct is a clean way to narrow a buffer temporarily.

The value returned by `save-restriction` is that returned by the last form in *body*, or `nil` if no body forms were given.

Caution: it is easy to make a mistake when using the `save-restriction` construct. Read the entire description here before you try it.

If *body* changes the current buffer, `save-restriction` still restores the restrictions on the original buffer (the buffer whose restrictions it saved from), but it does not restore the identity of the current buffer.

`save-restriction` does *not* restore point and the mark; use `save-excursion` for that. If you use both `save-restriction` and `save-excursion` together, `save-excursion` should come first (on the outside). Otherwise, the old point value would be restored with temporary narrowing still in effect. If the old point value were outside the limits of the temporary narrowing, this would fail to restore it accurately.

The `save-restriction` special form records the values of the beginning and end of the accessible portion as distances from the beginning and end of the buffer. In other words, it records the amount of inaccessible text before and after the accessible portion.

This method yields correct results if *body* does further narrowing. However, `save-restriction` can become confused if the body widens and then make changes outside the range of the saved narrowing. When this is what you want to do, `save-restriction` is not the right tool for the job. Here is what you must use instead:

```
(let ((beg (point-min-marker))
      (end (point-max-marker)))
  (unwind-protect
      (progn body)
    (save-excursion
      (set-buffer (marker-buffer beg))
      (narrow-to-region beg end))))
```

Here is a simple example of correct use of `save-restriction`:

```
---------- Buffer: foo ----------
This is the contents of foo
This is the contents of foo
This is the contents of foo⋆
---------- Buffer: foo ----------

(save-excursion
  (save-restriction
    (goto-char 1)
    (forward-line 2)
    (narrow-to-region 1 (point))
    (goto-char (point-min))
    (replace-string "foo" "bar")))

---------- Buffer: foo ----------
This is the contents of bar
This is the contents of bar
This is the contents of foo⋆
---------- Buffer: foo ----------
```

35 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.

35.1 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 34 [Positions], page 487, for a complete description of positions.

A marker has two attributes: the marker position, and the marker buffer. The marker position is an integer that is equivalent (at a given time) to the marker as a position in that buffer. But the marker's position value can change often during the life of the marker. Insertion and deletion of text in the buffer relocate the marker. The idea is that a marker positioned between two characters remains between those two characters despite insertion and deletion elsewhere in the buffer. Relocation changes the integer equivalent of the marker.

Deleting text around a marker's position leaves the marker between the characters immediately before and after the deleted text. Inserting text at the position of a marker normally leaves the marker in front of the new text—unless it is inserted with `insert-before-markers` (see Section 36.4 [Insertion], page 513).

Insertion and deletion in a buffer must check all the markers and relocate them if necessary. This slows processing in a buffer with a large number of markers. For this reason, it is a good idea to make a marker point nowhere if you are sure you don't need it any more. Unreferenced markers are garbage collected eventually, but until then will continue to use time if they do point somewhere.

Because it is common to perform arithmetic operations on a marker position, most of the arithmetic operations (including `+` and `-`) accept markers as arguments. In such cases, the marker stands for its current position.

Note that you can use extents to achieve the same functionality, and more, as markers. (Markers were defined before extents, which is why they both continue to exist.) A zero-length extent with the `detachable` property removed is almost identical to a marker. (See Section 40.3 [Extent Endpoints], page 586, for more information on zero-length extents.)

In particular:

- In order to get marker-like behavior in a zero-length extent, the `detachable` property must be removed (otherwise, the extent will disappear when text near it is deleted) and exactly one endpoint must be closed (if both endpoints are closed, the extent will expand to contain text inserted where it is located).
- If a zero-length extent has the `end-open` property but not the `start-open` property (this is the default), text inserted at the extent's location causes the extent to move forward, just like a marker.
- If a zero-length extent has the `start-open` property but not the `end-open` property, text inserted at the extent's location causes the extent to remain before the text, like what happens to markers when `insert-before-markers` is used.

- Markers end up after or before inserted text depending on whether `insert` or `insert-before-markers` was called. These functions do not affect zero-length extents differently; instead, the presence or absence of the `start-open` and `end-open` extent properties determines this, as just described.
- Markers are automatically removed from a buffer when they are no longer in use. Extents remain around until explicitly removed from a buffer.
- Many functions are provided for listing the extents in a buffer or in a region of a buffer. No such functions exist for markers.

Here are examples of creating markers, setting markers, and moving point to markers:

```
;; Make a new marker that initially does not point anywhere:
(setq m1 (make-marker))
     ⇒ #<marker in no buffer>

;; Set m1 to point between the 99th and 100th characters
;;   in the current buffer:
(set-marker m1 100)
     ⇒ #<marker at 100 in markers.texi>

;; Now insert one character at the beginning of the buffer:
(goto-char (point-min))
     ⇒ 1
(insert "Q")
     ⇒ nil

;; m1 is updated appropriately.
m1
     ⇒ #<marker at 101 in markers.texi>

;; Two markers that point to the same position
;;   are not eq, but they are equal.
(setq m2 (copy-marker m1))
     ⇒ #<marker at 101 in markers.texi>
(eq m1 m2)
     ⇒ nil
(equal m1 m2)
     ⇒ t

;; When you are finished using a marker, make it point nowhere.
(set-marker m1 nil)
     ⇒ #<marker in no buffer>
```

35.2 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 or either an integer, a character, or a marker. The latter tests are useful in connection with the arithmetic functions that work with any of markers, integers, or characters.

markerp *object* Function

This function returns `t` if *object* is a marker, `nil` otherwise. Note that integers are not markers, even though many functions will accept either a marker or an integer.

integer-or-marker-p *object* Function

This function returns `t` if *object* is an integer or a marker, `nil` otherwise.

integer-char-or-marker-p *object* Function

This function returns `t` if *object* is an integer, a character, or a marker, `nil` otherwise.

number-or-marker-p *object* Function

This function returns `t` if *object* is a number (either kind) or a marker, `nil` otherwise.

number-char-or-marker-p *object* Function

This function returns `t` if *object* is a number (either kind), a character, or a marker, `nil` otherwise.

35.3 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 created marker that does not point anywhere.

```
(make-marker)
     ⇒ #<marker in no buffer>
```

point-marker &optional *dont-copy-p buffer* Function

This function returns a marker that points to the present position of point in *buffer*, which defaults to the current buffer. See Section 34.1 [Point], page 487. For an example, see `copy-marker`, below.

Internally, a marker corresponding to point is always maintained. Normally the marker returned by `point-marker` is a copy; you may modify it with reckless abandon. However, if optional argument *dont-copy-p* is non-`nil`, then the real point-marker is returned; modifying the position of this marker will move point. It is illegal to change the buffer of it, or make it point nowhere.

point-min-marker &optional *buffer* Function

This function returns a new marker that points to the beginning of the accessible portion of *buffer*, which defaults to the current buffer. This will be the beginning of the buffer unless narrowing is in effect. See Section 34.4 [Narrowing], page 496.

point-max-marker &optional *buffer* Function

This function returns a new marker that points to the end of the accessible portion of *buffer*, which defaults to the current buffer. This will be the end of the buffer unless narrowing is in effect. See Section 34.4 [Narrowing], page 496.

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 integer argument less than 1, `copy-marker` returns a new marker that points to the beginning of the current buffer. If passed an integer argument greater than the length of the buffer, `copy-marker` returns a new marker that points to the end of the buffer.

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

(point)
     ⇒ 2139

(set-marker p 3000)
     ⇒ #<marker at 3000 in markers.texi>

(point)
     ⇒ 2139

(setq p (point-marker t))
     ⇒ #<marker at 2139 in markers.texi>
```

```
(set-marker p 3000)
     ⇒ #<marker at 3000 in markers.texi>
(point)
     ⇒ 3000
(copy-marker 0)
     ⇒ #<marker at 1 in markers.texi>
(copy-marker 20000)
     ⇒ #<marker at 7572 in markers.texi>
```

35.4 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 are considered `equal` (even though not `eq`) to each other if they have the same position and buffer, or if they both point nowhere.

35.5 Changing Marker Positions

This section describes how to change the position of an existing marker. When you do this, be sure you know whether the marker is used outside of your program, and, if so, what effects will result from moving it—otherwise, confusing things may happen in other parts of Emacs.

set-marker *marker position* &optional *buffer* — Function

This function moves *marker* to *position* in *buffer*. If *buffer* is not provided, it defaults to the current buffer.

If *position* is less than 1, `set-marker` moves *marker* to the beginning of the buffer. If *position* is greater than the size of the buffer, `set-marker` moves marker to the end of the buffer. If *position* is `nil` or a marker that points nowhere, then *marker* is set to point nowhere.

The value returned is *marker*.

```
(setq m (point-marker))
     ⇒ #<marker at 4714 in markers.texi>
(set-marker m 55)
     ⇒ #<marker at 55 in markers.texi>
(setq b (get-buffer "foo"))
     ⇒ #<buffer foo>
(set-marker m 0 b)
     ⇒ #<marker at 1 in foo>
```

move-marker *marker position* &optional *buffer* Function

This is another name for `set-marker`.

35.6 The Mark

One special marker in each buffer is designated *the mark*. It records a position for the user for the sake of commands such as *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.

Once the mark "exists" in a buffer, it normally never ceases to exist. However, it may become *inactive*, and usually does so after each command (other than simple motion commands and some commands that explicitly activate the mark). When the mark is active, the region between point and the mark is called the *active region* and is highlighted specially.

Many commands are designed so that when called interactively they operate on the text between point and the mark. Such commands work only when an active region exists, i.e. when the mark is active. (The reason for this is to prevent you from accidentally deleting or changing large chunks of your text.) If you are writing such a command, don't examine the mark directly; instead, use `interactive` with the '`r`' specification. This provides the values of point and the mark as arguments to the command in an interactive call, but permits other Lisp programs to specify arguments explicitly, and automatically signals an error if the command is called interactively when no active region exists. See Section 19.2.2 [Interactive Codes], page 283.

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.") However, only one active region can exist at a time. Activating the mark in one buffer automatically deactivates an active mark in any other buffer. Note that the user can explicitly activate a mark at any time by using the command `activate-region` (normally bound to *M-C-z*) or by using the command `exchange-point-and-mark` (normally bound to *C-x C-x*), which has the side effect of activating the mark.

Some people do not like active regions, so they disable this behavior by setting the variable `zmacs-regions` to `nil`. This makes the mark always active (except when a buffer is just created and the mark points nowhere), and turns off the highlighting of the region between point and the mark. Commands that explicitly retrieve the value of the mark should make sure that they behave correctly and consistently irrespective of the setting of `zmacs-regions`; some primitives are provided to ensure this behavior.

In addition to the mark, each buffer has a *mark ring* which is a list of markers containing previous values of the mark. When editing commands change the mark, they should normally save the old value of the mark on the mark ring. The variable `mark-ring-max` specifies the maximum number of entries in the mark ring; once the list becomes this long, adding a new element deletes the last element.

mark &optional *force buffer* Function

This function returns *buffer*'s mark position as an integer. *buffer* defaults to the current buffer if omitted.

If the mark is inactive, `mark` normally returns `nil`. However, if *force* is non-`nil`, then `mark` returns the mark position anyway—or `nil`, if the mark is not yet set for the buffer.

(Remember that if *zmacs-regions* is `nil`, the mark is always active as long as it exists, and the *force* argument will have no effect.)

If you are using this in an editing command, you are most likely making a mistake; see the documentation of `set-mark` below.

mark-marker *inactive-p buffer* Function

This function returns *buffer*'s mark. *buffer* defaults to the current buffer if omitted. This is the very marker that records the mark location inside XEmacs, 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.

If the mark is inactive, `mark-marker` normally returns `nil`. However, if *force* is non-`nil`, then `mark-marker` returns the mark anyway.

```
(setq m (mark-marker))
     ⇒ #<marker at 3420 in markers.texi>
(set-marker m 100)
     ⇒ #<marker at 100 in markers.texi>
(mark-marker)
     ⇒ #<marker at 100 in markers.texi>
```

Like any marker, this marker can be set to point at any buffer you like. We don't recommend that you make it point at any buffer other than the one of which it is the mark. If you do, it will yield perfectly consistent, but rather odd, results.

set-mark *position* &optional *buffer* Function

This function sets `buffer`'s mark to *position*, and activates the mark. *buffer* defaults to the current buffer if omitted. 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 XEmacs 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))).
```

exchange-point-and-mark &optional *dont-activate-region* — Command

This function exchanges the positions of point and the mark. It is intended for interactive use. The mark is also activated unless *dont-activate-region* is non-`nil`.

push-mark &optional *position nomsg activate buffer* — Function

This function sets *buffer*'s mark to *position*, and pushes a copy of the previous mark onto `mark-ring`. *buffer* defaults to the current buffer if omitted. If *position* is `nil`, then the value of point is used. `push-mark` returns `nil`.

If the last global mark pushed was not in *buffer*, also push *position* on the global mark ring (see below).

The function `push-mark` normally *does not* activate the mark. To do that, specify `t` for the argument *activate*.

A '`Mark set`' message is displayed unless *nomsg* is non-`nil`.

pop-mark — Function

This function pops off the top element of `mark-ring` and makes that mark become the buffer's actual mark. This does not move point in the buffer, and it does nothing if `mark-ring` is empty. It deactivates the mark.

The return value is not meaningful.

mark-ring — Variable

The value of this buffer-local variable is the list of saved former marks of the current buffer, most recent first.

```
mark-ring
⇒ (#<marker at 11050 in markers.texi>
    #<marker at 10832 in markers.texi>
    ...)
```

mark-ring-max — User Option

The value of this variable is the maximum size of `mark-ring`. If more marks than this are pushed onto the `mark-ring`, `push-mark` discards an old mark when it adds a new one.

In additional to a per-buffer mark ring, there is a *global mark ring*. Marks are pushed onto the global mark ring the first time you set a mark after switching buffers.

global-mark-ring Variable

The value of this variable is the list of saved former global marks, most recent first.

mark-ring-max User Option

The value of this variable is the maximum size of `global-mark-ring`. If more marks than this are pushed onto the `global-mark-ring`, `push-mark` discards an old mark when it adds a new one.

pop-global-mark Command

This function pops a mark off the global mark ring and jumps to that location.

35.7 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.

When `zmacs-regions` is non-`nil` (this is the default), the concept of an *active region* exists. The region is active when the corresponding mark is active. Note that only one active region at a time can exist – i.e. only one buffer's region is active at a time. See Section 35.6 [The Mark], page 504, for more information about active regions.

zmacs-regions User Option

If non-`nil` (the default), active regions are used. See Section 35.6 [The Mark], page 504, for a detailed explanation of what this means.

A number of functions are provided for explicitly determining the bounds of the region and whether it is active. Few programs need to use these functions, however. A command designed to operate on a region should normally use `interactive` with the '`r`' specification to find the beginning and end of the region. This lets other Lisp programs specify the bounds explicitly as arguments and automatically respects the user's setting for *zmacs-regions*. (See Section 19.2.2 [Interactive Codes], page 283.)

region-beginning &optional *buffer* Function

This function returns the position of the beginning of *buffer*'s region (as an integer). This is the position of either point or the mark, whichever is smaller. *buffer* defaults to the current buffer if omitted.

If the mark does not point anywhere, an error is signaled. Note that this function ignores whether the region is active.

region-end &optional *buffer* Function

This function returns the position of the end of *buffer*'s region (as an integer). This is the position of either point or the mark, whichever is larger. *buffer* defaults to the current buffer if omitted.

If the mark does not point anywhere, an error is signaled. Note that this function ignores whether the region is active.

region-exists-p Function

This function is non-`nil` if the region exists. If active regions are in use (i.e. `zmacs-regions` is true), this means that the region is active. Otherwise, this means that the user has pushed a mark in this buffer at some point in the past. If this function returns `nil`, a function that uses the '`r`' interactive specification will cause an error when called interactively.

region-active-p Function

If `zmacs-regions` is true, this is equivalent to `region-exists-p`. Otherwise, this function always returns false. This function is used by commands such as `fill-paragraph-or-region` and `capitalize-region-or-word`, which operate either on the active region or on something else (e.g. the word or paragraph at point).

zmacs-region-stays Variable

If a command sets this variable to true, the currently active region will remain activated when the command finishes. (Normally the region is deactivated when each command terminates.) If *zmacs-regions* is false, however, this has no effect. Under normal circumstances, you do not need to set this; use the interactive specification '`_`' instead, if you want the region to remain active.

zmacs-activate-region Function

This function activates the region in the current buffer (this is equivalent to activating the current buffer's mark). This will normally also highlight the text in the active region and set *zmacs-region-stays* to `t`. (If *zmacs-regions* is false, however, this function has no effect.)

zmacs-deactivate-region Function

This function deactivates the region in the current buffer (this is equivalent to deactivating the current buffer's mark). This will normally also unhighlight the text in the active region and set *zmacs-region-stays* to `nil`. (If *zmacs-regions* is false, however, this function has no effect.)

zmacs-update-region Function

This function updates the active region, if it's currently active. (If there is no active region, this function does nothing.) This has the effect of updating the highlighting on the text in the region; but you should never need to call this except under rather strange circumstances. The command loop automatically calls it when appropriate. Calling this function will call the hook `zmacs-update-region-hook`, if the region is active.

zmacs-activate-region-hook Variable

This normal hook is called when a region becomes active. (Usually this happens as a result of a command that activates the region, such as `set-mark-command`, `activate-region`, or `exchange-point-and-mark`.) Note that calling '`zmacs-activate-region`' will call this hook, even if the region is already active. If *zmacs-regions* is false, however, this hook will never get called under any circumstances.

zmacs-deactivate-region-hook Variable

This normal hook is called when an active region becomes inactive. (Calling '`zmacs-deactivate-region`' when the region is inactive will *not* cause this hook to be called.) If *zmacs-regions* is false, this hook will never get called.

zmacs-update-region-hook Variable

This normal hook is called when an active region is "updated" by `zmacs-update-region`. This normally gets called at the end of each command that sets *zmacs-region-stays* to `t`, indicating that the region should remain activated. The motion commands do this.

36 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 36.9 [Undo], page 523).

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 35 [Markers], page 499) or numeric character positions (see Chapter 34 [Positions], page 487). The order of these arguments does not matter; it is all right for *start* to be the end of the region and *end* the beginning. For example, `(delete-region 1 10)` and `(delete-region 10 1)` are equivalent. An `args-out-of-range` error is signaled if either *start* or *end* is outside the accessible portion of the buffer. In an interactive call, point and the mark are used for these arguments.

Throughout this chapter, "text" refers to the characters in the buffer, together with their properties (when relevant).

36.1 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 37.3 [Regexp Search], page 557.

Many of these functions take an optional *buffer* argument. In all such cases, the current buffer will be used if this argument is omitted. (In FSF Emacs, and earlier versions of XEmacs, these functions usually did not have these optional *buffer* arguments and always operated on the current buffer.)

char-after *position* &optional *buffer* Function

This function returns the character in the 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`. If optional argument *buffer* is `nil`, the current buffer is assumed.

In the following example, assume that the first character in the buffer is '`@`':

```
(char-to-string (char-after 1))
     ⇒ "@"
```

following-char &optional *buffer* Function

This function returns the character following point in the 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. If optional argument *buffer* is `nil`, the current buffer is assumed.

Remember that point is always between characters, and the terminal cursor normally appears over the character following point. Therefore, the character returned by `following-char` is the character the cursor is over.

In this example, point is between the '`a`' and the '`c`'.

```
---------- Buffer: foo ----------
Gentlemen may cry ``Pea⋆ce! Peace!,''
but there is no peace.
---------- Buffer: foo ----------

(char-to-string (preceding-char))
     ⇒ "a"
(char-to-string (following-char))
     ⇒ "c"
```

preceding-char &optional *buffer* Function

This function returns the character preceding point in the buffer. See above, under `following-char`, for an example. If point is at the beginning of the buffer, `preceding-char` returns 0. If optional argument *buffer* is `nil`, the current buffer is assumed.

bobp &optional *buffer* 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. If optional argument *buffer* is `nil`, the current buffer is assumed. See also `point-min` in Section 34.1 [Point], page 487.

eobp &optional *buffer* 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. If optional argument *buffer* is `nil`, the current buffer is assumed. See also `point-max` in See Section 34.1 [Point], page 487.

bolp &optional *buffer* Function

This function returns `t` if point is at the beginning of a line. If optional argument *buffer* is `nil`, the current buffer is assumed. See Section 34.2.4 [Text Lines], page 490. The beginning of the buffer (or its accessible portion) always counts as the beginning of a line.

eolp &optional *buffer* 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. If optional argument *buffer* is `nil`, the current buffer is assumed. The end of the buffer (or of its accessible portion) is always considered the end of a line.

36.2 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* &optional *buffer* Function
buffer-string *start end* &optional *buffer* Function

These functions are equivalent and return a string containing a copy of the text of the region defined by positions *start* and *end* in the buffer. If the arguments are not positions in the accessible portion of the buffer, `buffer-substring` signals an `args-out-of-range` error. If optional argument *buffer* is `nil`, the current buffer is assumed.

If the region delineated by *start* and *end* contains duplicable extents, they will be remembered in the string. See Section 40.9 [Duplicable Extents], page 596.

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
"
```

36.3 Comparing Text

This function lets you compare portions of the text in a buffer, without copying them into strings first.

compare-buffer-substrings *buffer1 start1 end1 buffer2 start2 end2* Function

This function lets you compare two substrings of the same buffer or two different buffers. The first three arguments specify one substring, giving a buffer and two positions within the buffer. The last three arguments specify the other substring in the same way. You can use `nil` for *buffer1*, *buffer2*, or both to stand for the current buffer.

The value is negative if the first substring is less, positive if the first is greater, and zero if they are equal. The absolute value of the result is one plus the index of the first differing characters within the substrings.

This function ignores case when comparing characters if `case-fold-search` is non-`nil`. It always ignores text properties.

Suppose the current buffer contains the text '`foobarbar haha!rara!`'; then in this example the two substrings are '`rbar `' and '`rara!`'. The value is 2 because the first substring is greater at the second character.

```
(compare-buffer-substring nil 6 11 nil 16 21)
     ⇒ 2
```

36.4 Inserting Text

Insertion means adding new text to a buffer. The inserted text goes at point—between the character before point and the character after point.

Insertion relocates markers that point at positions after the insertion point, so that they stay with the surrounding text (see Chapter 35 [Markers], page 499). When a marker points at the place of insertion, insertion normally doesn't relocate the marker, so that it points to the beginning of the inserted text; however, certain special functions such as `insert-before-markers` relocate such markers to point after the inserted text.

Some insertion functions leave point before the inserted text, while other functions leave it after. We call the former insertion *after point* and the latter insertion *before point*.

If a string with non-`nil` extent data is inserted, the remembered extents will also be inserted. See Section 40.9 [Duplicable Extents], page 596.

Insertion functions signal an error if the current buffer is read-only.

These functions copy text characters from strings and buffers along with their properties. The inserted characters have exactly the same properties as the characters they were copied from. By contrast, characters specified as separate arguments, not part of a string or buffer, inherit their text properties from the neighboring text.

insert &rest *args* — Function

This function inserts the strings and/or characters *args* into the current buffer, at point, moving point forward. In other words, it inserts the text before point. An error is signaled unless all *args* are either strings or characters. The value is `nil`.

insert-before-markers &rest *args* — Function

This function inserts the strings and/or characters *args* into the current buffer, at point, moving point forward. An error is signaled unless all *args* are either strings or characters. The value is `nil`.

This function is unlike the other insertion functions in that it relocates markers initially pointing at the insertion point, to point after the inserted text.

insert-string *string* &optional *buffer* — Function

This function inserts *string* into *buffer* before point. *buffer* defaults to the current buffer if omitted. This function is chiefly useful if you want to insert a string in a buffer other than the current one (otherwise you could just use `insert`).

insert-char *character count* &optional *buffer* — Function

This function inserts *count* instances of *character* into *buffer* before point. *count* must be a number, and *character* must be a character. The value is `nil`. If optional argument *buffer* is `nil`, the current buffer is assumed. (In FSF Emacs, the third argument is called *inherit* and refers to text properties.)

insert-buffer-substring *from-buffer-or-name* &optional *start end* — Function

This function inserts a portion of buffer *from-buffer-or-name* (which must already exist) into the current buffer before point. The text inserted is the region from *start* and *end*. (These arguments default to the beginning and end of the accessible portion of that buffer.) This function returns `nil`.

In this example, the form is executed with buffer 'bar' as the current buffer. We assume that buffer 'bar' is initially empty.

```
---------- Buffer: foo ----------
We hold these truths to be self-evident, that all
---------- Buffer: foo ----------

(insert-buffer-substring "foo" 1 20)
     ⇒ nil

---------- Buffer: bar ----------
We hold these truth⋆
---------- Buffer: bar ----------
```

36.5 User-Level Insertion Commands

This section describes higher-level commands for inserting text, commands intended primarily for the user but useful also in Lisp programs.

insert-buffer *from-buffer-or-name* Command

This command inserts the entire contents of *from-buffer-or-name* (which must exist) into the current buffer after point. It leaves the mark after the inserted text. The value is `nil`.

self-insert-command *count* Command

This command inserts the last character typed; it does so *count* times, before point, and returns `nil`. Most printing characters are bound to this command. In routine use, `self-insert-command` is the most frequently called function in XEmacs, but programs rarely use it except to install it on a keymap.

In an interactive call, *count* is the numeric prefix argument.

This command calls `auto-fill-function` whenever that is non-`nil` and the character inserted is a space or a newline (see Section 36.13 [Auto Filling], page 529).

This command performs abbrev expansion if Abbrev mode is enabled and the inserted character does not have word-constituent syntax. (See Chapter 39 [Abbrevs], page 579, and Section 38.2.1 [Syntax Class Table], page 570.)

This is also responsible for calling `blink-paren-function` when the inserted character has close parenthesis syntax (see Section 45.9 [Blinking], page 658).

newline &optional *number-of-newlines* Command

This command inserts newlines into the current buffer before point. If *number-of-newlines* is supplied, that many newline characters are inserted.

This function calls `auto-fill-function` if the current column number is greater than the value of `fill-column` and *number-of-newlines* is `nil`. Typically what `auto-fill-function` does is insert a newline; thus, the overall result in this case is to insert two newlines at different places: one at point, and another earlier in the line. `newline` does not auto-fill if *number-of-newlines* is non-`nil`.

This command indents to the left margin if that is not zero. See Section 36.12 [Margins], page 528.

The value returned is `nil`. In an interactive call, *count* is the numeric prefix argument.

split-line Command

This command splits the current line, moving the portion of the line after point down vertically so that it is on the next line directly below where it was before. Whitespace is inserted as needed at the beginning of the lower line, using the `indent-to` function. `split-line` returns the position of point.

Programs hardly ever use this function.

overwrite-mode Variable

This variable controls whether overwrite mode is in effect: a non-`nil` value enables the mode. It is automatically made buffer-local when set in any fashion.

36.6 Deleting Text

Deletion means removing part of the text in a buffer, without saving it in the kill ring (see Section 36.8 [The Kill Ring], page 519). Deleted text can't be yanked, but can be reinserted using the undo mechanism (see Section 36.9 [Undo], page 523). Some deletion functions do save text in the kill ring in some special cases.

All of the deletion functions operate on the current buffer, and all return a value of `nil`.

erase-buffer &optional *buffer* Function

This function deletes the entire text of *buffer*, leaving it empty. If the buffer is read-only, it signals a `buffer-read-only` error. Otherwise, it deletes the text without asking for any confirmation. It returns `nil`. *buffer* defaults to the current buffer if omitted.

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* &optional *buffer* Command

This command deletes the text in *buffer* in the region defined by *start* and *end*. The value is `nil`. If optional argument *buffer* is `nil`, the current buffer is assumed.

delete-char *count* &optional *killp* Command

This command deletes *count* characters directly after point, or before point if *count* is negative. If *killp* is non-`nil`, then it saves the deleted characters in the kill ring.

In an interactive call, *count* is the numeric prefix argument, and *killp* is the unprocessed prefix argument. Therefore, if a prefix argument is supplied, the text is saved in the kill ring. If no prefix argument is supplied, then one character is deleted, but not saved in the kill ring.

The value returned is always `nil`.

delete-backward-char *count* &optional *killp* Command

This command deletes *count* characters directly before point, or after point if *count* is negative. If *killp* is non-`nil`, then it saves the deleted characters in the kill ring.

In an interactive call, *count* is the numeric prefix argument, and *killp* is the unprocessed prefix argument. Therefore, if a prefix argument is supplied, the text is saved in the kill ring. If no prefix argument is supplied, then one character is deleted, but not saved in the kill ring.

The value returned is always `nil`.

backward-delete-char-untabify *count* &optional *killp* Command

This command deletes *count* characters backward, changing tabs into spaces. When the next character to be deleted is a tab, it is first replaced with the proper number of spaces to preserve alignment and then one of those spaces is deleted instead of the tab. If *killp* is non-`nil`, then the command saves the deleted characters in the kill ring.

Conversion of tabs to spaces happens only if *count* is positive. If it is negative, exactly −*count* characters after point are deleted.

In an interactive call, *count* is the numeric prefix argument, and *killp* is the unprocessed prefix argument. Therefore, if a prefix argument is supplied, the text is saved in the kill ring. If no prefix argument is supplied, then one character is deleted, but not saved in the kill ring.

The value returned is always `nil`.

36.7 User-Level Deletion Commands

This section describes higher-level commands for deleting text, commands intended primarily for the user but useful also in Lisp programs.

delete-horizontal-space Command

This function deletes all spaces and tabs around point. It returns `nil`.

In the following examples, we call `delete-horizontal-space` four times, once on each line, with point between the second and third characters on the line each time.

```
---------- Buffer: foo ----------
I ⋆thought
I ⋆     thought
We⋆ thought
Yo⋆u thought
---------- Buffer: foo ----------
```

```
(delete-horizontal-space)   ; Four times.
     ⇒ nil

---------- Buffer: foo ----------
Ithought
Ithought
Wethought
You thought
---------- Buffer: foo ----------
```

delete-indentation &optional *join-following-p* Command

This function joins the line point is on to the previous line, deleting any whitespace at the join and in some cases replacing it with one space. If *join-following-p* is non-`nil`, `delete-indentation` joins this line to the following line instead. The 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. See Section 36.12 [Margins], page 528.

In the example below, point is located on the line starting '`events`', and it makes no difference if there are trailing spaces in the preceding line.

```
---------- Buffer: foo ----------
When in the course of human
⋆    events, it becomes necessary
---------- Buffer: foo ----------

(delete-indentation)
     ⇒ nil

---------- Buffer: foo ----------
When in the course of human⋆ events, it becomes necessary
---------- Buffer: foo ----------
```

After the lines are joined, the function `fixup-whitespace` is responsible for deciding whether to leave a space at the junction.

fixup-whitespace Function

This function replaces all the white space surrounding point with either one space or no space, according to the context. It returns `nil`.

At the beginning or end of a line, the appropriate amount of space is none. Before a character with close parenthesis syntax, or after a character with open parenthesis or expression-prefix syntax, no space is also appropriate. Otherwise, one space is appropriate. See Section 38.2.1 [Syntax Class Table], page 570.

In the example below, `fixup-whitespace` is called the first time with point before the word '`spaces`' in the first line. For the second invocation, point is directly after the '`(`'.

```
---------- Buffer: foo ----------
This has too many     ⋆spaces
This has too many spaces at the start of (⋆   this list)
---------- Buffer: foo ----------
```

```
(fixup-whitespace)
     ⇒ nil
(fixup-whitespace)
     ⇒ nil
---------- Buffer: foo ----------
This has too many spaces
This has too many spaces at the start of (this list)
---------- Buffer: foo ----------
```

just-one-space Command

This command replaces any spaces and tabs around point with a single space. It returns `nil`.

delete-blank-lines Command

This function deletes blank lines surrounding point. If point is on a blank line with one or more blank lines before or after it, then all but one of them are deleted. If point is on an isolated blank line, then it is deleted. If point is on a nonblank line, the command deletes all blank lines following it.

A blank line is defined as a line containing only tabs and spaces.

`delete-blank-lines` returns `nil`.

36.8 The Kill Ring

Kill functions delete text like the deletion functions, but save it so that the user can reinsert it by *yanking*. Most of these functions have '`kill-`' in their name. By contrast, the functions whose names start with '`delete-`' normally do not save text for yanking (though they can still be undone); these are "deletion" functions.

Most of the kill commands are primarily for interactive use, and are not described here. What we do describe are the functions provided for use in writing such commands. You can use these functions to write commands for killing text. When you need to delete text for internal purposes within a Lisp function, you should normally use deletion functions, so as not to disturb the kill ring contents. See Section 36.6 [Deletion], page 516.

Killed text is saved for later yanking in the *kill ring*. This is a list that holds a number of recent kills, not just the last text kill. We call this a "ring" because yanking treats it as having elements in a cyclic order. The list is kept in the variable `kill-ring`, and can be operated on with the usual functions for lists; there are also specialized functions, described in this section, that treat it as a ring.

Some people think this use of the word "kill" is unfortunate, since it refers to operations that specifically *do not* destroy the entities "killed". This is in sharp contrast to ordinary life, in which death is permanent and "killed" entities do not come back to life. Therefore, other metaphors have been proposed. For example, the term "cut ring" makes sense to people who, in pre-computer days, used scissors and paste to cut up and rearrange manuscripts. However, it would be difficult to change the terminology now.

36.8.1 Kill Ring Concepts

The kill ring records killed text as strings in a list, most recent first. A short kill ring, for example, might look like this:

```
("some text" "a different piece of text" "even older text")
```

When the list reaches `kill-ring-max` entries in length, adding a new entry automatically deletes the last entry.

When kill commands are interwoven with other commands, each kill command makes a new entry in the kill ring. Multiple kill commands in succession build up a single entry in the kill ring, which would be yanked as a unit; the second and subsequent consecutive kill commands add text to the entry made by the first one.

For yanking, one entry in the kill ring is designated the "front" of the ring. Some yank commands "rotate" the ring by designating a different element as the "front." But this virtual rotation doesn't change the list itself—the most recent entry always comes first in the list.

36.8.2 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 determine whether the previous command was a kill command, and if so appends the killed text to the most recent entry.

kill-region *start end* Command

This function kills the text in the region defined by *start* and *end*. The text is deleted but saved in the kill ring, along with its text properties. The value is always `nil`.

In an interactive call, *start* and *end* are point and the mark.

If the buffer is read-only, `kill-region` modifies the kill ring just the same, then signals an error without modifying the buffer. This is convenient because it lets the user use all the kill commands to copy text into the kill ring from a read-only buffer.

copy-region-as-kill *start end* Command

This command saves the region defined by *start* and *end* on the kill ring (including text properties), but does not delete the text from the buffer. It returns `nil`. It also indicates the extent of the text copied by moving the cursor momentarily, or by displaying a message in the echo area.

The command does not set `this-command` to `kill-region`, so a subsequent kill command does not append to the same kill ring entry.

Don't call `copy-region-as-kill` in Lisp programs unless you aim to support Emacs 18. For Emacs 19, it is better to use `kill-new` or `kill-append` instead. See Section 36.8.4 [Low-Level Kill Ring], page 521.

36.8.3 Functions for Yanking

Yanking means reinserting an entry of previously killed text from the kill ring. The text properties are copied too.

yank &optional *arg* Command

This command inserts before point the text in the first entry in the kill ring. It positions the mark at the beginning of that text, and point at the end.

If *arg* is a list (which occurs interactively when the user types `C-u` with no digits), then `yank` inserts the text as described above, but puts point before the yanked text and puts the mark after it.

If *arg* is a number, then `yank` inserts the *arg*th most recently killed text—the *arg*th element of the kill ring list.

`yank` does not alter the contents of the kill ring or rotate it. It returns `nil`.

yank-pop *arg* Command

This command replaces the just-yanked entry from the kill ring with a different entry from the kill ring.

This is allowed only immediately after a `yank` or another `yank-pop`. At such a time, the region contains text that was just inserted by yanking. `yank-pop` deletes that text and inserts in its place a different piece of killed text. It does not add the deleted text to the kill ring, since it is already in the kill ring somewhere.

If *arg* is `nil`, then the replacement text is the previous element of the kill ring. If *arg* is numeric, the replacement is the *arg*th previous kill. If *arg* is negative, a more recent kill is the replacement.

The sequence of kills in the kill ring wraps around, so that after the oldest one comes the newest one, and before the newest one goes the oldest.

The value is always `nil`.

36.8.4 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 which designates the "front" of the kill ring by *n* places (from newer kills to older ones), and returns the text at that place in the ring.

If the optional second argument *do-not-move* is non-`nil`, then `current-kill` doesn't alter the yanking pointer; it just returns the *n*th kill, counting from the current yanking pointer.

If *n* is zero, indicating a request for the latest kill, `current-kill` calls the value of `interprogram-paste-function` (documented below) before consulting the kill ring.

kill-new *string* Function

This function puts the text *string* into the kill ring as a new entry at the front of the ring. It discards the oldest entry if appropriate. It also invokes the value of `interprogram-cut-function` (see below).

kill-append *string before-p* Function

This function appends the text *string* to the first entry in the kill ring. Normally *string* goes at the end of the entry, but if *before-p* is non-`nil`, it goes at the beginning. This function also invokes the value of `interprogram-cut-function` (see below).

interprogram-paste-function Variable

This variable provides a way of transferring killed text from other programs, when you are using a window system. Its value should be `nil` or a function of no arguments.

If the value is a function, `current-kill` calls it to get the "most recent kill". If the function returns a non-`nil` value, then that value is used as the "most recent kill". If it returns `nil`, then the first element of `kill-ring` is used.

The normal use of this hook is to get the X server's primary selection as the most recent kill, even if the selection belongs to another X client. See Section 51.1 [X Selections], page 713.

interprogram-cut-function Variable

This variable provides a way of communicating killed text to other programs, when you are using a window system. Its value should be `nil` or a function of one argument.

If the value is a function, `kill-new` and `kill-append` call it with the new first element of the kill ring as an argument.

The normal use of this hook is to set the X server's primary selection to the newly killed text.

36.8.5 Internals of the Kill Ring

The variable `kill-ring` holds the kill ring contents, in the form of a list of strings. The most recent kill is always at the front of the list.

The `kill-ring-yank-pointer` variable points to a link in the kill ring list, whose CAR is the text to yank next. We say it identifies the "front" of the ring. Moving `kill-ring-yank-pointer` to a different link is called *rotating the kill ring*. We call the kill ring a "ring" because the functions that move the yank pointer wrap around from the end of the list to the beginning, or vice-versa. Rotation of the kill ring is virtual; it does not change the value of `kill-ring`.

Both `kill-ring` and `kill-ring-yank-pointer` are Lisp variables whose values are normally lists. The word "pointer" in the name of the `kill-ring-yank-pointer` indicates that the variable's purpose is to identify one element of the list for use by the next yank command.

The value of `kill-ring-yank-pointer` is always `eq` to one of the links in the kill ring list. The element it identifies is the CAR of that link. Kill commands, which change the kill ring, also set this variable to the value of `kill-ring`. The effect is to rotate the ring so that the newly killed text is at the front.

Here is a diagram that shows the variable `kill-ring-yank-pointer` pointing to the second entry in the kill ring `("some text" "a different piece of text" "yet older text")`.

```
kill-ring       kill-ring-yank-pointer
  |               |
  |     ___ ___    --->  ___ ___      ___ ___
   --> |___|___|------> |___|___|--> |___|___|--> nil
         |               |            |
         |               |            |
         |               |             -->"yet older text"
         |               |
         |                --> "a different piece of text"
         |
          --> "some text"
```

This state of affairs might occur after *C-y* (`yank`) immediately followed by *M-y* (`yank-pop`).

kill-ring Variable

This variable holds the list of killed text sequences, most recently killed first.

kill-ring-yank-pointer Variable

This variable's value indicates which element of the kill ring is at the "front" of the ring for yanking. More precisely, the value is a tail of the value of `kill-ring`, and its CAR is the kill string that *C-y* should yank.

kill-ring-max User Option

The value of this variable is the maximum length to which the kill ring can grow, before elements are thrown away at the end. The default value for `kill-ring-max` is 30.

36.9 Undo

Most buffers have an *undo list*, which records all changes made to the buffer's text so that they can be undone. (The buffers that don't have one are usually special-purpose buffers for which XEmacs assumes that undoing is not useful.) All the primitives that modify the text in the buffer automatically add elements to the front of the undo list, which is in the variable `buffer-undo-list`.

buffer-undo-list Variable

This variable's value is the undo list of the current buffer. A value of `t` disables the recording of undo information.

Here are the kinds of elements an undo list can have:

integer This kind of element records a previous value of point. Ordinary cursor motion does not get any sort of undo record, but deletion commands use these entries to record where point was before the command.

(*beg* . *end*)
: This kind of element indicates how to delete text that was inserted. Upon insertion, the text occupied the range *beg*–*end* in the buffer.

(*text* . *position*)
: This kind of element indicates how to reinsert text that was deleted. The deleted text itself is the string *text*. The place to reinsert it is `(abs` *position*`)`.

(`t` *high* . *low*)
: This kind of element indicates that an unmodified buffer became modified. The elements *high* and *low* are two integers, each recording 16 bits of the visited file's modification time as of when it was previously visited or saved. `primitive-undo` uses those values to determine whether to mark the buffer as unmodified once again; it does so only if the file's modification time matches those numbers.

(`nil` *property value beg* . *end*)
: This kind of element records a change in a text property. Here's how you might undo the change:

```
(put-text-property beg end property value)
```

position
: This element indicates where point was at an earlier time. Undoing this element sets point to *position*. Deletion normally creates an element of this kind as well as a reinsertion element.

`nil`
: This element is a boundary. The elements between two boundaries are called a *change group*; normally, each change group corresponds to one keyboard command, and undo commands normally undo an entire group as a unit.

undo-boundary Function

This function places a boundary element in the undo list. The undo command stops at such a boundary, and successive undo commands undo to earlier and earlier boundaries. This function returns `nil`.

The editor command loop automatically creates an undo boundary before each key sequence is executed. Thus, each undo normally undoes the effects of one command. Self-inserting input characters are an exception. The command loop makes a boundary for the first such character; the next 19 consecutive self-inserting input characters do not make boundaries, and then the 20th does, and so on as long as self-inserting characters continue.

All buffer modifications add a boundary whenever the previous undoable change was made in some other buffer. This way, a command that modifies several buffers makes a boundary in each buffer it changes.

Calling this function explicitly is useful for splitting the effects of a command into more than one unit. For example, `query-replace` calls `undo-boundary` after each replacement, so that the user can undo individual replacements one by one.

primitive-undo *count list* Function

This is the basic function for undoing elements of an undo list. It undoes the first *count* elements of *list*, returning the rest of *list*. You could write this function in Lisp, but it is convenient to have it in C.

`primitive-undo` adds elements to the buffer's undo list when it changes the buffer. Undo commands avoid confusion by saving the undo list value at the beginning of a sequence of undo operations. Then the undo operations use and update the saved value. The new elements added by undoing are not part of this saved value, so they don't interfere with continuing to undo.

36.10 Maintaining Undo Lists

This section describes how to enable and disable undo information for a given buffer. It also explains how the undo list is truncated automatically so it doesn't get too big.

Recording of undo information in a newly created buffer is normally enabled to start with; but if the buffer name starts with a space, the undo recording is initially disabled. You can explicitly enable or disable undo recording with the following two functions, or by setting `buffer-undo-list` yourself.

buffer-enable-undo &optional *buffer-or-name* Command

This command enables recording undo information for buffer *buffer-or-name*, so that subsequent changes can be undone. If no argument is supplied, then the current buffer is used. This function does nothing if undo recording is already enabled in the buffer. It returns `nil`.

In an interactive call, *buffer-or-name* is the current buffer. You cannot specify any other buffer.

buffer-disable-undo &optional *buffer* Function
buffer-flush-undo &optional *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` is new as of Emacs versions 19.

As editing continues, undo lists get longer and longer. To prevent them from using up all available memory space, garbage collection trims them back to size limits you can set. (For this purpose, the "size" of an undo list measures the cons cells that make up the list, plus the strings of deleted text.) Two variables control the range of acceptable sizes: `undo-limit` and `undo-strong-limit`.

undo-limit Variable

This is the soft limit for the acceptable size of an undo list. The change group at which this size is exceeded is the last one kept.

undo-strong-limit Variable

This is the upper limit for the acceptable size of an undo list. The change group at which this size is exceeded is discarded itself (along with all older change groups). There is one exception: the very latest change group is never discarded no matter how big it is.

36.11 Filling

Filling means adjusting the lengths of lines (by moving the line breaks) so that they are nearly (but no greater than) a specified maximum width. Additionally, lines can be *justified*, which means inserting spaces to make the left and/or right margins line up precisely. The width is controlled by the variable `fill-column`. For ease of reading, lines should be no longer than 70 or so columns.

You can use Auto Fill mode (see Section 36.13 [Auto Filling], page 529) to fill text automatically as you insert it, but changes to existing text may leave it improperly filled. Then you must fill the text explicitly.

Most of the commands in this section return values that are not meaningful. All the functions that do filling take note of the current left margin, current right margin, and current justification style (see Section 36.12 [Margins], page 528). If the current justification style is `none`, the filling functions don't actually do anything.

Several of the filling functions have an argument *justify*. If it is non-`nil`, that requests some kind of justification. It can be `left`, `right`, `full`, or `center`, to request a specific style of justification. If it is `t`, that means to use the current justification style for this part of the text (see `current-justification`, below).

When you call the filling functions interactively, using a prefix argument implies the value `full` for *justify*.

fill-paragraph *justify* Command

This command fills the paragraph at or after point. If *justify* is non-`nil`, each line is justified as well. It uses the ordinary paragraph motion commands to find paragraph boundaries. See section "Paragraphs" in *The XEmacs User's Manual*.

fill-region *start end* &optional *justify* Command

This command fills each of the paragraphs in the region from *start* to *end*. It justifies as well if *justify* is non-`nil`.

The variable `paragraph-separate` controls how to distinguish paragraphs. See Section 37.8 [Standard Regexps], page 566.

fill-individual-paragraphs *start end* &optional *justify mail-flag* Command

This command fills each paragraph in the region according to its individual fill prefix. Thus, if the lines of a paragraph were indented with spaces, the filled paragraph will remain indented in the same fashion.

The first two arguments, *start* and *end*, are the beginning and end of the region to be filled. The third and fourth arguments, *justify* and *mail-flag*, are optional. If *justify* is non-`nil`, the paragraphs are justified as well as filled. If *mail-flag* is non-`nil`, it means the function is operating on a mail message and therefore should not fill the header lines.

Ordinarily, `fill-individual-paragraphs` regards each change in indentation as starting a new paragraph. If `fill-individual-varying-indent` is non-`nil`, then only separator lines separate paragraphs. That mode can handle indented paragraphs with additional indentation on the first line.

fill-individual-varying-indent User Option

This variable alters the action of `fill-individual-paragraphs` as described above.

fill-region-as-paragraph *start end* &optional *justify* Command

This command 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* 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 *how eop nosqueeze* Command

This command inserts spaces between the words of the current line so that the line ends exactly at `fill-column`. It returns `nil`.

The argument *how*, if non-`nil` specifies explicitly the style of justification. It can be `left`, `right`, `full`, `center`, or `none`. If it is `t`, that means to do follow specified justification style (see `current-justification`, below). `nil` means to do full justification.

If *eop* is non-`nil`, that means do left-justification when `current-justification` specifies full justification. This is used for the last line of a paragraph; even if the paragraph as a whole is fully justified, the last line should not be.

If *nosqueeze* is non-`nil`, that means do not change interior whitespace.

default-justification User Option

This variable's value specifies the style of justification to use for text that doesn't specify a style with a text property. The possible values are `left`, `right`, `full`, `center`, or `none`. The default value is `left`.

current-justification Function

This function returns the proper justification style to use for filling the text around point.

fill-paragraph-function Variable

This variable provides a way for major modes to override the filling of paragraphs. If the value is non-`nil`, `fill-paragraph` calls this function to do the work. If the function returns a non-`nil` value, `fill-paragraph` assumes the job is done, and immediately returns that value.

The usual use of this feature is to fill comments in programming language modes. If the function needs to fill a paragraph in the usual way, it can do so as follows:

```
(let ((fill-paragraph-function nil))
  (fill-paragraph arg))
```

use-hard-newlines Variable

If this variable is non-`nil`, the filling functions do not delete newlines that have the `hard` text property. These "hard newlines" act as paragraph separators.

36.12 Margins for Filling

fill-prefix User Option

This variable specifies a string of text that appears at the beginning of normal text lines and should be disregarded when filling them. Any line that fails to start with the fill prefix is considered the start of a paragraph; so is any line that starts with the fill prefix followed by additional whitespace. Lines that start with the fill prefix but no additional whitespace are ordinary text lines that can be filled together. The resulting filled lines also start with the fill prefix.

The fill prefix follows the left margin whitespace, if any.

fill-column User Option

This buffer-local variable specifies the maximum width of filled lines. Its value should be an integer, which is a number of columns. All the filling, justification and centering commands are affected by this variable, including Auto Fill mode (see Section 36.13 [Auto Filling], page 529).

As a practical matter, if you are writing text for other people to read, you should set `fill-column` to no more than 70. Otherwise the line will be too long for people to read comfortably, and this can make the text seem clumsy.

default-fill-column Variable

The value of this variable is the default value for `fill-column` in buffers that do not override it. This is the same as `(default-value 'fill-column)`.

The default value for `default-fill-column` is 70.

set-left-margin *from to margin* Command

This sets the `left-margin` property on the text from *from* to *to* to the value *margin*. If Auto Fill mode is enabled, this command also refills the region to fit the new margin.

set-right-margin *from to margin* Command

This sets the `right-margin` property on the text from *from* to *to* to the value *margin*. If Auto Fill mode is enabled, this command also refills the region to fit the new margin.

current-left-margin Function

This function returns the proper left margin value to use for filling the text around point. The value is the sum of the `left-margin` property of the character at the start of the current line (or zero if none), and the value of the variable `left-margin`.

current-fill-column Function

This function returns the proper fill column value to use for filling the text around point. The value is the value of the `fill-column` variable, minus the value of the `right-margin` property of the character after point.

move-to-left-margin &optional *n force* Command

This function moves point to the left margin of the current line. The column moved to is determined by calling the function `current-left-margin`. If the argument *n* is non-`nil`, `move-to-left-margin` moves forward $n-1$ lines first.

If *force* is non-`nil`, that says to fix the line's indentation if that doesn't match the left margin value.

delete-to-left-margin *from to* Function

This function removes left margin indentation from the text between *from* and *to*. The amount of indentation to delete is determined by calling `current-left-margin`. In no case does this function delete non-whitespace.

indent-to-left-margin Function

This is the default `indent-line-function`, used in Fundamental mode, Text mode, etc. Its effect is to adjust the indentation at the beginning of the current line to the value specified by the variable `left-margin`. This may involve either inserting or deleting whitespace.

left-margin Variable

This variable specifies the base left margin column. In Fundamental mode, ⟨LFD⟩ indents to this column. This variable automatically becomes buffer-local when set in any fashion.

36.13 Auto Filling

Auto Fill mode is a minor mode that fills lines automatically as text is inserted. This section describes the hook used by Auto Fill mode. For a description of functions that you can call explicitly to fill and justify existing text, see Section 36.11 [Filling], page 526.

Auto Fill mode also enables the functions that change the margins and justification style to refill portions of the text. See Section 36.12 [Margins], page 528.

auto-fill-function Variable

The value of this variable should be a function (of no arguments) to be called after self-inserting a space or a newline. It may be `nil`, in which case nothing special is done in that case.

The value of `auto-fill-function` is `do-auto-fill` when Auto-Fill mode is enabled. That is a function whose sole purpose is to implement the usual strategy for breaking a line.

> In older Emacs versions, this variable was named `auto-fill-hook`, but since it is not called with the standard convention for hooks, it was renamed to `auto-fill-function` in version 19.

36.14 Sorting Text

The sorting functions described in this section all rearrange text in a buffer. This is in contrast to the function `sort`, which rearranges the order of the elements of a list (see Section 5.6.3 [Rearrangement], page 87). The values returned by these functions are not meaningful.

sort-subr *reverse nextrecfun endrecfun* &optional *startkeyfun endkeyfun* Function

This function is the general text-sorting routine that divides a buffer into records and sorts them. Most of the commands in this section use this function.

To understand how `sort-subr` works, consider the whole accessible portion of the buffer as being divided into disjoint pieces called *sort records*. The records may or may not be contiguous; they may not overlap. A portion of each sort record (perhaps all of it) is designated as the sort key. Sorting rearranges the records in order by their sort keys.

Usually, the records are rearranged in order of ascending sort key. If the first argument to the `sort-subr` function, *reverse*, is non-`nil`, the sort records are rearranged in order of descending sort key.

The next four arguments to `sort-subr` are functions that are called to move point across a sort record. They are called many times from within `sort-subr`.

1. *nextrecfun* is called with point at the end of a record. This function moves point to the start of the next record. The first record is assumed to start at the position of point when `sort-subr` is called. Therefore, you should usually move point to the beginning of the buffer before calling `sort-subr`.

   This function can indicate there are no more sort records by leaving point at the end of the buffer.

2. *endrecfun* is called with point within a record. It moves point to the end of the record.

3. *startkeyfun* is called to move point from the start of a record to the start of the sort key. This argument is optional; if it is omitted, the whole record is the sort key. If supplied, the function should either return a non-`nil` value to be used as the sort key, or return `nil` to indicate that the sort key is in the buffer starting at point. In the latter case, *endkeyfun* is called to find the end of the sort key.

4. *endkeyfun* is called to move point from the start of the sort key to the end of the sort key. This argument is optional. If *startkeyfun* returns `nil` and this argument is omitted (or `nil`), then the sort key extends to the end of the record. There is no need for *endkeyfun* if *startkeyfun* returns a non-`nil` value.

As an example of `sort-subr`, here is the complete function definition for `sort-lines`:

```
;; Note that the first two lines of doc string
;; are effectively one line when viewed by a user.
(defun sort-lines (reverse beg end)
  "Sort lines in region alphabetically.
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-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 how to divide the buffer into sort records. At the end of each record, a search is done for this regular expression, and the text that matches it is 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 37.2 [Regular Expressions], page 550, for a description of the syntax and meaning of regular expressions.

The value of the *key-regexp* argument specifies what part of each record is the sort key. The *key-regexp* could match the whole record, or only a part. In the latter case, the rest of the record has no effect on the sorted order of records, but it is carried along when the record moves to its new position.

The *key-regexp* argument can refer to the text matched by a subexpression of *record-regexp*, or it can be a regular expression on its own.

If *key-regexp* is:

'`\`*digit*' then the text matched by the *digit*th '`\(...\)`' parenthesis grouping in *record-regexp* is the sort key.

'\&' then the whole record is the sort key.

a regular expression
: then sort-regexp-fields searches for a match for the regular expression within the record. If such a match is found, it is the sort key. If there is no match for *key-regexp* within a record then that record is ignored, which means its position in the buffer is not changed. (The other records may move around it.)

For example, if you plan to sort all the lines in the region by the first word on each line starting with the letter 'f', you should set *record-regexp* to '^.*$' and set *key-regexp* to '\<f\w*\>'. The resulting expression looks like this:

```
(sort-regexp-fields nil "^.*$" "\\<f\\w*\\>"
                    (region-beginning)
                    (region-end))
```

If you call sort-regexp-fields interactively, it prompts for *record-regexp* and *key-regexp* in the minibuffer.

sort-lines *reverse start end* Command
: This command alphabetically sorts lines in the region between *start* and *end*. If *reverse* is non-nil, the sort is in reverse order.

sort-paragraphs *reverse start end* Command
: This command alphabetically sorts paragraphs in the region between *start* and *end*. If *reverse* is non-nil, the sort is in reverse order.

sort-pages *reverse start end* Command
: This command alphabetically sorts pages in the region between *start* and *end*. If *reverse* is non-nil, the sort is in reverse order.

sort-fields *field start end* Command
: This command sorts lines in the region between *start* and *end*, comparing them alphabetically by the *field*th field of each line. Fields are separated by whitespace and numbered starting from 1. If *field* is negative, sorting is by the −*field*th field from the end of the line. This command is useful for sorting tables.

sort-numeric-fields *field start end* Command
: This command sorts lines in the region between *start* and *end*, comparing them numerically by the *field*th field of each line. The specified field must contain a number in each line of the region. Fields are separated by whitespace and numbered starting from 1. If *field* is negative, sorting is by the −*field*th field from the end of the line. This command is useful for sorting tables.

sort-columns *reverse* &optional *beg end* Command
: This command sorts the lines in the region between *beg* and *end*, comparing them alphabetically by a certain range of columns. The column positions of *beg* and *end* bound the range of columns to sort on.

If *reverse* is non-nil, the sort is in reverse order.

One unusual thing about this command is that the entire line containing position *beg*, and the entire line containing position *end*, are included in the region sorted.

Note that `sort-columns` uses the `sort` utility program, and so cannot work properly on text containing tab characters. Use `M-x untabify` to convert tabs to spaces before sorting.

36.15 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).

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 45.10 [Usual Display], page 658.

Column number computations ignore the width of the window and the amount of horizontal scrolling. Consequently, a column value can be arbitrarily high. The first (or leftmost) column is numbered 0.

current-column Function

This function returns the horizontal position of point, measured in columns, counting from 0 at the left margin. The column position is the sum of the widths of all the displayed representations of the characters between the start of the current line and point.

For an example of using `current-column`, see the description of `count-lines` in Section 34.2.4 [Text Lines], page 490.

move-to-column *column* &optional *force* Function

This function moves point to *column* in the current line. The calculation of *column* takes into account the widths of the displayed representations of the characters between the start of the line and point.

If column *column* is beyond the end of the line, point moves to the end of the line. If *column* is negative, point moves to the beginning of the line.

If it is impossible to move to column *column* because that is in the middle of a multicolumn character such as a tab, point moves to the end of that character. However, if *force* is non-`nil`, and *column* is in the middle of a tab, then `move-to-column` converts the tab into spaces so that it can move precisely to column *column*. Other multicolumn characters can cause anomalies despite *force*, since there is no way to split them.

The argument *force* also has an effect if the line isn't long enough to reach column *column*; in that case, it says to add whitespace at the end of the line to reach that column.

If *column* is not an integer, an error is signaled.

The return value is the column number actually moved to.

36.16 Indentation

The indentation functions are used to examine, move to, and change whitespace that is at the beginning of a line. Some of the functions can also change whitespace elsewhere on a line. Columns and indentation count from zero at the left margin.

36.16.1 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*. Otherwise the function does nothing if point is already 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.

36.16.2 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`, it calls `insert-tab` instead. (That is a trivial command that inserts a tab character.)

newline-and-indent Command

This function inserts a newline, then indents the new line (the one following the newline just inserted) according to the major mode.

It does indentation by calling the current `indent-line-function`. In programming language modes, this is the same thing ⟨TAB⟩ does, but in some text modes, where ⟨TAB⟩ inserts a tab, `newline-and-indent` indents to the column specified by `left-margin`.

reindent-then-newline-and-indent Command

This command reindents the current line, inserts a newline at point, and then reindents the new line (the one following the newline just inserted).

This command does indentation on both lines according to the current major mode, by calling the current value of `indent-line-function`. In programming language modes, this is the same thing ⟨TAB⟩ does, but in some text modes, where ⟨TAB⟩ inserts a tab, `reindent-then-newline-and-indent` indents to the column specified by `left-margin`.

36.16.3 Indenting an Entire Region

This section describes commands that indent all the lines in the region. They return unpredictable values.

indent-region *start end to-column* Command

This command indents each nonblank line starting between *start* (inclusive) and *end* (exclusive). If *to-column* is `nil`, `indent-region` indents each nonblank line by calling the current mode's indentation function, the value of `indent-line-function`.

If *to-column* is non-`nil`, it should be an integer specifying the number of columns of indentation; then this function gives each line exactly that much indentation, by either adding or deleting whitespace.

If there is a fill prefix, `indent-region` indents each line by making it start with the fill prefix.

indent-region-function Variable

The value of this variable is a function that can be used by `indent-region` as a short cut. You should design the function so that it will produce the same results as indenting the lines of the region one by one, but presumably faster.

If the value is `nil`, there is no short cut, and `indent-region` actually works line by line.

A short-cut function is useful in modes such as C mode and Lisp mode, where the `indent-line-function` must scan from the beginning of the function definition: applying it to each line would be quadratic in time. The short cut can update the scan information as it moves through the lines indenting them; this takes linear time. In a mode where indenting a line individually is fast, there is no need for a short cut.

`indent-region` with a non-`nil` argument *to-column* has a different meaning and does not use this variable.

indent-rigidly *start end count* Command

This command indents all lines starting between *start* (inclusive) and *end* (exclusive) sideways by *count* columns. This "preserves the shape" of the affected region, moving it as a rigid unit. Consequently, this command is useful not only for indenting regions of unindented text, but also for indenting regions of formatted code.

For example, if *count* is 3, this command adds 3 columns of indentation to each of the lines beginning in the region specified.

In Mail mode, *C-c C-y* (`mail-yank-original`) uses `indent-rigidly` to indent the text copied from the message being replied to.

indent-code-rigidly *start end columns* &optional *nochange-regexp* Function

This is like `indent-rigidly`, except that it doesn't alter lines that start within strings or comments.

In addition, it doesn't alter a line if *nochange-regexp* matches at the beginning of the line (if *nochange-regexp* is non-`nil`).

36.16.4 Indentation Relative to Previous Lines

This section describes two commands that indent the current line based on the contents of previous lines.

indent-relative &optional *unindented-ok* Command

This command inserts whitespace at point, extending to the same column as the next *indent point* of the previous nonblank line. An indent point is a non-whitespace character following whitespace. The next indent point is the first one at a column greater than the current column of point. For example, if point is underneath and to the left of the first non-blank character of a line of text, it moves to that column by inserting whitespace.

If the previous nonblank line has no next indent point (i.e., none at a great enough column position), `indent-relative` either does nothing (if *unindented-ok* is non-`nil`) or calls `tab-to-tab-stop`. Thus, if point is underneath and to the right of the last column of a short line of text, this command ordinarily moves point to the next tab stop by inserting whitespace.

The return value of `indent-relative` is unpredictable.

In the following example, point is at the beginning of the second line:

```
            This line is indented twelve spaces.
    ⋆The quick brown fox jumped.
```

Evaluation of the expression (indent-relative nil) produces the following:

```
            This line is indented twelve spaces.
            ⋆The quick brown fox jumped.
```

In this 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. It calls `indent-relative` with `t` as the *unindented-ok* argument. The return value is unpredictable.

If the previous nonblank line has no indent points beyond the current column, this command does nothing.

36.16.5 Adjustable "Tab Stops"

This section explains the mechanism for user-specified "tab stops" and the mechanisms that use and set them. The name "tab stops" is used because the feature is similar to that of the tab stops on a typewriter. The feature works by inserting an appropriate number of spaces and tab characters to reach the next tab stop column; it does not affect the display of tab characters in the buffer (see Section 45.10 [Usual Display], page 658). Note that the ⟨TAB⟩ character as input uses this tab stop feature only in a few major modes, such as Text mode.

tab-to-tab-stop Command

This command inserts spaces or tabs up to the next tab stop column defined by `tab-stop-list`. It searches the list for an element greater than the current column number, and uses that element as the column to indent to. It does nothing if no such element is found.

tab-stop-list User Option

This variable is the list of tab stop columns used by `tab-to-tab-stops`. The elements should be integers in increasing order. The tab stop columns need not be evenly spaced.

Use *M-x edit-tab-stops* to edit the location of tab stops interactively.

36.16.6 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`.

36.17 Case Changes

The case change commands described here work on text in the current buffer. See Section 4.11 [Character Case], page 70, for case conversion commands that work on strings and characters. See Section 4.12 [Case Tables], page 71, for how to customize which characters are upper or lower case and how to convert them.

capitalize-region *start end* Command

This function capitalizes all words in the region defined by *start* and *end*. To capitalize means to convert each word's first character to upper case and convert the rest of each word to lower case. The function returns `nil`.

If one end of the region is in the middle of a word, the part of the word within the region is treated as an entire word.

When `capitalize-region` is called interactively, *start* and *end* are point and the mark, with the smallest first.

```
---------- Buffer: foo ----------
This is the contents of the 5th foo.
---------- Buffer: foo ----------

(capitalize-region 1 44)
⇒ nil

---------- Buffer: foo ----------
This Is The Contents Of The 5th Foo.
---------- Buffer: foo ----------
```

downcase-region *start end* Command

This function converts all of the letters in the region defined by *start* and *end* to lower case. The function returns `nil`.

When `downcase-region` is called interactively, *start* and *end* are point and the mark, with the smallest first.

upcase-region *start end* Command

This function converts all of the letters in the region defined by *start* and *end* to upper case. The function returns `nil`.

When `upcase-region` is called interactively, *start* and *end* are point and the mark, with the smallest first.

capitalize-word *count* Command

This function capitalizes *count* words after point, moving point over as it does. To capitalize means to convert each word's first character to upper case and convert the rest of each word to lower case. If *count* is negative, the function capitalizes the −*count* previous words but does not move point. The value is `nil`.

If point is in the middle of a word, the part of the word before point is ignored when moving forward. The rest is treated as an entire word.

When `capitalize-word` is called interactively, *count* is set to the numeric prefix argument.

downcase-word *count* Command

This function converts the *count* words after point to all lower case, moving point over as it does. If *count* is negative, it converts the −*count* previous words but does not move point. The value is `nil`.

When `downcase-word` is called interactively, *count* is set to the numeric prefix argument.

upcase-word *count* Command

This function converts the *count* words after point to all upper case, moving point over as it does. If *count* is negative, it converts the −*count* previous words but does not move point. The value is `nil`.

When `upcase-word` is called interactively, *count* is set to the numeric prefix argument.

36.18 Text Properties

Text properties are an alternative interface to extents (see Chapter 40 [Extents], page 585), and are built on top of them. They are useful when you want to view textual properties as being attached to the characters themselves rather than to intervals of characters. The text property interface is compatible with FSF Emacs.

Each character position in a buffer or a string can have a *text property list*, much like the property list of a symbol (see Section 5.9 [Property Lists], page 96). The properties belong to a particular character at a particular place, such as, the letter '`T`' at the beginning of this sentence or the first '`o`' in '`foo`'—if the same character occurs in two different places, the two occurrences generally have different properties.

Each property has a name and a value. Both of these can be any Lisp object, but the name is normally a symbol. The usual way to access the property list is to specify a name and ask what value corresponds to it.

Note that FSF Emacs also looks at the `category` property to find defaults for text properties. We consider this too bogus to implement.

Copying text between strings and buffers preserves the properties along with the characters; this includes such diverse functions as `substring`, `insert`, and `buffer-substring`.

36.18.1 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 36.18.3 [Property Search], page 542, 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.

get-char-property *pos prop* &optional *object* — Function

This function is like `get-text-property`, except that it checks all extents, not just text-property extents.

text-properties-at *position* &optional *object* — Function

This function returns the entire property list of the character at *position* in the string or buffer *object*. If *object* is `nil`, it defaults to the current buffer.

default-text-properties — Variable

This variable holds a property list giving default values for text properties. Whenever a character does not specify a value for a property, the value stored in this list is used instead. Here is an example:

```
(setq default-text-properties '(foo 69))
;; Make sure character 1 has no properties of its own.
(set-text-properties 1 2 nil)
;; What we get, when we ask, is the default value.
(get-text-property 1 'foo)
     ⇒ 69
```

36.18.2 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 (see Section 36.9 [Undo], page 523).

put-text-property *start end prop value* &optional *object* — Function

This function sets the *prop* property to *value* for the text between *start* and *end* in the string or buffer *object*. If *object* is `nil`, it defaults to the current buffer.

add-text-properties *start end props* &optional *object* Function

This function 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 5.9 [Property Lists], page 96): a list whose elements include the property names followed alternately by the corresponding values.

The return value is `t` if the function actually changed some property's value; `nil` otherwise (if *props* is `nil` or its values agree with those in the text).

For example, here is how to set the `comment` and `face` properties of a range of text:

```
(add-text-properties start end
                     '(comment t face highlight))
```

remove-text-properties *start end props* &optional *object* Function

This function deletes specified text properties from the text between *start* and *end* in the string or buffer *object*. If *object* is `nil`, it defaults to the current buffer.

The argument *props* specifies which properties to delete. It should have the form of a property list (see Section 5.9 [Property Lists], page 96): a list whose elements are property names alternating with corresponding values. But only the names matter—the values that accompany them are ignored. For example, here's how to remove the `face` property.

```
(remove-text-properties start end '(face nil))
```

The return value is `t` if the function actually changed some property's value; `nil` otherwise (if *props* is `nil` or if no character in the specified text had any of those properties).

set-text-properties *start end props* &optional *object* Function

This function completely replaces the text property list for the text between *start* and *end* in the string or buffer *object*. If *object* is `nil`, it defaults to the current buffer.

The argument *props* is the new property list. It should be a list whose elements are property names alternating with corresponding values.

After `set-text-properties` returns, all the characters in the specified range have identical properties.

If *props* is `nil`, the effect is to get rid of all properties from the specified range of text. Here's an example:

```
(set-text-properties start end nil)
```

See also the function `buffer-substring-without-properties` (see Section 36.2 [Buffer Contents], page 512) which copies text from the buffer but does not copy its properties.

36.18.3 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. They use `eq` for comparing property values. In all cases, *object* defaults to the current buffer.

For high performance, it's very important to use the *limit* argument to these functions, especially the ones that search for a single property—otherwise, they may spend a long time scanning to the end of the buffer, if the property you are interested in does not change.

Remember that a position is always between two characters; the position returned by these functions is between two characters with different properties.

next-property-change *pos* &optional *object limit* Function

The function scans the text forward from position *pos* in the string or buffer *object* till it finds a change in some text property, then returns the position of the change. In other words, it returns the position of the first character beyond *pos* whose properties are not identical to those of the character just after *pos*.

If *limit* is non-`nil`, then the scan ends at position *limit*. If there is no property change before that point, `next-property-change` returns *limit*.

The value is `nil` if the properties remain unchanged all the way to the end of *object* and *limit* is `nil`. If the value is non-`nil`, it is a position greater than or equal to *pos*. The value equals *pos* only when *limit* equals *pos*.

Here is an example of how to scan the buffer by chunks of text within which all properties are constant:

```
(while (not (eobp))
  (let ((plist (text-properties-at (point)))
        (next-change
         (or (next-property-change (point) (current-buffer))
             (point-max))))
    Process text from point to next-change...
    (goto-char next-change)))
```

next-single-property-change *pos prop* &optional *object limit* Function

The function scans the text forward from position *pos* in the string or buffer *object* till it finds a change in the *prop* property, then returns the position of the change. In other words, it returns the position of the first character beyond *pos* whose *prop* property differs from that of the character just after *pos*.

If *limit* is non-`nil`, then the scan ends at position *limit*. If there is no property change before that point, `next-single-property-change` returns *limit*.

The value is `nil` if the property remains unchanged all the way to the end of *object* and *limit* is `nil`. If the value is non-`nil`, it is a position greater than or equal to *pos*; it equals *pos* only if *limit* equals *pos*.

previous-property-change *pos* &optional *object limit* Function

This is like `next-property-change`, but scans back from *pos* instead of forward. If the value is non-`nil`, it is a position less than or equal to *pos*; it equals *pos* only if *limit* equals *pos*.

previous-single-property-change *pos prop* &optional *object limit* Function

This is like `next-single-property-change`, but scans back from *pos* instead of forward. If the value is non-`nil`, it is a position less than or equal to *pos*; it equals *pos* only if *limit* equals *pos*.

text-property-any *start end prop value* &optional *object* Function

This function returns non-`nil` if at least one character between *start* and *end* has a property *prop* whose value is *value*. More precisely, it returns the position of the first such character. Otherwise, it returns `nil`.

The optional fifth argument, *object*, specifies the string or buffer to scan. Positions are relative to *object*. The default for *object* is the current buffer.

text-property-not-all *start end prop value* &optional *object* Function

This function returns non-`nil` if at least one character between *start* and *end* 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*. The default for *object* is the current buffer.

36.18.4 Properties with Special Meanings

The predefined properties are the same as those for extents. See Section 40.6 [Extent Properties], page 591.

36.18.5 Saving Text Properties in Files

You can save text properties in files, and restore text properties when inserting the files, using these two hooks:

write-region-annotate-functions Variable

This variable's value is a list of functions for `write-region` to run to encode text properties in some fashion as annotations to the text being written in the file. See Section 28.4 [Writing to Files], page 397.

Each function in the list is called with two arguments: the start and end of the region to be written. These functions should not alter the contents of the buffer. Instead, they should return lists indicating annotations to write in the file in addition to the text in the buffer.

Each function should return a list of elements of the form `(`*position* `.` *string*`)`, where *position* is an integer specifying the relative position in the text to be written, and *string* is the annotation to add there.

Each list returned by one of these functions must be already sorted in increasing order by *position*. If there is more than one function, `write-region` merges the lists destructively into one sorted list.

When `write-region` actually writes the text from the buffer to the file, it intermixes the specified annotations at the corresponding positions. All this takes place without modifying the buffer.

after-insert-file-functions Variable

This variable holds a list of functions for `insert-file-contents` to call after inserting a file's contents. These functions should scan the inserted text for annotations, and convert them to the text properties they stand for.

Each function receives one argument, the length of the inserted text; point indicates the start of that text. The function should scan that text for annotations, delete them, and create the text properties that the annotations specify. The function should return the updated length of the inserted text, as it stands after those changes. The value returned by one function becomes the argument to the next function.

These functions should always return with point at the beginning of the inserted text.

The intended use of `after-insert-file-functions` is for converting some sort of textual annotations into actual text properties. But other uses may be possible.

We invite users to write Lisp programs to store and retrieve text properties in files, using these hooks, and thus to experiment with various data formats and find good ones. Eventually we hope users will produce good, general extensions we can install in Emacs.

We suggest not trying to handle arbitrary Lisp objects as property names or property values—because a program that general is probably difficult to write, and slow. Instead, choose a set of possible data types that are reasonably flexible, and not too hard to encode.

See Section 28.13 [Format Conversion], page 417, for a related feature.

36.19 Substituting for a Character Code

The following functions replace characters within a specified region based on their character codes.

subst-char-in-region *start end old-char new-char* &optional *noundo* Function

This function replaces all occurrences of the character *old-char* with the character *new-char* in the region of the current buffer defined by *start* and *end*.

If *noundo* is non-`nil`, then `subst-char-in-region` does not record the change for undo and does not mark the buffer as modified. This feature is used for controlling selective display (see Section 45.6 [Selective Display], page 654).

`subst-char-in-region` does not move point and returns `nil`.

```
---------- Buffer: foo ----------
This is the contents of the buffer before.
---------- Buffer: foo ----------

(subst-char-in-region 1 20 ?i ?X)
     ⇒ nil

---------- Buffer: foo ----------
ThXs Xs the contents of the buffer before.
---------- Buffer: foo ----------
```

translate-region *start end table* Function

This function applies a translation table to the characters in the buffer between positions *start* and *end*.

The translation table *table* is a string; `(aref` *table ochar*`)` gives the translated character corresponding to *ochar*. If the length of *table* is less than 256, any characters with codes larger than the length of *table* are not altered by the translation.

The return value of `translate-region` is the number of characters that were actually changed by the translation. This does not count characters that were mapped into themselves in the translation table.

36.20 Registers

A register is a sort of variable used in XEmacs 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 that 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 XEmacs 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.

get-register *reg* Function

This function returns the contents of the register *reg*, or `nil` if it has no contents.

set-register *reg value* Function

This function sets the contents of register *reg* to *value*. A register can be set to any value, but the other register functions expect only certain data types. The return value is *value*.

view-register *reg* Command

This command displays what is contained in register *reg*.

insert-register *reg* &optional *beforep* Command

This command inserts contents of register *reg* into the current buffer.

Normally, this command puts point before the inserted text, and the mark after it. However, if the optional second argument *beforep* is non-`nil`, it puts the mark before and point after. You can pass a non-`nil` second argument *beforep* to this function interactively by supplying any prefix argument.

If the register contains a rectangle, then the rectangle is inserted with its upper left corner at point. This means that text is inserted in the current line and underneath it on successive lines.

If the register contains something other than saved text (a string) or a rectangle (a list), currently useless things happen. This may be changed in the future.

36.21 Transposition of Text

This subroutine is used by the transposition commands.

transpose-regions *start1 end1 start2 end2* &optional *leave-markers* Function

This function exchanges two nonoverlapping portions of the buffer. Arguments *start1* and *end1* specify the bounds of one portion and arguments *start2* and *end2* specify the bounds of the other portion.

Normally, `transpose-regions` relocates markers with the transposed text; a marker previously positioned within one of the two transposed portions moves along with that portion, thus remaining between the same two characters in their new position. However, if *leave-markers* is non-`nil`, `transpose-regions` does not do this—it leaves all markers unrelocated.

36.22 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).

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.

Buffer changes made while executing the following hooks don't themselves cause any change hooks to be invoked.

before-change-functions Variable

This variable holds a list of a functions to call before any buffer modification. Each function gets two arguments, the beginning and end of the region that is about to change, represented as integers. The buffer that is about to change is always the current buffer.

after-change-functions Variable

This variable holds a list of a functions to call after any buffer modification. Each function receives three arguments: the beginning and end of the region just changed, and the length of the text that existed before the change. (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.

before-change-function Variable

This obsolete variable holds one function to call before any buffer modification (or `nil` for no function). It is called just like the functions in `before-change-functions`.

after-change-function Variable

This obsolete variable holds one function to call after any buffer modification (or `nil` for no function). It is called just like the functions in `after-change-functions`.

first-change-hook Variable

This variable is a normal hook that is run whenever a buffer is changed that was previously in the unmodified state.

37 Searching and Matching

XEmacs provides two ways to search through a buffer for specified text: exact string searches and regular expression searches. After a regular expression search, you can examine the *match data* to determine which text matched the whole regular expression or various portions of it.

The '`skip-chars...`' functions also perform a kind of searching. See Section 34.2.7 [Skipping Characters], page 494.

37.1 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 initially at the beginning of the line. Then `(search-forward "fox")` moves point after the last letter of '`fox`':

```
---------- Buffer: foo ----------
⋆The quick brown fox jumped over the lazy dog.
---------- Buffer: foo ----------

(search-forward "fox")
     ⇒ 20

---------- Buffer: foo ----------
The quick brown fox⋆ jumped over the lazy dog.
---------- Buffer: foo ----------
```

The argument *limit* specifies the upper bound to the search. (It must be a position in the current buffer.) No match extending after that position is accepted. If *limit* is omitted or `nil`, it defaults to the end of the accessible portion of the buffer.

What happens when the search fails depends on the value of *noerror*. If *noerror* is `nil`, a `search-failed` error is signaled. If *noerror* is `t`, `search-forward` returns `nil` and does nothing. If *noerror* is neither `nil` nor `t`, then `search-forward` moves point to the upper bound and returns `nil`. (It would be more consistent now to return the new position of point in that case, but some programs may depend on a value of `nil`.)

If *repeat* is supplied (it must be a positive number), then the search is repeated that many times (each time starting at the end of the previous time's match). If these successive searches succeed, the function succeeds, moving point and returning its new value. Otherwise the search fails.

search-backward *string* &optional *limit noerror repeat* Command

This function searches backward from point for *string*. It is just like `search-forward` except that it searches backwards and leaves point at the beginning of the match.

word-search-forward *string* &optional *limit noerror repeat* Command

This function searches forward from point for a "word" match for *string*. If it finds a match, it sets point to the end of the match found, and returns the new value of point.

Word matching regards *string* as a sequence of words, disregarding punctuation that separates them. It searches the buffer for the same sequence of words. Each word must be distinct in the buffer (searching for the word '`ball`' does not match the word '`balls`'), but the details of punctuation and spacing are ignored (searching for '`ball boy`' does match '`ball. Boy!`').

In this example, point is initially at the beginning of the buffer; the search leaves it between the '`y`' and the '`!`'.

```
---------- Buffer: foo ----------
⋆He said "Please!  Find
the ball boy!"
---------- Buffer: foo ----------

(word-search-forward "Please find the ball, boy.")
     ⇒ 35

---------- Buffer: foo ----------
He said "Please!  Find
the ball boy⋆!"
---------- Buffer: foo ----------
```

If *limit* is non-`nil` (it must be a position in the current buffer), then it is the upper bound to the search. The match found must not extend after that position.

If *noerror* is `nil`, then `word-search-forward` signals an error if the search fails. If *noerror* is `t`, then it returns `nil` instead of signaling an error. If *noerror* is neither `nil` nor `t`, it moves point to *limit* (or the end of the buffer) and returns `nil`.

If *repeat* is non-`nil`, then the search is repeated that many times. Point is positioned at the end of the last match.

word-search-backward *string* &optional *limit noerror repeat* Command

This function searches backward from point for a word match to *string*. This function is just like `word-search-forward` except that it searches backward and normally leaves point at the beginning of the match.

37.2 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 using them.

To gain a thorough understanding of regular expressions and how to use them to best advantage, we recommend that you study *Mastering Regular Expressions, by Jeffrey E.F. Friedl, O'Reilly and Associates, 1997.* (It's known as the "Hip Owls" book, because of the picture on its cover.) You might also read the manuals to ⟨undefined⟩ [(gawk)Top], page ⟨undefined⟩, ⟨undefined⟩ [(ed)Top], page ⟨undefined⟩, *sed*, *grep*, ⟨undefined⟩ [(perl)Top], page ⟨undefined⟩, ⟨undefined⟩ [(regex)Top], page ⟨undefined⟩, ⟨undefined⟩ [(rx)Top], page ⟨undefined⟩, *pcre*, and ⟨undefined⟩ [(flex)Top], page ⟨undefined⟩. All of these programs and libraries make effective use of regular expressions.

The XEmacs regular expression syntax most closely resembles that of *ed*, or *grep*, the GNU versions of which all utilize the GNU *regex* library. XEmacs' version of *regex* has recently been extended with some Perl–like capabilities, which are described in the next section.

37.2.1 Syntax of Regular Expressions

Regular expressions have a syntax in which a few characters are special constructs and the rest are *ordinary*. An ordinary character is a simple regular expression that matches that character and nothing else. The special characters are '`.`', '`*`', '`+`', '`?`', '`[`', '`]`', '`^`', '`$`', and '`\`'; no new special characters will be defined in the future. Any other character appearing in a regular expression is ordinary, unless a '`\`' precedes it.

For example, '`f`' is not a special character, so it is ordinary, and therefore '`f`' is a regular expression that matches the string '`f`' and no other string. (It does *not* match the string '`ff`'.) Likewise, '`o`' is a regular expression that matches only '`o`'.

Any two regular expressions *a* and *b* can be concatenated. The result is a regular expression that matches a string if *a* matches some amount of the beginning of that string and *b* matches the rest of the string.

As a simple example, we can concatenate the regular expressions '`f`' and '`o`' to get the regular expression '`fo`', which matches only the string '`fo`'. Still trivial. To do something more powerful, you need to use one of the special characters. Here is a list of them:

. (Period)
: is a special character that matches any single character except a newline. Using concatenation, we can make regular expressions like '`a.b`', which matches any three-character string that begins with '`a`' and ends with '`b`'.

`*`
: is not a construct by itself; it is a quantifying suffix operator that means to repeat the preceding regular expression 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; it is "greedy". 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, in matching '`ca*ar`' against the string '`caaar`', the '`a*`' first tries to match all three '`a`'s; but the rest of the pattern is '`ar`' and there is only 'r' left to match, so this try fails. The next alternative is for 'a*' to match only two 'a's. With this choice, the rest of the regexp matches successfully.

Nested repetition operators can be extremely slow if they specify backtracking loops. For example, it could take hours for the regular expression '\(x+y*\)*a' to match the sequence 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxz'. The slowness is because Emacs must try each imaginable way of grouping the 35 'x''s before concluding that none of them can work. To make sure your regular expressions run fast, check nested repetitions carefully.

`+` is a quantifying suffix operator similar to '*' except that the preceding expression must match at least once. It is also "greedy". So, for example, 'ca+r' matches the strings 'car' and 'caaaar' but not the string 'cr', whereas 'ca*r' matches all three strings.

`?` is a quantifying suffix operator similar to '*', except that the preceding expression can match either once or not at all. For example, 'ca?r' matches 'car' or 'cr', but does not match anything else.

`*?` works just like '*', except that rather than matching the longest match, it matches the shortest match. '*?' is known as a *non-greedy* quantifier, a regexp construct borrowed from Perl.

This construct is very useful for when you want to match the text inside a pair of delimiters. For instance, '/*.*?*/' will match C comments in a string. This could not be so elegantly achieved without the use of a non-greedy quantifier.

This construct has not been available prior to XEmacs 20.4. It is not available in FSF Emacs.

`+?` is the '+' analog to '*?'.

`\{n,m\}` serves as an interval quantifier, analogous to '*' or '+', but specifies that the expression must match at least *n* times, but no more than *m* times. This syntax is supported by most Unix regexp utilities, and has been introduced to XEmacs for the version 20.3.

`[ ... ]` '[' begins a *character set*, which is terminated by a ']'. In the simplest case, the characters between the two brackets 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.

The usual regular expression special characters are not special inside a character set. A completely different set of special characters exists inside character sets: ']', '-' and '^'.

'-' is used for ranges of characters. To write a range, write 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.

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 character in the set, or put it immediately after a range. (You can replace one individual character *c* with the range ‘*c*-*c*’ to make a place to put the ‘-’.) There is no way to write a set containing just ‘-’ and ‘]’.

To include ‘^’ in a set, put it anywhere but at the beginning of 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’ that occurs at the beginning of a line.

When matching a string instead of a buffer, ‘^’ matches at the beginning of the string or after a newline character ‘\n’.

`$`
: is similar to ‘^’ but matches only at the end of a line. Thus, ‘x+$’ matches a string of one ‘x’ or more at the end of a line.

When matching a string instead of a buffer, ‘$’ matches at the end of the string or before a newline character ‘\n’.

`\`
: has two functions: it quotes the special characters (including ‘\’), and it introduces additional special constructs.

Because ‘\’ quotes special characters, ‘\$’ is a regular expression that matches only ‘$’, and ‘\[’ is a regular expression that matches only ‘[’, and so on.

Note that ‘\’ also has special meaning in the read syntax of Lisp strings (see Section 2.4.8 [String Type], page 25), and must be quoted with ‘\’. For example, the regular expression that matches the ‘\’ character is ‘\\’. To write a Lisp string that contains the characters ‘\\’, Lisp syntax requires you to quote each ‘\’ with another ‘\’. Therefore, the read syntax for a regular expression matching ‘\’ is "\\\\".

Please note: For historical compatibility, special characters are treated as ordinary ones if they are in contexts where their special meanings make no sense. For example, ‘*foo’ treats ‘*’ as ordinary since there is no preceding expression on which the ‘*’ can act. It is poor practice to depend on this behavior; quote the special character anyway, regardless of where it appears.

For the most part, ‘\’ followed by any character matches only that character. However, there are several exceptions: characters that, 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 an expression for a suffix operator such as '`*`' to act 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 that happens to be assigned as a second meaning to the same '`\( ... \)`' construct because there is no conflict in practice between the two meanings. Here is an explanation of this feature:

`\`*digit* matches the same text that matched the *digit*th occurrence of a '`\( ... \)`' construct.

In other words, after the end of a '`\( ... \)`' construct. the matcher remembers the beginning and end of the text matched by that construct. Then, later on in the regular expression, you can use '`\`' followed by *digit* to match that same text, whatever it may have been.

The strings matching the first nine '`\( ... \)`' constructs appearing in a regular expression are assigned numbers 1 through 9 in the order that the open parentheses appear in the regular expression. So you can use '`\1`' through '`\9`' to refer to the text matched by the corresponding '`\( ... \)`' constructs.

For example, '`\(.*\)\1`' matches any newline-free string that is composed of two identical halves. The '`\(.*\)`' matches the first half, which may be anything, but the '`\1`' that follows must match the same exact text.

`\(?: ... \)`

is called a *shy* grouping operator, and it is used just like '`\( ... \)`', except that it does not cause the match substring to be recorded for future reference.

This is useful when you need to use a lot of nested grouping '`\( ... \)`' constructs to express complex alternation, but only want to memoize, or capture, one or two of the subexpression matches. Since '`\(?: ... \)`' doesn't capture a sub-match, it also doesn't need to be counted when you count '`\( ... \)`' groups to figure the '`match-string`' index. That turns out to be a very convenient characteristic.

This situation occurs where parts of a regular expression have been automaticly generated by a program that builds them from lists of strings, and the static code following the matching operation must access a specific match number. Here's an example that shows this.

We will assume that `(require 'regexp-opt)` has been executed already, to ensure that '`regexp-opt.el`', which is part of the `xemacs-devel` package, is loaded. In a real program, lets pretend that *varnames* would be a list of strings holding the names of some variables extracted somehow from the text of a program source you are editing and running this function on. For the purposes of this illustration, we can just bind it in the `let*` expression.

```
(let* ((varnames '("k" "n" "i" "j" "varname"))
       (keys-regexp (regexp-opt
                     (mapcar #'symbol-name
                             '(if then else elif
                               case in of do while
                               with for next unless
                               cond begin end))))
      (varname-regexp (regexp-opt varnames))
      (contrived-regexp (concat "\\(" keys-regexp "\\)"
                                "\\s-(\\s-\\("
                                varname-regexp
                                "\\)\\s-)"))
      (keyname "")
      (varname ""))
  ;; In the body of this particular defun, we:
  (re-search-forward contrived-regexp nil t)
  ;; ... and it finds a match.  Now we want to extract the
  ;; text that it matched on, and save it into keyname
  ;; and varname.
  (setq keyname (match-string 1)
        varname (match-string 2))
  ;; ... and then do something with those values.
  (list keyname varname))

;; Here's something for it to match, so you can try it with
;; C-x C-e
;; while ( j ) do ...
```

Here you should see that if the regular expression returned by `regexp-opt` did not use '`\(?: ... \)`' for grouping, and instead used '`\( ... \)`', it would be necessary to count the number of opening parentheses in the `keys-regexp` and to use that figure to calculate which match number is matched by the `varname-regexp`. It is much more convenient to be able to just ask for the second match string.

The shy grouping operator has been borrowed from Perl, and has not been available prior to XEmacs 20.3, nor is it available in FSF Emacs.

`\w` matches any word-constituent character. The editor syntax table determines which characters these are. See Chapter 38 [Syntax Tables], page 569.

`\W` matches any character that is not a word constituent.

`\scode` matches any character whose syntax is *code*. Here *code* is a character that represents a syntax code: thus, '`w`' for word constituent, '`-`' for whitespace, '`(`' for open parenthesis, etc. See Chapter 38 [Syntax Tables], page 569, for a list of syntax codes and the characters that stand for them.

`\S`*code*
: matches any character whose syntax is not *code*.

The following regular expression constructs match the empty string—that is, they don't use up any characters—but whether they match depends on the context.

`` \` ``
: matches the empty string, but only at the beginning of the buffer or string being matched against.

`\'`
: matches the empty string, but only at the end of the buffer or string being matched against.

`\=`
: matches the empty string, but only at point. (This construct is not defined when matching against a string.)

`\b`
: matches the empty string, but only at the beginning or end of a word. Thus, '`\bfoo\b`' matches any occurrence of '`foo`' as a separate word. '`\bballs?\b`' matches '`ball`' or '`balls`' as a separate word.

`\B`
: matches the empty string, but *not* at the beginning or end of a word.

`\<`
: matches the empty string, but only at the beginning of a word.

`\>`
: matches the empty string, but only at the end of a word.

Not every string is a valid regular expression. For example, a string with unbalanced square brackets is invalid (with a few exceptions, such as '`[]]`'), and so is a string that ends with a single '`\`'. If an invalid regular expression is passed to any of the search functions, an `invalid-regexp` error is signaled.

regexp-quote *string* — Function

This function returns a regular expression string that matches exactly *string* and nothing else. This allows you to request an exact string match when calling a function that wants a regular expression.

```
(regexp-quote "^The cat$")
     ⇒ "\\^The cat\\$"
```

One use of `regexp-quote` is to combine an exact string match with context described as a regular expression. For example, this searches for the string that is the value of `string`, surrounded by whitespace:

```
(re-search-forward
 (concat "\\s-" (regexp-quote string) "\\s-"))
```

37.2.2 Complex Regexp Example

Here is a complicated regexp, used by XEmacs to recognize the end of a sentence together with any whitespace that follows. It is the value of the variable `sentence-end`.

First, we show the regexp as a string in Lisp syntax to distinguish spaces from tab characters. The string constant begins and ends with a double-quote. '`\"`' stands for a double-quote as part of the string, '`\\`' for a backslash as part of the string, '`\t`' for a tab and '`\n`' for a newline.

```
"[.?!][]\"')}]*\\($\\| $\\|\t\\|  \\)[ \t\n]*"
```

In contrast, if you evaluate the variable `sentence-end`, you will see the following:

```
sentence-end
⇒
"[.?!][]\"')}]*\\($\\| $\\|  \\|  \\)[
]*"
```

In this output, tab and newline appear as themselves.

This regular expression contains four parts in succession and can be deciphered as follows:

`[.?!]` The first part of the pattern is a character set that matches any one of three characters: period, question mark, and exclamation mark. The match must begin with one of these three characters.

`[]\"')}]*`
: The second part of the pattern matches any closing braces and quotation marks, zero or more of them, that may follow the period, question mark or exclamation mark. The `\"` is Lisp syntax for a double-quote in a string. The '`*`' at the end indicates that the immediately preceding regular expression (a character 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 mark the parentheses and vertical bars as regular expression syntax; the parentheses delimit a group and the vertical bars separate alternatives. The dollar sign is used to match the end of a line.

`[ \t\n]*` Finally, the last part of the pattern matches any additional whitespace beyond the minimum needed to end a sentence.

37.3 Regular Expression Searching

In XEmacs, you can search for the next match for a regexp either incrementally or not. Incremental search commands are described in the *The XEmacs Reference Manual*. See section "Regular Expression Search" in *The XEmacs Reference 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 match found. It returns the new value of point.

If *limit* is non-`nil` (it must be a position in the current buffer), then it is the upper bound to the search. No match extending after that position is accepted.

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 *repeat* is supplied (it must be a positive number), then the search is repeated that many times (each time starting at the end of the previous time's match).

If these successive searches succeed, the function succeeds, moving point and returning its new value. Otherwise the search fails.

In the following example, point is initially before the '`T`'. Evaluating the search call moves point to the end of that line (between the '`t`' of '`hat`' and the newline).

```
---------- Buffer: foo ----------
I read "⋆The cat in the hat
comes back" twice.
---------- Buffer: foo ----------

(re-search-forward "[a-z]+" nil t 5)
     ⇒ 27

---------- Buffer: foo ----------
I read "The cat in the hat⋆
comes back" twice.
---------- Buffer: foo ----------
```

re-search-backward *regexp* &optional *limit noerror repeat* Command

This function searches backward in the current buffer for a string of text that is matched by the regular expression *regexp*, leaving point at the beginning of the first text found.

This function is analogous to `re-search-forward`, but they are not simple mirror images. `re-search-forward` finds the match whose beginning is as close as possible to the starting point. If `re-search-backward` were a perfect mirror image, it would find the match whose end is as close as possible. However, in fact it finds the match whose beginning is as close as possible. The reason is that matching a regular expression at a given spot always works from beginning to end, and starts at a specified beginning position.

A true mirror-image of `re-search-forward` would require a special feature for matching regexps from end to beginning. It's not worth the trouble of implementing that.

string-match *regexp string* &optional *start* Function

This function returns the index of the start of the first match for the regular expression *regexp* in *string*, or `nil` if there is no match. If *start* is non-`nil`, the search starts at that index in *string*.

For example,

```
(string-match
 "quick" "The quick brown fox jumped quickly.")
     ⇒ 4
(string-match
 "quick" "The quick brown fox jumped quickly." 8)
     ⇒ 27
```

The index of the first character of the string is 0, the index of the second character is 1, and so on.

After this function returns, the index of the first character beyond the match is available as `(match-end 0)`. See Section 37.6 [Match Data], page 561.

```
(string-match
 "quick" "The quick brown fox jumped quickly." 8)
     ⇒ 27
(match-end 0)
     ⇒ 32
```

split-string *string* &optional *pattern* Function

This function splits *string* to substrings delimited by *pattern*, and returns a list of substrings. If *pattern* is omitted, it defaults to '`[ \f\t\n\r\v]+`', which means that it splits *string* by white–space.

```
(split-string "foo bar")
     ⇒ ("foo" "bar")
(split-string "something")
     ⇒ ("something")
(split-string "a:b:c" ":")
     ⇒ ("a" "b" "c")
(split-string ":a::b:c" ":")
     ⇒ ("" "a" "" "b" "c")
```

split-path *path* Function

This function splits a search path into a list of strings. The path components are separated with the characters specified with `path-separator`. Under Unix, `path-separator` will normally be '`:`', while under Windows, it will be '`;`'.

looking-at *regexp* Function

This function determines whether the text in the current buffer directly following point matches the regular expression *regexp*. "Directly following" means precisely that: the search is "anchored" and it can succeed only starting with the first character following point. The result is `t` if so, `nil` otherwise.

This function does not move point, but it updates the match data, which you can access using `match-beginning` and `match-end`. See Section 37.6 [Match Data], page 561.

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
```

37.4 POSIX Regular Expression Searching

The usual regular expression functions do backtracking when necessary to handle the '`\|`' and repetition constructs, but they continue this only until they find *some* match. Then they succeed and report the first match found.

This section describes alternative search functions which perform the full backtracking specified by the POSIX standard for regular expression matching. They continue backtracking until they have tried all possibilities and found all matches, so they can report the longest match, as required by POSIX. This is much slower, so use these functions only when you really need the longest match.

In Emacs versions prior to 19.29, these functions did not exist, and the functions described above implemented full POSIX backtracking.

posix-search-forward *regexp* &optional *limit noerror repeat* — Function

This is like `re-search-forward` except that it performs the full backtracking specified by the POSIX standard for regular expression matching.

posix-search-backward *regexp* &optional *limit noerror repeat* — Function

This is like `re-search-backward` except that it performs the full backtracking specified by the POSIX standard for regular expression matching.

posix-looking-at *regexp* — Function

This is like `looking-at` except that it performs the full backtracking specified by the POSIX standard for regular expression matching.

posix-string-match *regexp string* &optional *start* — Function

This is like `string-match` except that it performs the full backtracking specified by the POSIX standard for regular expression matching.

37.5 Search and Replace

perform-replace *from-string replacements query-flag regexp-flag delimited-flag* &optional *repeat-count map* — Function

This function is the guts of `query-replace` and related commands. It searches for occurrences of *from-string* and replaces some or all of them. If *query-flag* is `nil`, it replaces all occurrences; otherwise, it asks the user what to do about each one.

If *regexp-flag* is non-`nil`, then *from-string* is considered a regular expression; otherwise, it must match literally. If *delimited-flag* is non-`nil`, then only replacements surrounded by word boundaries are considered.

The argument *replacements* specifies what to replace occurrences with. If it is a string, that string is used. It can also be a list of strings, to be used in cyclic order.

If *repeat-count* is non-`nil`, it should be an integer. Then it specifies how many times to use each of the strings in the *replacements* list before advancing cyclicly to the next one.

Normally, the keymap `query-replace-map` defines the possible user responses for queries. The argument *map*, if non-`nil`, is a keymap to use instead of `query-replace-map`.

query-replace-map Variable

This variable holds a special keymap that defines the valid user responses for `query-replace` and related functions, as well as `y-or-n-p` and `map-y-or-n-p`. It is unusual in two ways:

- The "key bindings" are not commands, just symbols that are meaningful to the functions that use this map.
- Prefix keys are not supported; each key binding must be for a single event key sequence. This is because the functions don't use read key sequence to get the input; instead, they read a single event and look it up "by hand."

Here are the meaningful "bindings" for `query-replace-map`. Several of them are meaningful only for `query-replace` and friends.

`act` Do take the action being considered—in other words, "yes."

`skip` Do not take action for this question—in other words, "no."

`exit` Answer this question "no," and give up on the entire series of questions, assuming that the answers will be "no."

`act-and-exit`
Answer this question "yes," and give up on the entire series of questions, assuming that subsequent answers will be "no."

`act-and-show`
Answer this question "yes," but show the results—don't advance yet to the next question.

`automatic`
Answer this question and all subsequent questions in the series with "yes," without further user interaction.

`backup` Move back to the previous place that a question was asked about.

`edit` Enter a recursive edit to deal with this question—instead of any other action that would normally be taken.

`delete-and-edit`
Delete the text being considered, then enter a recursive edit to replace it.

`recenter` Redisplay and center the window, then ask the same question again.

`quit` Perform a quit right away. Only `y-or-n-p` and related functions use this answer.

`help` Display some help, then ask again.

37.6 The Match Data

XEmacs keeps track of the positions of the start and end of segments of text found during a regular expression search. This means, for example, that you can search for a complex pattern, such as a date in an Rmail message, and then extract parts of the match under control of the pattern.

Because the match data normally describe the most recent search only, you must be careful not to do another search inadvertently between the search you wish to refer back to and the use of the match data. If you can't avoid another intervening search, you must save and restore the match data around it, to prevent it from being overwritten.

37.6.1 Simple Match Data Access

This section explains how to use the match data to find out what was matched by the last search or match operation.

You can ask about the entire matching text, or about a particular parenthetical subexpression of a regular expression. The *count* argument in the functions below specifies which. If *count* is zero, you are asking about the entire match. If *count* is positive, it specifies which subexpression you want.

Recall that the subexpressions of a regular expression are those expressions grouped with escaped parentheses, '`\(...\)`'. The *count*th subexpression is found by counting occurrences of '`\(`' from the beginning of the whole regular expression. The first subexpression is numbered 1, the second 2, and so on. Only regular expressions can have subexpressions—after a simple string search, the only information available is about the entire match.

match-string *count* &optional *in-string* Function

> This function returns, as a string, the text matched in the last search or match operation. It returns the entire text if *count* is zero, or just the portion corresponding to the *count*th parenthetical subexpression, if *count* is positive. If *count* is out of range, or if that subexpression didn't match anything, the value is `nil`.
>
> If the last such operation was done against a string with `string-match`, then you should pass the same string as the argument *in-string*. Otherwise, after a buffer search or match, you should omit *in-string* or pass `nil` for it; but you should make sure that the current buffer when you call `match-string` is the one in which you did the searching or matching.

match-beginning *count* Function

> This function returns the position of the start of text matched by the last regular expression searched for, or a subexpression of it.
>
> If *count* is zero, then the value is the position of the start of the entire match. Otherwise, *count* specifies a subexpression in the regular expression, and the value of the function is the starting position of the match for that subexpression.
>
> The value is `nil` for a subexpression inside a '`\|`' alternative that wasn't used in the match.

match-end *count* Function

> This function is like `match-beginning` except that it returns the position of the end of the match, rather than the position of the beginning.

Here is an example of using the match data, with a comment showing the positions within the text:

```
(string-match "\\(qu\\)\\(ick\\)"
              "The quick fox jumped quickly.")
              ;0123456789
     ⇒ 4

(match-string 0 "The quick fox jumped quickly.")
     ⇒ "quick"
(match-string 1 "The quick fox jumped quickly.")
     ⇒ "qu"
(match-string 2 "The quick fox jumped quickly.")
     ⇒ "ick"

(match-beginning 1)       ; The beginning of the match
     ⇒ 4                  ;    with ‘qu’ is at index 4.

(match-beginning 2)       ; The beginning of the match
     ⇒ 6                  ;    with ‘ick’ is at index 6.

(match-end 1)             ; The end of the match
     ⇒ 6                  ;    with ‘qu’ is at index 6.

(match-end 2)             ; The end of the match
     ⇒ 9                  ;    with ‘ick’ is at index 9.
```

Here is another example. Point is initially located at the beginning of the line. Searching moves point to between the space and the word ‘in’. The beginning of the entire match is at the 9th character of the buffer (‘T’), and the beginning of the match for the first subexpression is at the 13th character (‘c’).

```
(list
  (re-search-forward "The \\(cat \\)")
  (match-beginning 0)
  (match-beginning 1))
    ⇒ (9 9 13)

---------- Buffer: foo ----------
I read "The cat ⋆in the hat comes back" twice.
        ^   ^
        9  13
---------- Buffer: foo ----------
```

(In this case, the index returned is a buffer position; the first character of the buffer counts as 1.)

37.6.2 Replacing the Text That Matched

This function replaces the text matched by the last search with *replacement*.

replace-match *replacement* &optional *fixedcase literal string* Function

This function replaces the text in the buffer (or in *string*) that was matched by the last search. It replaces that text with *replacement*.

If you did the last search in a buffer, you should specify `nil` for *string*. Then `replace-match` does the replacement by editing the buffer; it leaves point at the end of the replacement text, and returns `t`.

If you did the search in a string, pass the same string as *string*. Then `replace-match` does the replacement by constructing and returning a new string.

If *fixedcase* is non-`nil`, then the case of the replacement text is not changed; otherwise, the replacement text is converted to a different case depending upon the capitalization of the text to be replaced. If the original text is all upper case, the replacement text is converted to upper case. If the first word of the original text is capitalized, then the first word of the replacement text is capitalized. If the original text contains just one word, and that word is a capital letter, `replace-match` considers this a capitalized first word rather than all upper case.

If `case-replace` is `nil`, then case conversion is not done, regardless of the value of *fixed-case*. See Section 37.7 [Searching and Case], page 565.

If *literal* is non-`nil`, then *replacement* is inserted exactly as it is, the only alterations being case changes as needed. If it is `nil` (the default), then the character '`\`' is treated specially. If a '`\`' appears in *replacement*, then it must be part of one of the following sequences:

'`\&`'
: '`\&`' stands for the entire text being replaced.

'`\`*n*'
: '`\`*n*', where *n* is a digit, stands for the text that matched the *n*th subexpression in the original regexp. Subexpressions are those expressions grouped inside '`\(...\)`'.

'`\\`'
: '`\\`' stands for a single '`\`' in the replacement text.

37.6.3 Accessing the Entire Match Data

The functions `match-data` and `set-match-data` read or write the entire match data, all at once.

match-data Function

This function returns a newly constructed list containing all the information on what text the last search matched. Element zero is the position of the beginning of the match for the whole expression; element one is the position of the end of the match for the expression. The next two elements are the positions of the beginning and end of the match for the first subexpression, and so on. In general, element number $2n$ corresponds to `(match-beginning` *n*`)`; and element number $2n + 1$ corresponds to `(match-end` *n*`)`.

All the elements are markers or `nil` if matching was done on a buffer, and all are integers or `nil` if matching was done on a string with `string-match`. (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>)
```

set-match-data *match-list* Function

This function sets the match data from the elements of *match-list*, which should be a list that was the value of a previous call to `match-data`.

If *match-list* refers to a buffer that doesn't exist, you don't get an error; that sets the match data in a meaningless but harmless way.

`store-match-data` is an alias for `set-match-data`.

37.6.4 Saving and Restoring the Match Data

When you call a function that may do a search, you may need to save and restore the match data around that call, if you want to preserve the match data from an earlier search for later use. Here is an example that shows the problem that arises if you fail to save the match data:

```
(re-search-forward "The \\(cat \\)")
     ⇒ 48
(foo)                       ; Perhaps foo does
                            ;     more searching.
(match-end 0)
     ⇒ 61                   ; Unexpected result—not 48!
```

You can save and restore the match data with `save-match-data`:

save-match-data *body...* Macro

This special form executes *body*, saving and restoring the match data around it.

You can use `set-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.
    (set-match-data data)))
```

Emacs automatically saves and restores the match data when it runs process filter functions (see Section 49.9.2 [Filter Functions], page 684) and process sentinels (see Section 49.10 [Sentinels], page 687).

37.7 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 buffer-local variable; altering the variable affects only the current buffer. (See Section 10.9.1 [Intro to Buffer-Local], page 157.) Alternatively, you may change the value of `default-case-fold-search`, which is the default value of `case-fold-search` for buffers that do not override it.

Note that the user-level incremental search feature handles case distinctions differently. When given a lower case letter, it looks for a match of either case, but when given an upper case letter, it looks for an upper case letter only. But this has nothing to do with the searching functions Lisp functions use.

case-replace User Option

This variable determines whether the replacement functions should preserve case. If the variable is `nil`, that means to use the replacement text verbatim. A non-`nil` value means to convert the case of the replacement text according to the text being replaced.

The function `replace-match` is where this variable actually has its effect. See Section 37.6.2 [Replacing Match], page 563.

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)`.

37.8 Standard Regular Expressions Used in Editing

This section describes some variables that hold regular expressions used for certain purposes in editing:

page-delimiter Variable

This is the regexp describing line-beginnings that separate pages. The default value is `"^\014"` (i.e., `"^^L"` or `"^\C-l"`); this matches a line that starts with a formfeed character.

The following two regular expressions should *not* assume the match always starts at the beginning of a line; they should not use '`^`' to anchor the match. Most often, the paragraph commands do check for a match only at the beginning of a line, which means that '`^`' would be superfluous. When there is a nonzero left margin, they accept matches that start after the left margin. In that case, a '`^`' would be incorrect. However, a '`^`' is harmless in modes where a left margin is never used.

paragraph-separate Variable

This is the regular expression for recognizing the beginning of a line that separates paragraphs. (If you change this, you may have to change `paragraph-start` also.) The default value is `"[ \t\f]*$"`, which matches a line that consists entirely of spaces, tabs, and form feeds (after its left margin).

paragraph-start Variable

This is the regular expression for recognizing the beginning of a line that starts *or* separates paragraphs. The default value is `"[ \t\n\f]"`, which matches a line starting with a space, tab, newline, or form feed (after its left margin).

sentence-end Variable

This is the regular expression describing the end of a sentence. (All paragraph boundaries also end sentences, regardless.) The default value is:

```
"[.?!][]\"')}]*\\($\\| $\\|\t\\| \\)[ \t\n]*"
```

This means a period, question mark or exclamation mark, followed optionally by a closing parenthetical character, followed by tabs, spaces or new lines.

For a detailed explanation of this regular expression, see Section 37.2.2 [Regexp Example], page 556.

38 Syntax Tables

A *syntax table* specifies the syntactic textual function of each character. 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 34.2.2 [Word Motion], page 489) and the list motion functions (see Section 34.2.6 [List Motion], page 493) as well as the functions in this chapter.

38.1 Syntax Table Concepts

Under XEmacs 20, a syntax table is a particular subtype of the primitive char table type (see Section 4.13 [Char Tables], page 73), and each element of the char table is an integer that encodes the syntax of the character in question, or a cons of such an integer and a matching character (for characters with parenthesis syntax).

Under XEmacs 19, a syntax table is a vector of 256 elements; it contains one entry for each of the 256 possible characters in an 8-bit byte. Each element is an integer that encodes the syntax of the character in question. (The matching character, if any, is embedded in the bits of this integer.)

Syntax tables are used only for moving across text, not for the Emacs Lisp reader. XEmacs 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, XEmacs makes the choice of syntax table local to each buffer. Typically, each major mode has its own syntax table and installs that table in each buffer that uses that mode. Changing this table alters the syntax in all those buffers as well as in any buffers subsequently put in that mode. Occasionally several similar modes share one syntax table. See Section 26.1.2 [Example Major Modes], page 363, for an example of how to set up a syntax table.

A syntax table can inherit the data for some characters from the standard syntax table, while specifying other characters itself. The "inherit" syntax class means "inherit this character's syntax from the standard syntax table." Most major modes' syntax tables inherit the syntax of character codes 0 through 31 and 128 through 255. This is useful with character sets such as ISO Latin-1 that have additional alphabetic characters in the range 128 to 255. Just changing the standard syntax for these characters affects all major modes.

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.

38.2 Syntax Descriptors

This section describes the syntax classes and flags that denote the syntax of a character, and how they are represented as a *syntax descriptor*, which is a Lisp string that you pass to `modify-syntax-entry` to specify the desired syntax.

XEmacs defines a number of *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 that is frequently in that class; however, its meaning as a designator is unvarying and independent of what syntax that character currently has.

A syntax descriptor is a Lisp string that specifies a syntax class, a matching character (used only for the parenthesis classes) and flags. The first character is the designator for a syntax class. The second character is the character to match; if it is unused, put a space there. Then come the characters for any desired flags. If no matching character or flags are needed, one character is sufficient.

For example, the 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).

38.2.1 Table of Syntax Classes

Here is a table of syntax classes, the characters that stand for them, their meanings, and examples of their use.

whitespace character Syntax class

Whitespace characters (designated with '` `' or '`-`') separate symbols and words from each other. Typically, whitespace characters have no other syntactic significance, and multiple whitespace characters are syntactically equivalent to a single one. Space, tab, newline and formfeed are almost always classified as 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, XEmacs indicates momentarily the matching open parenthesis when you insert a close parenthesis. See Section 45.9 [Blinking], page 658.

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 XEmacs Lisp, the delimiters for lists and vectors ('()' and '[]') are classified as parenthesis characters.

string quote Syntax class

String quote characters (designated with '"') are used in many languages, including Lisp and C, to delimit string constants. The same string quote character appears at the beginning and the end of a string. Such quoted strings do not nest.

The parsing facilities of XEmacs 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 XEmacs 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 34.2.2 [Word Motion], page 489.

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 34.2.2 [Word Motion], page 489.

This class is used for backslash in TEX mode.

paired delimiter Syntax class

Paired delimiter characters (designated with '`$`') are like string quote characters except that the syntactic properties of the characters between the delimiters are not suppressed. Only TEX mode uses a paired delimiter presently—the '`$`' that both enters and leaves 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. 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 various 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.

inherit Syntax class

This syntax class does not specify a syntax. It says to look in the standard syntax table to find the syntax of this character. The designator for this syntax code is '`@`'.

38.2.2 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 supports two comment styles simultaneously in any one syntax table. This is for the sake of C++. Each style of comment syntax has its own comment-start sequence and its own comment-end sequence. Each comment must stick to one style or the other; thus, if it starts with the comment-start sequence of style "b", it must also end with the comment-end sequence of style "b".

  The two comment-start sequences must begin with the same character; only the second character may differ. Mark the second character of the "b"-style comment-start sequence with the 'b' flag.

  A comment-end sequence (one or two characters) applies to the "b" style if its first character has the 'b' flag set; otherwise, it applies to the "a" style.

  The appropriate comment syntax settings for C++ are as follows:

  '/' '124b'

  '*' '23'

  newline '>b'

  This defines four comment-delimiting sequences:

  '/*' This is a comment-start sequence for "a" style because the second character, '*', does not have the 'b' flag.

  '//' This is a comment-start sequence for "b" style because the second character, '/', does have the 'b' flag.

  '*/' This is a comment-end sequence for "a" style because the first character, '*', does not have the 'b' flag

  newline This is a comment-end sequence for "b" style, because the newline character has the 'b' flag.

- 'p' identifies an additional "prefix character" for Lisp syntax. These characters are treated as whitespace when they appear between expressions. When they appear within an expression, they are handled according to their usual syntax codes.

  The function `backward-prefix-chars` moves back over these characters, as well as over characters whose primary syntax class is prefix ('''). See Section 38.4 [Motion and Syntax], page 575.

38.3 Syntax Table Functions

In this section we describe functions for creating, accessing and altering syntax tables.

make-syntax-table &optional *table* Function

This function creates a new syntax table. Character codes 0 through 31 and 128 through 255 are set up to inherit from the standard syntax table. The other character codes are set up by copying what the standard syntax table says about them.

Most major mode syntax tables are created in this way.

copy-syntax-table &optional *table* Function

This function constructs a copy of *table* and returns it. If *table* is not supplied (or is `nil`), it returns a copy of the current syntax table. Otherwise, an error is signaled if *table* is not a syntax table.

modify-syntax-entry *char syntax-descriptor* &optional *table* Command

This function sets the syntax entry for *char* according to *syntax-descriptor*. The syntax is changed only for *table*, which defaults to the current buffer's syntax table, and not in any other syntax table. The argument *syntax-descriptor* specifies the desired syntax; this is a string beginning with a class designator character, and optionally containing a matching character and flags as well. See Section 38.2 [Syntax Descriptors], page 570.

This function always returns `nil`. The old syntax information in the table for this character is discarded.

An error is signaled if the first character of the syntax descriptor is not one of the twelve syntax class designator characters. An error is also signaled if *char* is not a character.

Examples:

```
;; Put the space character in class whitespace.
(modify-syntax-entry ?\  " ")
     ⇒ nil

;; Make '$' an open parenthesis character,
;;   with '^' as its matching close.
(modify-syntax-entry ?$ "(^")
     ⇒ nil

;; Make '^' a close parenthesis character,
;;   with '$' as its matching open.
(modify-syntax-entry ?^ ")$")
     ⇒ nil

;; Make '/' a punctuation character,
;;   the first character of a start-comment sequence,
;;   and the second character of an end-comment sequence.
;;   This is used in C mode.
(modify-syntax-entry ?/ ". 14")
     ⇒ nil
```

char-syntax *character* Function

This function returns the syntax class of *character*, represented by its mnemonic designator character. This *only* returns the class, not any matching parenthesis or flags.

An error is signaled if *char* is not a character.

The following examples apply to C mode. The first example shows that the syntax class of space is whitespace (represented by a space). The second example shows that the syntax of '/' is punctuation. This does not show the fact that it is also part of comment-start and -end sequences. The third example shows that open parenthesis is in the class of open parentheses. This does not show the fact that it has a matching character, ')'.

```
(char-to-string (char-syntax ?\ ))
     ⇒ " "

(char-to-string (char-syntax ?/))
     ⇒ "."

(char-to-string (char-syntax ?\())
     ⇒ "("
```

set-syntax-table *table* &optional *buffer* Function

This function makes *table* the syntax table for *buffer*, which defaults to the current buffer if omitted. It returns *table*.

syntax-table &optional *buffer* Function

This function returns the syntax table for *buffer*, which defaults to the current buffer if omitted.

38.4 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 buffer* Function

This function moves point forward across characters having syntax classes mentioned in *syntaxes*. It stops when it encounters the end of the buffer, or position *limit* (if specified), or a character it is not supposed to skip. Optional argument *buffer* defaults to the current buffer if omitted.

skip-syntax-backward *syntaxes* &optional *limit buffer* Function

This function moves point backward across characters whose syntax classes are mentioned in *syntaxes*. It stops when it encounters the beginning of the buffer, or position *limit* (if specified), or a character it is not supposed to skip. Optional argument *buffer* defaults to the current buffer if omitted.

backward-prefix-chars &optional *buffer* Function

This function moves point backward over any number of characters with expression prefix syntax. This includes both characters in the expression prefix syntax class, and characters with the 'p' flag. Optional argument *buffer* defaults to the current buffer if omitted.

38.5 Parsing Balanced Expressions

Here are several functions for parsing and scanning balanced expressions, also known as *sexps*, in which parentheses match in pairs. The syntax table controls the interpretation of characters, so these functions can be used for Lisp expressions when in Lisp mode and for C expressions when in C mode. See Section 34.2.6 [List Motion], page 493, for convenient higher-level functions for moving over balanced expressions.

parse-partial-sexp *start limit* &optional *target-depth stop-before state stop-comment buffer* Function

This function parses a sexp in the current buffer starting at *start*, not scanning past *limit*. It stops at position *limit* or when certain criteria described below are met, and sets point to the location where parsing stops. It returns a value describing the status of the parse at the point where it stops.

If *state* is `nil`, *start* is assumed to be at the top level of parenthesis structure, such as the beginning of a function definition. Alternatively, you might wish to resume parsing in the middle of the structure. To do this, you must provide a *state* argument that describes the initial status of parsing.

If the third argument *target-depth* is non-`nil`, parsing stops if the depth in parentheses becomes equal to *target-depth*. The depth starts at 0, or at whatever is given in *state*.

If the fourth argument *stop-before* is non-`nil`, parsing stops when it comes to any character that starts a sexp. If *stop-comment* is non-`nil`, parsing stops when it comes to the start of a comment.

The fifth argument *state* is an eight-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 eight elements describing the final state of the parse:

0. The depth in parentheses, counting from 0.
1. The character position of the start of the innermost parenthetical grouping containing the stopping point; `nil` if none.
2. The character position of the start of the last complete subexpression terminated; `nil` if none.
3. Non-`nil` if inside a string. More precisely, this is the character that will terminate the string.
4. `t` if inside a comment (of either style).
5. `t` if point is just after a quote character.
6. The minimum parenthesis depth encountered during this scan.
7. `t` if inside a comment of style "b".

Elements 0, 3, 4, 5 and 7 are significant in the argument *state*.

This function is most often used to compute indentation for languages that have nested parentheses.

scan-lists *from count depth* &optional *buffer noerror* Function

This function scans forward *count* balanced parenthetical groupings from character number *from*. It returns the character position where the scan stops.

If *depth* is nonzero, parenthesis depth counting begins from that value. The only candidates for stopping are places where the depth in parentheses becomes zero; `scan-lists` counts *count* such places and then stops. Thus, a positive value for *depth* means go out *depth* levels of parenthesis.

Scanning ignores comments if `parse-sexp-ignore-comments` is non-`nil`.

If the scan reaches the beginning or end of the buffer (or its accessible portion), and the depth is not zero, an error is signaled. If the depth is zero but the count is not used up, `nil` is returned.

If optional arg *buffer* is non-`nil`, scanning occurs in that buffer instead of in the current buffer.

If optional arg *noerror* is non-`nil`, `scan-lists` will return `nil` instead of signalling an error.

scan-sexps *from count* &optional *buffer noerror* Function

This function scans forward *count* sexps from character position *from*. It returns the character position where the scan stops.

Scanning ignores comments if `parse-sexp-ignore-comments` is non-`nil`.

If the scan reaches the beginning or end of (the accessible part of) the buffer in the middle of a parenthetical grouping, an error is signaled. If it reaches the beginning or end between groupings but before count is used up, `nil` is returned.

If optional arg *buffer* is non-`nil`, scanning occurs in that buffer instead of in the current buffer.

If optional arg *noerror* is non-`nil`, `scan-sexps` will return nil instead of signalling an error.

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* &optional *buffer* 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*.

Optional argument *buffer* defaults to the current buffer.

To move forward over all comments and whitespace following point, use `(forward-comment (buffer-size))`. `(buffer-size)` is a good argument to use, because the number of comments in the buffer cannot exceed that many.

38.6 Some Standard Syntax Tables

Most of the major modes in XEmacs have their own syntax tables. Here are several of them:

standard-syntax-table Function

This function returns the standard syntax table, which is the syntax table used in Fundamental mode.

text-mode-syntax-table Variable

The value of this variable is the syntax table used in Text mode.

c-mode-syntax-table Variable

The value of this variable is the syntax table for C-mode buffers.

emacs-lisp-mode-syntax-table Variable

The value of this variable is the syntax table used in Emacs Lisp mode by editing commands. (It has no effect on the Lisp `read` function.)

38.7 Syntax Table Internals

Each element of a syntax table is an integer that encodes the syntax of one character: the syntax class, possible matching character, and flags. Lisp programs don't usually work with the elements directly; the Lisp-level syntax table functions usually work with syntax descriptors (see Section 38.2 [Syntax Descriptors], page 570).

The low 8 bits of each element of a syntax table indicate 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
13	inherit

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.

39 Abbrevs And Abbrev Expansion

An abbreviation or *abbrev* is a string of characters that may be expanded to a longer string. The user can insert the abbrev string and find it replaced automatically with the expansion of the abbrev. This saves typing.

The set of abbrevs currently in effect is recorded in an *abbrev table*. Each buffer has a local abbrev table, but normally all buffers in the same major mode share one abbrev table. There is also a global abbrev table. Normally both are used.

An abbrev table is represented as an obarray containing a symbol for each abbreviation. The symbol's name is the abbreviation; its value is the expansion; its function definition is the hook function to do the expansion (see Section 39.3 [Defining Abbrevs], page 580); its property list cell contains the use count, the number of times the abbreviation has been expanded. Because these symbols are not interned in the usual obarray, they will never appear as the result of reading a Lisp expression; in fact, normally they are never used except by the code that handles abbrevs. Therefore, it is safe to use them in an extremely nonstandard way. See Section 7.3 [Creating Symbols], page 113.

For the user-level commands for abbrevs, see section "Abbrev Mode" in *The XEmacs Reference Manual*.

39.1 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 of `abbrev-mode` for buffers that do not override it. This is the same as `(default-value 'abbrev-mode)`.

39.2 Abbrev Tables

This section describes how to create and manipulate abbrev tables.

make-abbrev-table Function

> This function creates and returns a new, empty abbrev table—an obarray containing no symbols. It is a vector filled with zeros.

clear-abbrev-table *table* Function

> This function undefines all the abbrevs in abbrev table *table*, leaving it empty. The function returns `nil`.

define-abbrev-table *tabname definitions* Function

This function defines *tabname* (a symbol) as an abbrev table name, i.e., as a variable whose value is an abbrev table. It defines abbrevs in the table according to *definitions*, a list of elements of the form `(`*abbrevname expansion hook usecount*`)`. The 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`, the description is human-oriented. Otherwise the description is a Lisp expression—a call to `define-abbrev-table` that would define *name* exactly as it is currently defined.

39.3 Defining Abbrevs

These functions define an abbrev in a specified abbrev table. `define-abbrev` is the low-level basic function, while `add-abbrev` is used by commands that ask for information from the user.

add-abbrev *table type arg* Function

This function adds an abbreviation to abbrev table *table* based on information from the user. The argument *type* is a string describing in English the kind of abbrev this will be (typically, `"global"` or `"mode-specific"`); this is used in prompting the user. The argument *arg* is the number of words in the expansion.

The return value is the symbol that internally represents the new abbrev, or `nil` if the user declines to confirm redefining an existing abbrev.

define-abbrev *table name expansion hook* Function

This function defines an abbrev in *table* named *name*, to expand to *expansion*, and call *hook*. The return value is an uninterned symbol that represents the abbrev inside XEmacs; 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* when *hook* is called.

The use count of the abbrev is initialized to zero.

only-global-abbrevs User Option

If this variable is non-`nil`, it means that the user plans to use global abbrevs only. This tells the commands that define mode-specific abbrevs to define global ones instead. This variable does not alter the behavior of the functions in this section; it is examined by their callers.

39.4 Saving Abbrevs in Files

A file of saved abbrev definitions is actually a file of Lisp code. The abbrevs are saved in the form of a Lisp program to define the same abbrev tables with the same contents. Therefore, you can load the file with `load` (see Section 14.1 [How Programs Do Loading], page 195). 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 XEmacs 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 XEmacs commands to offer to save your abbrevs.

write-abbrev-file *filename* Command

Save all abbrev definitions, in all abbrev tables, in the file *filename*, in the form of a Lisp program that when loaded will define the same abbrevs. This function returns `nil`.

39.5 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. If *table* is `nil`, this function tries first the current buffer's local abbrev table, and second the global abbrev table.

abbrev-expansion *abbrev* &optional *table* — Function

This function returns the string that *abbrev* would expand into (as defined by the abbrev tables used for the current buffer). The optional argument *table* specifies the abbrev table to use, as in `abbrev-symbol`.

expand-abbrev — Command

This command expands the abbrev before point, if any. If point does not follow an abbrev, this command does nothing. The command returns `t` if it did expansion, `nil` otherwise.

abbrev-prefix-mark &optional *arg* — Command

Mark current point as the beginning of an abbrev. The next call to `expand-abbrev` will use the text from here to point (where it is then) as the abbrev to expand, rather than using the previous word as usual.

abbrev-all-caps — User Option

When this is set non-`nil`, an abbrev entered entirely in upper case is expanded using all upper case. Otherwise, an abbrev entered entirely in upper case is expanded by capitalizing each word of the expansion.

abbrev-start-location — Variable

This is the buffer position for `expand-abbrev` to use as the start of the next abbrev to be expanded. (`nil` means use the word before point instead.) `abbrev-start-location` is set to `nil` each time `expand-abbrev` is called. This variable is also set by `abbrev-prefix-mark`.

abbrev-start-location-buffer — Variable

The value of this variable is the buffer for which `abbrev-start-location` has been set. Trying to expand an abbrev in any other buffer clears `abbrev-start-location`. This variable is set by `abbrev-prefix-mark`.

last-abbrev — Variable

This is the `abbrev-symbol` of the 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, after case conversion (if any). Its value is `nil` if the abbrev has already been unexpanded. This contains information left by `expand-abbrev` for the sake of the `unexpand-abbrev` command.

pre-abbrev-expand-hook — Variable

This is a normal hook whose functions are executed, in sequence, just before any expansion of an abbrev. See Section 26.4 [Hooks], page 378. Since it is a normal hook, the hook functions receive no arguments. However, they can find the abbrev to be expanded by looking in the buffer before point.

The following sample code shows a simple use of `pre-abbrev-expand-hook`. If the user terminates an abbrev with a punctuation character, the hook function asks for confirmation. Thus, this hook allows the user to decide whether to expand the abbrev, and aborts expansion if it is not confirmed.

```
(add-hook 'pre-abbrev-expand-hook 'query-if-not-space)

;; This is the function invoked by pre-abbrev-expand-hook.

;; If the user terminated the abbrev with a space, the function does
;; nothing (that is, it returns so that the abbrev can expand).  If the
;; user entered some other character, this function asks whether
;; expansion should continue.

;; If the user answers the prompt with y, the function returns
;; nil (because of the not function), but that is
;; acceptable; the return value has no effect on expansion.

(defun query-if-not-space ()
  (if (/= ?\  (preceding-char))
      (if (not (y-or-n-p "Do you want to expand this abbrev? "))
          (error "Not expanding this abbrev"))))
```

39.6 Standard Abbrev Tables

Here we list the variables that hold the abbrev tables for the preloaded major modes of XEmacs.

global-abbrev-table Variable

This is the abbrev table for mode-independent abbrevs. The abbrevs defined in it apply to all buffers. Each buffer may also have a local abbrev table, whose abbrev definitions take precedence over those in the global table.

local-abbrev-table Variable

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

fundamental-mode-abbrev-table Variable

This is the local abbrev table used in Fundamental mode; in other words, it is the local abbrev table in all buffers in Fundamental mode.

text-mode-abbrev-table Variable

This is the local abbrev table used in Text mode.

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.

40 Extents

An *extent* is a region of text (a start position and an end position) that is displayed in a particular face and can have certain other properties such as being read-only. Extents can overlap each other. XEmacs efficiently handles buffers with large numbers of extents in them.

extentp *object* Function

This returns `t` if *object* is an extent.

40.1 Introduction to Extents

An extent is a region of text within a buffer or string that has certain properties associated with it. The properties of an extent primarily affect the way the text contained in the extent is displayed. Extents can freely overlap each other in a buffer or string. Extents are invisible to functions that merely examine the text of a buffer or string.

Please note: An alternative way to add properties to a buffer or string is to use text properties. See Section 36.18 [Text Properties], page 539.

An extent is logically a Lisp object consisting of a start position, an end position, a buffer or string to which these positions refer, and a property list. As text is inserted into a buffer, the start and end positions of the extent are automatically adjusted as necessary to keep the extent referring to the same text in the buffer. If text is inserted at the boundary of an extent, the extent's `start-open` and `end-open` properties control whether the text is included as part of the extent. If the text bounded by an extent is deleted, the extent becomes *detached*; its start and end positions are no longer meaningful, but it maintains all its other properties and can later be reinserted into a buffer. (None of these considerations apply to strings, because text cannot be inserted into or deleted from a string.)

Each extent has a face or list of faces associated with it, which controls the way in which the text bounded by the extent is displayed. If an extent's face is `nil` or its properties are partially undefined, the corresponding properties from the default face for the frame is used. If two or more extents overlap, or if a list of more than one face is specified for a particular extent, the corresponding faces are merged to determine the text's displayed properties. Every extent has a *priority* that determines which face takes precedence if the faces conflict. (If two extents have the same priority, the one that comes later in the display order takes precedence. See Section 40.3 [Extent Endpoints], page 586.) Higher-numbered priority values correspond to a higher priority, and priority values can be negative. Every extent is created with a priority of 0, but this can be changed with `set-extent-priority`. Within a single extent with a list of faces, faces earlier in the list have a higher priority than faces later in the list.

Extents can be set to respond specially to key and mouse events within the extent. An extent's `keymap` property controls the effect of key and mouse strokes within the extent's text, and the `mouse-face` property controls whether the extent is highlighted when the mouse moves over it. See Section 40.10 [Extents and Events], page 597.

An extent can optionally have a *begin-glyph* or *end-glyph* associated with it. A begin-glyph or end-glyph is a pixmap or string that will be displayed either at the start or end of an extent or in the margin of the line that the start or end of the extent lies in, depending on the extent's layout policy. Begin-glyphs and end-glyphs are used to implement annotations, and you should use the annotation API functions in preference to the lower-level extent functions. For more information, See Chapter 44 [Annotations], page 641.

If an extent has its `detachable` property set, it will become *detached* (i.e. no longer in the buffer) when all its text its deleted. Otherwise, it will simply shrink down to zero-length and sit it the same place in the buffer. By default, the `detachable` property is set on newly-created extents. See Section 40.7 [Detached Extents], page 595.

If an extent has its `duplicable` property set, it will be remembered when a string is created from text bounded by the extent. When the string is re-inserted into a buffer, the extent will also be re-inserted. This mechanism is used in the kill, yank, and undo commands. See Section 40.9 [Duplicable Extents], page 596.

40.2 Creating and Modifying Extents

make-extent *from to* &optional *object* — Function

This function makes an extent for the range [*from*, *to*) in *object* (a buffer or string). *object* defaults to the current buffer. Insertions at point *to* will be outside of the extent; insertions at *from* will be inside the extent, causing the extent to grow (see Section 40.3 [Extent Endpoints], page 586). This is the same way that markers behave. The extent is initially detached if both *from* and *to* are `nil`, and in this case *object* defaults to `nil`, meaning the extent is in no buffer or string (see Section 40.7 [Detached Extents], page 595).

delete-extent *extent* — Function

This function removes *extent* from its buffer and destroys it. This does not modify the buffer's text, only its display properties. The extent cannot be used thereafter. To remove an extent in such a way that it can be re-inserted later, use `detach-extent`. See Section 40.7 [Detached Extents], page 595.

extent-object *extent* — Function

This function returns the buffer or string that *extent* is in. If the return value is `nil`, this means that the extent is detached; however, a detached extent will not necessarily return a value of `nil`.

extent-live-p *extent* — Function

This function returns `nil` if *extent* is deleted, and `t` otherwise.

40.3 Extent Endpoints

Every extent has a start position and an end position, and logically affects the characters between those positions. Normally the start and end positions must both be valid positions in the extent's buffer or string. However, both endpoints can be `nil`, meaning the extent is detached. See Section 40.7 [Detached Extents], page 595.

Whether the extent overlaps its endpoints is governed by its `start-open` and `end-open` properties. Insertion of a character at a closed endpoint will expand the extent to include that character; insertion at an open endpoint will not. Similarly, functions such as `extent-at` that scan over all extents overlapping a particular position will include extents with a closed endpoint at that position, but not extents with an open endpoint.

Note that the `start-closed` and `end-closed` properties are equivalent to `start-open` and `end-open` with the opposite sense.

Both endpoints can be equal, in which case the extent includes no characters but still exists in the buffer or string. Zero-length extents are used to represent annotations (see Chapter 44 [Annotations], page 641) and can be used as a more powerful form of a marker. Deletion of all the characters in an extent may or may not result in a zero-length extent; this depends on the `detachable` property (see Section 40.7 [Detached Extents], page 595). Insertion at the position of a zero-length extent expands the extent if both endpoints are closed; goes before the extent if it has the `start-open` property; and goes after the extent if it has the `end-open` property. Zero-length extents with both the `start-open` and `end-open` properties are treated as if their starting point were closed. Deletion of a character on a side of a zero-length extent whose corresponding endpoint is closed causes the extent to be detached if its `detachable` property is set; if the corresponding endpoint is open, the extent remains in the buffer, moving as necessary.

Extents are ordered within a buffer or string by increasing start position, and then by decreasing end position (this is called the *display order*).

extent-start-position *extent* Function

This function returns the start position of *extent*.

extent-end-position *extent* Function

This function returns the end position of *extent*.

extent-length *extent* Function

This function returns the length of *extent* in characters. If the extent is detached, this returns `0`. If the extent is not detached, this is equivalent to

```
(- (extent-end-position extent) (extent-start-position extent))
```

set-extent-endpoints *extent start end* &optional *buffer-or-string* Function

This function sets the start and end position of *extent* to *start* and *end*. If both are `nil`, this is equivalent to `detach-extent`.

buffer-or-string specifies the new buffer or string that the extent should be in, and defaults to *extent*'s buffer or string. (If `nil`, and *extent* is in no buffer and no string, it defaults to the current buffer.)

See documentation on `detach-extent` for a discussion of undo recording.

40.4 Finding Extents

The following functions provide a simple way of determining the extents in a buffer or string. A number of more sophisticated primitives for mapping over the extents in a range of a buffer or string are also provided (see Section 40.5 [Mapping Over Extents], page 589). When reading through this section, keep in mind the way that extents are ordered (see Section 40.3 [Extent Endpoints], page 586).

extent-list &optional *buffer-or-string from to flags* Function

This function returns a list of the extents in *buffer-or-string*. *buffer-or-string* defaults to the current buffer if omitted. *from* and *to* can be used to limit the range over which extents are returned; if omitted, all extents in the buffer or string are returned.

More specifically, if a range is specified using *from* and *to*, only extents that overlap the range (i.e. begin or end inside of the range) are included in the list. *from* and *to* default to the beginning and end of *buffer-or-string*, respectively.

flags controls how end cases are treated. For a discussion of this, and exactly what "overlap" means, see `map-extents`.

Functions that create extents must be prepared for the possibility that there are other extents in the same area, created by other functions. To deal with this, functions typically mark their own extents by setting a particular property on them. The following function makes it easier to locate those extents.

extent-at *pos* &optional *object property before at-flag* Function

This function finds the "smallest" extent (i.e., the last one in the display order) at (i.e., overlapping) *pos* in *object* (a buffer or string) having *property* set. *object* defaults to the current buffer. *property* defaults to `nil`, meaning that any extent will do. Returns `nil` if there is no matching extent at *pos*. If the fourth argument *before* is not `nil`, it must be an extent; any returned extent will precede that extent. This feature allows `extent-at` to be used by a loop over extents.

at-flag controls how end cases are handled (i.e. what "at" really means), and should be one of:

`nil`

`after` An extent is at *pos* if it covers the character after *pos*. This is consistent with the way that text properties work.

`before` An extent is at *pos* if it covers the character before *pos*.

`at` An extent is at *pos* if it overlaps or abuts *pos*. This includes all zero-length extents at *pos*.

Note that in all cases, the start-openness and end-openness of the extents considered is ignored. If you want to pay attention to those properties, you should use `map-extents`, which gives you more control.

The following low-level functions are provided for explicitly traversing the extents in a buffer according to the display order. These functions are mostly intended for debugging – in normal operation, you should probably use `mapcar-extents` or `map-extents`, or loop using the *before* argument to `extent-at`, rather than creating a loop using `next-extent`.

next-extent *extent* Function

Given an extent *extent*, this function returns the next extent in the buffer or string's display order. If *extent* is a buffer or string, this returns the first extent in the buffer or string.

previous-extent *extent* Function

Given an extent *extent*, this function returns the previous extent in the buffer or string's display order. If *extent* is a buffer or string, this returns the last extent in the buffer or string.

40.5 Mapping Over Extents

The most basic and general function for mapping over extents is called `map-extents`. You should read through the definition of this function to familiarize yourself with the concepts and optional arguments involved. However, in practice you may find it more convenient to use the function `mapcar-extents` or to create a loop using the `before` argument to `extent-at` (see Section 40.4 [Finding Extents], page 587).

map-extents *function* &optional *object from to maparg flags property value* Function

This function maps *function* over the extents which overlap a region in *object*. *object* is normally a buffer or string but could be an extent (see below). The region is normally bounded by [*from*, *to*) (i.e. the beginning of the region is closed and the end of the region is open), but this can be changed with the *flags* argument (see below for a complete discussion).

function is called with the arguments (extent, *maparg*). The arguments *object*, *from*, *to*, *maparg*, and *flags* are all optional and default to the current buffer, the beginning of *object*, the end of *object*, *nil*, and *nil*, respectively. `map-extents` returns the first non-`nil` result produced by *function*, and no more calls to *function* are made after it returns non-`nil`.

If *object* is an extent, *from* and *to* default to the extent's endpoints, and the mapping omits that extent and its predecessors. This feature supports restarting a loop based on `map-extents`. Note: *object* must be attached to a buffer or string, and the mapping is done over that buffer or string.

An extent overlaps the region if there is any point in the extent that is also in the region. (For the purpose of overlap, zero-length extents and regions are treated as closed on both ends regardless of their endpoints' specified open/closedness.) Note that the endpoints of an extent or region are considered to be in that extent or region if and only if the corresponding end is closed. For example, the extent [5,7] overlaps the region [2,5] because 5 is in both the extent and the region. However, (5,7] does not overlap [2,5] because 5 is not in the extent, and neither [5,7] nor (5,7] overlaps the region [2,5) because 5 is not in the region.

The optional *flags* can be a symbol or a list of one or more symbols, modifying the behavior of `map-extents`. Allowed symbols are:

`end-closed`
: The region's end is closed.

`start-open`
: The region's start is open.

`all-extents-closed`
: Treat all extents as closed on both ends for the purpose of determining whether they overlap the region, irrespective of their actual open- or closedness.

`all-extents-open`
: Treat all extents as open on both ends.

`all-extents-closed-open`
: Treat all extents as start-closed, end-open.

`all-extents-open-closed`
: Treat all extents as start-open, end-closed.

`start-in-region`
: In addition to the above conditions for extent overlap, the extent's start position must lie within the specified region. Note that, for this condition, open start positions are treated as if 0.5 was added to the endpoint's value, and open end positions are treated as if 0.5 was subtracted from the endpoint's value.

`end-in-region`
: The extent's end position must lie within the region.

`start-and-end-in-region`
: Both the extent's start and end positions must lie within the region.

`start-or-end-in-region`
: Either the extent's start or end position must lie within the region.

`negate-in-region`
: The condition specified by a `*-in-region` flag must *not* hold for the extent to be considered.

At most one of `all-extents-closed`, `all-extents-open`, `all-extents-closed-open`, and `all-extents-open-closed` may be specified.

At most one of `start-in-region`, `end-in-region`, `start-and-end-in-region`, and `start-or-end-in-region` may be specified.

If optional arg *property* is non-`nil`, only extents with that property set on them will be visited. If optional arg *value* is non-`nil`, only extents whose value for that property is `eq` to *value* will be visited.

If you want to map over extents and accumulate a list of results, the following function may be more convenient than `map-extents`.

mapcar-extents *function* &optional *predicate buffer-or-string from to flags property value* — Function

This function applies *function* to all extents which overlap a region in *buffer-or-string*. The region is delimited by *from* and *to*. *function* is called with one argument, the extent. A list of the values returned by *function* is returned. An optional *predicate* may be used to further limit the extents over which *function* is mapped. The optional arguments *flags*, *property*, and *value* may also be used to control the extents passed to *predicate* or *function*, and have the same meaning as in `map-extents`.

map-extent-children *function* &optional *object from to maparg flags property value* — Function

This function is similar to `map-extents`, but differs in that:

- It only visits extents which start in the given region.
- After visiting an extent *e*, it skips all other extents which start inside *e* but end before *e*'s end.

Thus, this function may be used to walk a tree of extents in a buffer:

```
(defun walk-extents (buffer &optional ignore)
  (map-extent-children 'walk-extents buffer))
```

extent-in-region-p *extent* &optional *from to flags* — Function

This function returns *t* if `map-extents` would visit *extent* if called with the given arguments.

40.6 Properties of Extents

Each extent has a property list associating property names with values. Some property names have predefined meanings, and can usually only assume particular values. Assigning other values to such a property either cause the value to be converted into a legal value (e.g., assigning anything but `nil` to a Boolean property will cause the value of `t` to be assigned to the property) or will cause an error. Property names without predefined meanings can be assigned any value. An undefined property is equivalent to a property with a value of `nil`, or with a particular default value in the case of properties with predefined meanings. Note that, when an extent is created, the `end-open` and `detachable` properties are set on it.

If an extent has a parent, all of its properties actually derive from that parent (or from the root ancestor if the parent in turn has a parent), and setting a property of the extent actually sets that property on the parent. See Section 40.8 [Extent Parents], page 596.

extent-property *extent property* — Function

This function returns the value of *property* in *extent*. If *property* is undefined, `nil` is returned.

extent-properties *extent* — Function

This function returns a list of all of *extent*'s properties that do not have the value of `nil` (or the default value, for properties with predefined meanings).

set-extent-property *extent property value* — Function

This function sets *property* to *value* in *extent*. (If *property* has a predefined meaning, only certain values are allowed, and some values may be converted to others before being stored.)

set-extent-properties *extent plist* Function

Change some properties of *extent*. *plist* is a property list. This is useful to change many extent properties at once.

The following table lists the properties with predefined meanings, along with their allowable values.

`detached`
: (Boolean) Whether the extent is detached. Setting this is the same as calling `detach-extent`. See Section 40.7 [Detached Extents], page 595.

`destroyed`
: (Boolean) Whether the extent has been deleted. Setting this is the same as calling `delete-extent`.

`priority`
: (integer) The extent's redisplay priority. Defaults to 0. See Section 40.1 [Intro to Extents], page 585. This property can also be set with `set-extent-priority` and accessed with `extent-priority`.

`start-open`
: (Boolean) Whether the start position of the extent is open, meaning that characters inserted at that position go outside of the extent. See Section 40.3 [Extent Endpoints], page 586.

`start-closed`
: (Boolean) Same as `start-open` but with the opposite sense. Setting this property clears `start-open` and vice-versa.

`end-open`
: (Boolean) Whether the end position of the extent is open, meaning that characters inserted at that position go outside of the extent. This is `t` by default. See Section 40.3 [Extent Endpoints], page 586.

`end-closed`
: (Boolean) Same as `end-open` but with the opposite sense. Setting this property clears `end-open` and vice-versa.

`read-only`
: (Boolean) Whether text within this extent will be unmodifiable.

`face`
: (face, face name, list of faces or face names, or `nil`) The face in which to display the extent's text. This property can also be set with `set-extent-face` and accessed with `extent-face`. Note that if a list of faces is specified, the faces are merged together, with faces earlier in the list having priority over faces later in the list.

`mouse-face`
: (face, face name, list of faces or face names, or `nil`) The face used to display the extent when the mouse moves over it. This property can also be set with `set-extent-mouse-face` and accessed with `extent-mouse-face`. Note that if a list of faces is specified, the faces are merged together, with faces earlier in the list having priority over faces later in the list. See Section 40.10 [Extents and Events], page 597.

`pointer` (pointer glyph) The glyph used as the pointer when the mouse moves over the extent. This takes precedence over the `text-pointer-glyph` and `nontext-pointer-glyph` variables. If for any reason this glyph is an invalid pointer, the standard glyphs will be used as fallbacks. See Section 43.4 [Mouse Pointer], page 638.

`detachable`
(Boolean) Whether this extent becomes detached when all of the text it covers is deleted. This is `t` by default. See Section 40.7 [Detached Extents], page 595.

`duplicable`
(Boolean) Whether this extent should be copied into strings, so that kill, yank, and undo commands will restore or copy it. See Section 40.9 [Duplicable Extents], page 596.

`unique` (Boolean) Meaningful only in conjunction with `duplicable`. When this is set, there may be only one instance of this extent attached at a time. See Section 40.9 [Duplicable Extents], page 596.

`invisible`
(Boolean) If `t`, text under this extent will not be displayed – it will look as if the text is not there at all.

`keymap` (keymap or `nil`) This keymap is consulted for mouse clicks on this extent or keypresses made while `point` is within the extent. See Section 40.10 [Extents and Events], page 597.

`copy-function`
This is a hook that is run when a duplicable extent is about to be copied from a buffer to a string (or the kill ring). See Section 40.9 [Duplicable Extents], page 596.

`paste-function`
This is a hook that is run when a duplicable extent is about to be copied from a string (or the kill ring) into a buffer. See Section 40.9 [Duplicable Extents], page 596.

`begin-glyph`
(glyph or `nil`) This extent's begin glyph. See Chapter 44 [Annotations], page 641.

`end-glyph`
(glyph or `nil`) This extent's end glyph. See Chapter 44 [Annotations], page 641.

`begin-glyph-layout`
(`text`, `whitespace`, `inside-margin`, or `outside-margin`) The layout policy for this extent's begin glyph. Defaults to `text`. See Chapter 44 [Annotations], page 641.

`end-glyph-layout`
(`text`, `whitespace`, `inside-margin`, or `outside-margin`) The layout policy for this extent's end glyph. Defaults to `text`. See Chapter 44 [Annotations], page 641.

`initial-redisplay-function`
: (any funcallable object) The function to be called the first time (a part of) the extent is redisplayed. It will be called with the extent as its argument.

  This is used by `lazy-shot` to implement lazy font-locking. The functionality is still experimental, and may change without further notice.

The following convenience functions are provided for accessing particular properties of an extent.

extent-face *extent* — Function

This function returns the `face` property of *extent*. This might also return a list of face names. Do not modify this list directly! Instead, use `set-extent-face`.

Note that you can use `eq` to compare lists of faces as returned by `extent-face`. In other words, if you set the face of two different extents to two lists that are `equal` but not `eq`, then the return value of `extent-face` on the two extents will return the identical list.

extent-mouse-face *extent* — Function

This function returns the `mouse-face` property of *extent*. This might also return a list of face names. Do not modify this list directly! Instead, use `set-extent-mouse-face`.

Note that you can use `eq` to compare lists of faces as returned by `extent-mouse-face`, just like for `extent-face`.

extent-priority *extent* — Function

This function returns the `priority` property of *extent*.

extent-keymap *extent* — Function

This function returns the `keymap` property of *extent*.

extent-begin-glyph-layout *extent* — Function

This function returns the `begin-glyph-layout` property of *extent*, i.e. the layout policy associated with the *extent*'s begin glyph.

extent-end-glyph-layout *extent* — Function

This function returns the `end-glyph-layout` property of *extent*, i.e. the layout policy associated with the *extent*'s end glyph.

extent-begin-glyph *extent* — Function

This function returns the `begin-glyph` property of *extent*, i.e. the glyph object displayed at the beginning of *extent*. If there is none, `nil` is returned.

extent-end-glyph *extent* — Function

This function returns the `end-glyph` property of *extent*, i.e. the glyph object displayed at the end of *extent*. If there is none, `nil` is returned.

The following convenience functions are provided for setting particular properties of an extent.

set-extent-priority *extent pri* Function

This function sets the `priority` property of *extent* to *pri*.

set-extent-face *extent face* Function

This function sets the `face` property of *extent* to *face*.

set-extent-mouse-face *extent face* Function

This function sets the `mouse-face` property of *extent* to *face*.

set-extent-keymap *extent keymap* Function

This function sets the `keymap` property of *extent* to *keymap*. *keymap* must be either a keymap object, or `nil`.

set-extent-begin-glyph-layout *extent layout* Function

This function sets the `begin-glyph-layout` property of *extent* to *layout*.

set-extent-end-glyph-layout *extent layout* Function

This function sets the `end-glyph-layout` property of *extent* to *layout*.

set-extent-begin-glyph *extent begin-glyph* &optional *layout* Function

This function sets the `begin-glyph` and `glyph-layout` properties of *extent* to *begin-glyph* and *layout*, respectively. (*layout* defaults to `text` if not specified.)

set-extent-end-glyph *extent end-glyph* &optional *layout* Function

This function sets the `end-glyph` and `glyph-layout` properties of *extent* to *end-glyph* and *layout*, respectively. (*layout* defaults to `text` if not specified.)

set-extent-initial-redisplay-function *extent function* Function

This function sets the `initial-redisplay-function` property of the extent to *function*.

40.7 Detached Extents

A detached extent is an extent that is not attached to a buffer or string but can be re-inserted. Detached extents have a start position and end position of `nil`. Extents can be explicitly detached using `detach-extent`. An extent is also detached when all of its characters are all killed by a deletion, if its `detachable` property is set; if this property is not set, the extent becomes a zero-length extent. (Zero-length extents with the `detachable` property set behave specially. See Section 40.3 [Extent Endpoints], page 586.)

detach-extent *extent* Function

This function detaches *extent* from its buffer or string. If *extent* has the `duplicable` property, its detachment is tracked by the undo mechanism. See Section 40.9 [Duplicable Extents], page 596.

extent-detached-p *extent* Function

This function returns `nil` if *extent* is detached, and `t` otherwise.

copy-extent *extent* &optional *object* Function

This function makes a copy of *extent*. It is initially detached. Optional argument *object* defaults to *extent*'s object (normally a buffer or string, but could be `nil`).

insert-extent *extent* &optional *start end no-hooks object* Function

This function inserts *extent* from *start* to *end* in *object* (a buffer or string). If *extent* is detached from a different buffer or string, or in most cases when *extent* is already attached, the extent will first be copied as if with `copy-extent`. This function operates the same as if `insert` were called on a string whose extent data calls for *extent* to be inserted, except that if *no-hooks* is non-`nil`, *extent*'s `paste-function` will not be invoked. See Section 40.9 [Duplicable Extents], page 596.

40.8 Extent Parents

An extent can have a parent extent set for it. If this is the case, the extent derives all its properties from that extent and has no properties of its own. The only "properties" that the extent keeps are the buffer or string it refers to and the start and end points. (More correctly, the extent's own properties are shadowed. If you later change the extent to have no parent, its own properties will become visible again.)

It is possible for an extent's parent to itself have a parent, and so on. Through this, a whole tree of extents can be created, all deriving their properties from one root extent. Note, however, that you cannot create an inheritance loop – this is explicitly disallowed.

Parent extents are used to implement the extents over the modeline.

set-extent-parent *extent parent* Function

This function sets the parent of *extent* to *parent*. If *parent* is `nil`, the extent is set to have no parent.

extent-parent *extent* Function

This function return the parents (if any) of *extent*, or `nil`.

extent-children *extent* Function

This function returns a list of the children (if any) of *extent*. The children of an extent are all those extents whose parent is that extent. This function does not recursively trace children of children.

extent-descendants *extent* Function

This function returns a list of all descendants of *extent*, including *extent*. This recursively applies `extent-children` to any children of *extent*, until no more children can be found.

40.9 Duplicable Extents

If an extent has the `duplicable` property, it will be copied into strings, so that kill, yank, and undo commands will restore or copy it.

Specifically:

- When a string is created using `buffer-substring` or `buffer-string`, any duplicable extents in the region corresponding to the string will be copied into the string (see Section 36.2 [Buffer Contents], page 512). When the string in inserted into a buffer using `insert`, `insert-before-markers`, `insert-buffer` or `insert-buffer-substring`, the extents in the string will be copied back into the buffer (see Section 36.4 [Insertion], page 513). The extents in a string can, of course, be retrieved explicitly using the standard extent primitives over the string.
- Similarly, when text is copied or cut into the kill ring, any duplicable extents will be remembered and reinserted later when the text is pasted back into a buffer.
- When `concat` is called on strings, the extents in the strings are copied into the resulting string.
- When `substring` is called on a string, the relevant extents are copied into the resulting string.
- When a duplicable extent is detached by `detach-extent` or string deletion, or inserted by `insert-extent` or string insertion, the action is recorded by the undo mechanism so that it can be undone later. Note that if an extent gets detached and then a later undo causes the extent to get reinserted, the new extent will not be 'eq' to the original extent.
- Extent motion, face changes, and attachment via `make-extent` are not recorded by the undo mechanism. This means that extent changes which are to be undo-able must be performed by character editing, or by insertion and detachment of duplicable extents.
- A duplicable extent's `copy-function` property, if non-`nil`, should be a function, and will be run when a duplicable extent is about to be copied from a buffer to a string (or the kill ring). It is called with three arguments: the extent and the buffer positions within it which are being copied. If this function returns `nil`, then the extent will not be copied; otherwise it will.
- A duplicable extent's `paste-function` property, if non-`nil`, should be a function, and will be run when a duplicable extent is about to be copied from a string (or the kill ring) into a buffer. It is called with three arguments: the original extent and the buffer positions which the copied extent will occupy. (This hook is run after the corresponding text has already been inserted into the buffer.) Note that the extent argument may be detached when this function is run. If this function returns `nil`, no extent will be inserted. Otherwise, there will be an extent covering the range in question.

  Note: if the extent to be copied is already attached to the buffer and overlaps the new range, the extent will simply be extended and the `paste-function` will not be called.

40.10 Interaction of Extents with Keyboard and Mouse Events

If an extent has the `mouse-face` property set, it will be highlighted when the mouse passes over it. Highlighting is accomplished by merging the extent's face with the face or faces specified by the `mouse-face` property. The effect is as if a pseudo-extent with the `mouse-face` face were inserted after the extent in the display order (see Section 40.3 [Extent Endpoints], page 586, display order).

mouse-highlight-priority Variable

This variable holds the priority to use when merging in the highlighting pseudo-extent. The default is 1000. This is purposely set very high so that the highlighting pseudo-extent shows up even if there are other extents with various priorities at the same location.

You can also explicitly cause an extent to be highlighted. Only one extent at a time can be highlighted in this fashion, and any other highlighted extent will be de-highlighted.

highlight-extent *extent* &optional *highlight-p* Function

This function highlights (if *highlight-p* is non-`nil`) or de-highlights (if *highlight-p* is `nil`) *extent*, if *extent* has the `mouse-face` property. (Nothing happens if *extent* does not have the `mouse-face` property.)

force-highlight-extent *extent* &optional *highlight-p* Function

This function is similar to `highlight-extent` but highlights or de-highlights the extent regardless of whether it has the `mouse-face` property.

If an extent has a `keymap` property, this keymap will be consulted for mouse clicks on the extent and keypresses made while `point` is within the extent. The behavior of mouse clicks and keystrokes not defined in the keymap is as normal for the buffer.

40.11 Atomic Extents

If the Lisp file '`atomic-extents`' is loaded, then the atomic extent facility is available. An *atomic extent* is an extent for which `point` cannot be positioned anywhere within it. This ensures that when selecting text, either all or none of the extent is selected.

To make an extent atomic, set its `atomic` property.

41 Specifiers

A specifier is an object used to keep track of a property whose value may vary depending on the particular situation (e.g. particular buffer displayed in a particular window) that it is used in. The value of many built-in properties, such as the font, foreground, background, and such properties of a face and variables such as `modeline-shadow-thickness` and `top-toolbar-height`, is actually a specifier object. The specifier object, in turn, is "instanced" in a particular situation to yield the real value of the property in that situation.

specifierp *object* Function

This function returns non-`nil` if *object* is a specifier.

41.1 Introduction to Specifiers

Sometimes you may want the value of a property to vary depending on the context the property is used in. A simple example of this in XEmacs is buffer-local variables. For example, the variable `modeline-format`, which controls the format of the modeline, can have different values depending on the particular buffer being edited. The variable has a default value which most modes will use, but a specialized package such as Calendar might change the variable so as to tailor the modeline to its own purposes.

Other properties (such as those that can be changed by the `modify-frame-parameters` function, for example the color of the text cursor) can have frame-local values, although it might also make sense for them to have buffer-local values. In other cases, you might want the property to vary depending on the particular window within the frame that applies (e.g. the top or bottom window in a split frame), the device type that that frame appears on (X or tty), etc. Perhaps you can envision some more complicated scenario where you want a particular value in a specified buffer, another value in all other buffers displayed on a particular frame, another value in all other buffers displayed in all other frames on any mono (two-color, e.g. black and white only) displays, and a default value in all other circumstances.

A *specifier* is a generalization of this, allowing a great deal of flexibility in controlling exactly what value a property has in which circumstances. It is most commonly used for display properties, such as an image or the foreground color of a face. As a simple example, you can specify that the foreground of the default face be

- blue for a particular buffer
- green for all other buffers

As a more complicated example, you could specify that the foreground of the default face be

- forest green for all buffers displayed in a particular Emacs window, or green if the X server doesn't recognize the color '`forest green`'
- blue for all buffers displayed in a particular frame
- red for all other buffers displayed on a color device
- white for all other buffers

41.2 In-Depth Overview of a Specifier

A specifier object encapsulates a set of *specifications*, each of which says what its value should be if a particular condition applies. For example, one specification might be "The value should be darkseagreen2 on X devices" another might be "The value should be blue in the *Help* buffer". In specifier terminology, these conditions are called *locales* and the values are called *instantiators*. Given a specifier, a logical question is "What is its value in a particular situation?" This involves looking through the specifications to see which ones apply to this particular situation, and perhaps preferring one over another if more than one applies. In specifier terminology, a "particular situation" is called a *domain*, and determining its value in a particular domain is called *instancing*. Most of the time, a domain is identified by a particular window. For example, if the redisplay engine is drawing text in the default face in a particular window, it retrieves the specifier for the foreground color of the default face and *instances* it in the domain given by that window; in other words, it asks the specifier, "What is your value in this window?".

More specifically, a specifier contains a set of *specifications*, each of which associates a *locale* (a window object, a buffer object, a frame object, a device object, or the symbol `global`) with an *inst-list*, which is a list of one or more *inst-pairs*. (For each possible locale, there can be at most one specification containing that locale.) Each inst-pair is a cons of a *tag set* (an unordered list of zero or more symbols, or *tags*) and an *instantiator* (the allowed form of this varies depending on the type of specifier). In a given specification, there may be more than one inst-pair with the same tag set; this is unlike for locales.

The tag set is used to restrict the sorts of devices over which the instantiator is valid and to uniquely identify instantiators added by a particular application, so that different applications can work on the same specifier and not interfere with each other. Each tag can have a *predicate* associated with it, which is a function of one argument (a device) that specifies whether the tag matches that particular device. (If a tag does not have a predicate, it matches all devices.) All tags in a tag set must match a device for the associated inst-pair to be instantiable over that device. (A null tag set is perfectly valid.)

The valid device types (normally `x`, `tty`, and `stream`) and device classes (normally `color`, `grayscale`, and `mono`) can always be used as tags, and match devices of the associated type or class (see Chapter 33 [Consoles and Devices], page 483). User-defined tags may be defined, with an optional predicate specified. An application can create its own tag, use it to mark all its instantiators, and be fairly confident that it will not interfere with other applications that modify the same specifier – Functions that add a specification to a specifier usually only overwrite existing inst-pairs with the same tag set as was given, and a particular tag or tag set can be specified when removing instantiators.

When a specifier is instanced in a domain, both the locale and the tag set can be viewed as specifying necessary conditions that must apply in that domain for an instantiator to be considered as a possible result of the instancing. More specific locales always override more general locales (thus, there is no particular ordering of the specifications in a specifier); however, the tag sets are simply considered in the order that the inst-pairs occur in the specification's inst-list.

Note also that the actual object that results from the instancing (called an *instance object*) may not be the same as the instantiator from which it was derived. For some specifier types (such as integer specifiers and boolean specifiers), the instantiator will be returned directly as the instance object. For other types, however, this is not the case. For example, for font specifiers, the instantiator is a font-description string and the instance object is a font-instance object, which describes how the font is displayed on a particular device. A font-instance object encapsulates such things as the actual font name used to display the font on that device (a font-description string under X is usually a wildcard specification that may resolve to different font names, with possibly different foundries, widths, etc., on different devices), the extra properties of that font on that device, etc. Furthermore, this conversion (called *instantiation*) might fail – a font or color might not exist on a particular device, for example.

41.3 How a Specifier Is Instanced

Instancing of a specifier in a particular window domain proceeds as follows:

- First, XEmacs searches for a specification whose locale is the same as the window. If that fails, the search is repeated, looking for a locale that is the same as the window's buffer. If that fails, the search is repeated using the window's frame, then using the device that frame is on. Finally, the specification whose locale is the symbol `global` (if there is such a specification) is considered.
- The inst-pairs contained in the specification that was found are considered in their order in the inst-list, looking for one whose tag set matches the device that is derived from the window domain. (The tag set is an unordered list of zero or more tag symbols. For all tags that have predicates associated with them, the predicate must match the device.)
- If a matching tag set is found, the corresponding instantiator is passed to the specifier's instantiation method, which is specific to the type of the specifier. If it succeeds, the resulting instance object is returned as the result of the instancing and the instancing is done. Otherwise, the operation continues, looking for another matching inst-pair in the current specification.
- When there are no more inst-pairs to be considered in the current specification, the search starts over, looking for another specification as in the first step above.
- If all specifications are exhausted and no instance object can be derived, the instancing fails. (Actually, this is not completely true. Some specifier objects for built-in properties have a *fallback* value, which is either an inst-list or another specifier object, that is consulted if the instancing is about to fail. If it is an inst-list, the searching proceeds using the inst-pairs in that list. If it is a specifier, the entire instancing starts over using that specifier instead of the given one. Fallback values are set by the C code and cannot be modified, except perhaps indirectly, using any Lisp functions. The purpose of them is to supply some values to make sure that instancing of built-in properties can't fail and to implement some basic specifier inheritance, such as the fact that faces inherit their properties from the `default` face.)

It is also possible to instance a specifier over a frame domain or device domain instead of over a window domain. The C code, for example, instances the `top-toolbar-height` variable over a frame domain in order to determine the height of a frame's top toolbar.

Instancing over a frame or device is similar to instancing over a window except that specifications for locales that cannot be derived from the domain are ignored. Specifically, instancing over a frame looks first for frame locales, then device locales, then the `global` locale. Instancing over a device domain looks only for device locales and the `global` locale.

41.4 Specifier Types

There are various different types of specifiers. The type of a specifier controls what sorts of instantiators are valid, how an instantiator is instantiated, etc. Here is a list of built-in specifier types:

`boolean`
: The valid instantiators are the symbols `t` and `nil`. Instance objects are the same as instantiators so no special instantiation function is needed.

`integer`
: The valid instantiators are integers. Instance objects are the same as instantiators so no special instantiation function is needed. `modeline-shadow-thickness` is an example of an integer specifier (negative thicknesses indicate that the shadow is drawn recessed instead of raised).

`natnum`
: The valid instantiators are natnums (non-negative integers). Instance objects are the same as instantiators so no special instantiation function is needed. Natnum specifiers are used for dimension variables such as `top-toolbar-height`.

`generic`
: All Lisp objects are valid instantiators. Instance objects are the same as instantiators so no special instantiation function is needed.

`font`
: The valid instantiators are strings describing fonts or vectors indicating inheritance from the font of some face. Instance objects are font-instance objects, which are specific to a particular device. The instantiation method for font specifiers can fail, unlike for integer, natnum, boolean, and generic specifiers.

`color`
: The valid instantiators are strings describing colors or vectors indicating inheritance from the foreground or background of some face. Instance objects are color-instance objects, which are specific to a particular device. The instantiation method for color specifiers can fail, as for font specifiers.

`image`
: Images are perhaps the most complicated type of built-in specifier. The valid instantiators are strings (a filename, inline data for a pixmap, or text to be displayed in a text glyph) or vectors describing inline data of various sorts or indicating inheritance from the background-pixmap property of some face. Instance objects are either strings (for text images), image-instance objects (for pixmap images), or subwindow objects (for subwindow images). The instantiation method for image specifiers can fail, as for font and color specifiers.

`face-boolean`
: The valid instantiators are the symbols `t` and `nil` and vectors indicating inheritance from a boolean property of some face. Specifiers of this sort are used for all of the built-in boolean properties of faces. Instance objects are either the symbol `t` or the symbol `nil`.

`toolbar` The valid instantiators are toolbar descriptors, which are lists of toolbar-button descriptors (each of which is a vector of two or four elements). See Chapter 23 [Toolbar], page 351, for more information.

Color and font instance objects can also be used in turn as instantiators for a new color or font instance object. Since these instance objects are device-specific, the instantiator can be used directly as the new instance object, but only if they are of the same device. If the devices differ, the base color or font of the instantiating object is effectively used instead as the instantiator.

See Chapter 42 [Faces and Window-System Objects], page 615, for more information on fonts, colors, and face-boolean specifiers. See Chapter 43 [Glyphs], page 625, for more information about image specifiers. See Chapter 23 [Toolbar], page 351, for more information on toolbar specifiers.

specifier-type *specifier* Function

This function returns the type of *specifier*. The returned value will be a symbol: one of `integer`, `boolean`, etc., as listed in the above table.

Functions are also provided to query whether an object is a particular kind of specifier:

boolean-specifier-p *object* Function

This function returns non-`nil` if *object* is a boolean specifier.

integer-specifier-p *object* Function

This function returns non-`nil` if *object* is an integer specifier.

natnum-specifier-p *object* Function

This function returns non-`nil` if *object* is a natnum specifier.

generic-specifier-p *object* Function

This function returns non-`nil` if *object* is a generic specifier.

face-boolean-specifier-p *object* Function

This function returns non-`nil` if *object* is a face-boolean specifier.

toolbar-specifier-p *object* Function

This function returns non-`nil` if *object* is a toolbar specifier.

font-specifier-p *object* Function

This function returns non-`nil` if *object* is a font specifier.

color-specifier-p *object* Function

This function returns non-`nil` if *object* is a color specifier.

image-specifier-p *object* Function

This function returns non-`nil` if *object* is an image specifier.

41.5 Adding specifications to a Specifier

add-spec-to-specifier *specifier instantiator* &optional *locale tag-set how-to-add* Function

This function adds a specification to *specifier*. The specification maps from *locale* (which should be a window, buffer, frame, device, or the symbol `global`, and defaults to `global`) to *instantiator*, whose allowed values depend on the type of the specifier. Optional argument *tag-set* limits the instantiator to apply only to the specified tag set, which should be a list of tags all of which must match the device being instantiated over (tags are a device type, a device class, or tags defined with `define-specifier-tag`). Specifying a single symbol for *tag-set* is equivalent to specifying a one-element list containing that symbol. Optional argument *how-to-add* specifies what to do if there are already specifications in the specifier. It should be one of

`prepend` Put at the beginning of the current list of instantiators for *locale*.

`append` Add to the end of the current list of instantiators for *locale*.

`remove-tag-set-prepend`
This is the default. Remove any existing instantiators whose tag set is the same as *tag-set*; then put the new instantiator at the beginning of the current list.

`remove-tag-set-append`
Remove any existing instantiators whose tag set is the same as *tag-set*; then put the new instantiator at the end of the current list.

`remove-locale`
Remove all previous instantiators for this locale before adding the new spec.

`remove-locale-type`
Remove all specifications for all locales of the same type as *locale* (this includes *locale* itself) before adding the new spec.

`remove-all`
Remove all specifications from the specifier before adding the new spec.

`remove-tag-set-prepend` is the default.

You can retrieve the specifications for a particular locale or locale type with the function `specifier-spec-list` or `specifier-specs`.

add-spec-list-to-specifier *specifier spec-list* &optional *how-to-add* Function

This function adds a *spec-list* (a list of specifications) to *specifier*. The format of a spec-list is

((*locale* (*tag-set* . *instantiator*) ...) ...)

where

- *locale* := a window, a buffer, a frame, a device, or `global`
- *tag-set* := an unordered list of zero or more *tags*, each of which is a symbol
- *tag* := a device class (see Chapter 33 [Consoles and Devices], page 483), a device type, or a tag defined with `define-specifier-tag`
- *instantiator* := format determined by the type of specifier

The pair `(`*tag-set* `.` *instantiator*`)` is called an *inst-pair*. A list of inst-pairs is called an *inst-list*. The pair `(`*locale* `.` *inst-list*`)` is called a *specification*. A spec-list, then, can be viewed as a list of specifications.

how-to-add specifies how to combine the new specifications with the existing ones, and has the same semantics as for `add-spec-to-specifier`.

In many circumstances, the higher-level function `set-specifier` is more convenient and should be used instead.

let-specifier *specifier-list* &rest *body* Macro

This special form temporarily adds specifications to specifiers, evaluates forms in *body* and restores the specifiers to their previous states. The specifiers and their temporary specifications are listed in *specifier-list*.

The format of *specifier-list* is

```
((specifier value &optional locale tag-set how-to-add) ...)
```

specifier is the specifier to be temporarily modified. *value* is the instantiator to be temporarily added to specifier in *locale*. *locale*, *tag-set* and *how-to-add* have the same meaning as in `add-spec-to-specifier`.

This special form is implemented as a macro; the code resulting from macro expansion will add specifications to specifiers using `add-spec-to-specifier`. After forms in *body* are evaluated, the temporary specifications are removed and old specifier spec-lists are restored.

locale, *tag-set* and *how-to-add* may be omitted, and default to `nil`. The value of the last form in *body* is returned.

NOTE: If you want the specifier's instance to change in all circumstances, use `(selected-window)` as the *locale*. If *locale* is `nil` or omitted, it defaults to `global`.

The following example removes the 3D modeline effect in the currently selected window for the duration of a second:

```
(let-specifier ((modeline-shadow-thickness 0 (selected-window)))
  (sit-for 1))
```

set-specifier *specifier value* &optional *how-to-add* Function

This function adds some specifications to *specifier*. *value* can be a single instantiator or tagged instantiator (added as a global specification), a list of tagged and/or untagged instantiators (added as a global specification), a cons of a locale and instantiator or locale and instantiator list, a list of such conses, or nearly any other reasonable form. More specifically, *value* can be anything accepted by `canonicalize-spec-list`.

how-to-add is the same as in `add-spec-to-specifier`.

Note that `set-specifier` is exactly complementary to `specifier-specs` except in the case where *specifier* has no specs at all in it but `nil` is a valid instantiator (in that case, `specifier-specs` will return `nil` (meaning no specs) and `set-specifier` will interpret the `nil` as meaning "I'm adding a global instantiator and its value is `nil`"), or in strange cases where there is an ambiguity between a spec-list and an inst-list, etc. (The built-in specifier types are designed in such a way as to avoid any such ambiguities.)

If you want to work with spec-lists, you should probably not use these functions, but should use the lower-level functions `specifier-spec-list` and `add-spec-list-to-specifier`. These functions always work with fully-qualified spec-lists; thus, there is no ambiguity.

canonicalize-inst-pair *inst-pair specifier-type* &optional *noerror* Function

This function canonicalizes the given *inst-pair*.

specifier-type specifies the type of specifier that this *spec-list* will be used for.

Canonicalizing means converting to the full form for an inst-pair, i.e. `(`*tag-set* `.` *instantiator*`)`. A single, untagged instantiator is given a tag set of `nil` (the empty set), and a single tag is converted into a tag set consisting only of that tag.

If *noerror* is non-`nil`, signal an error if the inst-pair is invalid; otherwise return `t`.

canonicalize-inst-list *inst-list specifier-type* &optional *noerror* Function

This function canonicalizes the given *inst-list* (a list of inst-pairs).

specifier-type specifies the type of specifier that this *inst-list* will be used for.

Canonicalizing means converting to the full form for an inst-list, i.e. `((`*tag-set* `.` *instantiator*`) ...)`. This function accepts a single inst-pair or any abbreviation thereof or a list of (possibly abbreviated) inst-pairs. (See `canonicalize-inst-pair`.)

If *noerror* is non-`nil`, signal an error if the inst-list is invalid; otherwise return `t`.

canonicalize-spec *spec specifier-type* &optional *noerror* Function

This function canonicalizes the given *spec* (a specification).

specifier-type specifies the type of specifier that this *spec-list* will be used for.

Canonicalizing means converting to the full form for a spec, i.e. `(`*locale* `(`*tag-set* `.` *instantiator*`) ...)`. This function accepts a possibly abbreviated inst-list or a cons of a locale and a possibly abbreviated inst-list. (See `canonicalize-inst-list`.)

If *noerror* is `nil`, signal an error if the specification is invalid; otherwise return `t`.

canonicalize-spec-list *spec-list specifier-type* &optional *noerror* Function

This function canonicalizes the given *spec-list* (a list of specifications).

specifier-type specifies the type of specifier that this *spec-list* will be used for.

Canonicalizing means converting to the full form for a spec-list, i.e. `((`*locale* `(`*tag-set* `.` *instantiator*`) ...) ...)`. This function accepts a possibly abbreviated specification or a list of such things. (See `canonicalize-spec`.) This is the function used to convert spec-lists accepted by `set-specifier` and such into a form suitable for `add-spec-list-to-specifier`.

This function tries extremely hard to resolve any ambiguities, and the built-in specifier types (font, image, toolbar, etc.) are designed so that there won't be any ambiguities.

If *noerror* is `nil`, signal an error if the spec-list is invalid; otherwise return `t`.

41.6 Retrieving the Specifications from a Specifier

specifier-spec-list *specifier* &optional *locale tag-set exact-p* Function

This function returns the spec-list of specifications for *specifier* in *locale*.

If *locale* is a particular locale (a window, buffer, frame, device, or the symbol `global`), a spec-list consisting of the specification for that locale will be returned.

If *locale* is a locale type (i.e. a symbol `window`, `buffer`, `frame`, or `device`), a spec-list of the specifications for all locales of that type will be returned.

If *locale* is `nil` or the symbol `all`, a spec-list of all specifications in *specifier* will be returned.

locale can also be a list of locales, locale types, and/or `all`; the result is as if `specifier-spec-list` were called on each element of the list and the results concatenated together.

Only instantiators where *tag-set* (a list of zero or more tags) is a subset of (or possibly equal to) the instantiator's tag set are returned. (The default value of `nil` is a subset of all tag sets, so in this case no instantiators will be screened out.) If *exact-p* is non-`nil`, however, *tag-set* must be equal to an instantiator's tag set for the instantiator to be returned.

specifier-specs *specifier* &optional *locale tag-set exact-p* Function

This function returns the specification(s) for *specifier* in *locale*.

If *locale* is a single locale or is a list of one element containing a single locale, then a "short form" of the instantiators for that locale will be returned. Otherwise, this function is identical to `specifier-spec-list`.

The "short form" is designed for readability and not for ease of use in Lisp programs, and is as follows:

1. If there is only one instantiator, then an inst-pair (i.e. cons of tag and instantiator) will be returned; otherwise a list of inst-pairs will be returned.
2. For each inst-pair returned, if the instantiator's tag is `any`, the tag will be removed and the instantiator itself will be returned instead of the inst-pair.
3. If there is only one instantiator, its value is `nil`, and its tag is `any`, a one-element list containing `nil` will be returned rather than just `nil`, to distinguish this case from there being no instantiators at all.

specifier-fallback *specifier* Function

This function returns the fallback value for *specifier*. Fallback values are provided by the C code for certain built-in specifiers to make sure that instancing won't fail even if all specs are removed from the specifier, or to implement simple inheritance behavior (e.g. this method is used to ensure that faces other than `default` inherit their attributes from `default`). By design, you cannot change the fallback value, and specifiers created with `make-specifier` will never have a fallback (although a similar, Lisp-accessible capability may be provided in the future to allow for inheritance).

The fallback value will be an inst-list that is instanced like any other inst-list, a specifier of the same type as *specifier* (results in inheritance), or `nil` for no fallback.

When you instance a specifier, you can explicitly request that the fallback not be consulted. (The C code does this, for example, when merging faces.) See `specifier-instance`.

41.7 Working With Specifier Tags

A specifier tag set is an entity that is attached to an instantiator and can be used to restrict the scope of that instantiator to a particular device class or device type and/or to mark instantiators added by a particular package so that they can be later removed.

A specifier tag set consists of a list of zero of more specifier tags, each of which is a symbol that is recognized by XEmacs as a tag. (The valid device types and device classes are always tags, as are any tags defined by `define-specifier-tag`.) It is called a "tag set" (as opposed to a list) because the order of the tags or the number of times a particular tag occurs does not matter.

Each tag has a predicate associated with it, which specifies whether that tag applies to a particular device. The tags which are device types and classes match devices of that type or class. User-defined tags can have any predicate, or none (meaning that all devices match). When attempting to instance a specifier, a particular instantiator is only considered if the device of the domain being instanced over matches all tags in the tag set attached to that instantiator.

Most of the time, a tag set is not specified, and the instantiator gets a null tag set, which matches all devices.

valid-specifier-tag-p *tag* Function

This function returns non-`nil` if *tag* is a valid specifier tag.

valid-specifier-tag-set-p *tag-set* Function

This function returns non-`nil` if *tag-set* is a valid specifier tag set.

canonicalize-tag-set *tag-set* Function

This function canonicalizes the given tag set. Two canonicalized tag sets can be compared with `equal` to see if they represent the same tag set. (Specifically, canonicalizing involves sorting by symbol name and removing duplicates.)

device-matches-specifier-tag-set-p *device tag-set* Function

This function returns non-`nil` if *device* matches specifier tag set *tag-set*. This means that *device* matches each tag in the tag set.

define-specifier-tag *tag* &optional *predicate* Function

This function defines a new specifier tag. If *predicate* is specified, it should be a function of one argument (a device) that specifies whether the tag matches that particular device. If *predicate* is omitted, the tag matches all devices.

You can redefine an existing user-defined specifier tag. However, you cannot redefine the built-in specifier tags (the device types and classes) or the symbols `nil`, `t`, `all`, or `global`.

device-matching-specifier-tag-list &optional *device* Function

This function returns a list of all specifier tags matching *device*. *device* defaults to the selected device if omitted.

specifier-tag-list Function

This function returns a list of all currently-defined specifier tags. This includes the built-in ones (the device types and classes).

specifier-tag-predicate *tag* Function

This function returns the predicate for the given specifier tag.

41.8 Functions for Instancing a Specifier

specifier-instance *specifier* &optional *domain default no-fallback* Function

This function instantiates *specifier* (return its value) in *domain*. If no instance can be generated for this domain, return *default*.

domain should be a window, frame, or device. Other values that are legal as a locale (e.g. a buffer) are not valid as a domain because they do not provide enough information to identify a particular device (see `valid-specifier-domain-p`). *domain* defaults to the selected window if omitted.

Instantiating a specifier in a particular domain means determining the specifier's "value" in that domain. This is accomplished by searching through the specifications in the specifier that correspond to all locales that can be derived from the given domain, from specific to general. In most cases, the domain is an Emacs window. In that case specifications are searched for as follows:

1. A specification whose locale is the window itself;
2. A specification whose locale is the window's buffer;
3. A specification whose locale is the window's frame;
4. A specification whose locale is the window's frame's device;
5. A specification whose locale is the symbol `global`.

If all of those fail, then the C-code-provided fallback value for this specifier is consulted (see `specifier-fallback`). If it is an inst-list, then this function attempts to instantiate that list just as when a specification is located in the first five steps above. If the fallback is a specifier, `specifier-instance` is called recursively on this specifier and the return value used. Note, however, that if the optional argument *no-fallback* is non-`nil`, the fallback value will not be consulted.

Note that there may be more than one specification matching a particular locale; all such specifications are considered before looking for any specifications for more general locales. Any particular specification that is found may be rejected because it is tagged to a particular device class (e.g. `color`) or device type (e.g. `x`) or both and the device for the given domain does not match this, or because the specification is not valid for the device of the given domain (e.g. the font or color name does not exist for this particular X server).

The returned value is dependent on the type of specifier. For example, for a font specifier (as returned by the `face-font` function), the returned value will be a font-instance object. For images, the returned value will be a string, pixmap, or subwindow.

specifier-instance-from-inst-list *specifier domain inst-list* &optional *default* — Function

This function attempts to convert a particular inst-list into an instance. This attempts to instantiate *inst-list* in the given *domain*, as if *inst-list* existed in a specification in *specifier*. If the instantiation fails, *default* is returned. In most circumstances, you should not use this function; use `specifier-instance` instead.

41.9 Example of Specifier Usage

Now let us present an example to clarify the theoretical discussions we have been through. In this example, we will use the general specifier functions for clarity. Keep in mind that many types of specifiers, and some other types of objects that are associated with specifiers (e.g. faces), provide convenience functions making it easier to work with objects of that type.

Let us consider the background color of the default face. A specifier is used to specify how that color will appear in different domains. First, let's retrieve the specifier:

```
(setq sp (face-property 'default 'background))
    ⇒   #<color-specifier 0x3da>

(specifier-specs sp)
    ⇒   ((#<buffer "device.c"> (nil . "forest green"))
                 (#<window on "Makefile" 0x8a2b> (nil . "hot pink"))
                 (#<x-frame "emacs" 0x4ac> (nil . "puke orange")
   (nil . "moccasin"))
                 (#<x-frame "VM" 0x4ac> (nil . "magenta"))
 (global ((tty) . "cyan") (nil . "white"))
                )
```

Then, say we want to determine what the background color of the default face is for the window currently displaying the buffer '`*scratch*`'. We call

```
(get-buffer-window "*scratch*")
    ⇒ #<window on "*scratch*" 0x4ad>
(window-frame (get-buffer-window "*scratch*"))
    ⇒ #<x-frame "emacs" 0x4ac>
(specifier-instance sp (get-buffer-window "*scratch*"))
    ⇒ #<color-instance moccasin 47=(FFFF,E4E4,B5B5) 0x6309>
```

Note that we passed a window to `specifier-instance`, not a buffer. We cannot pass a buffer because a buffer by itself does not provide enough information. The buffer might not be displayed anywhere at all, or could be displayed in many different frames on different devices.

The result is arrived at like this:

1. First, we look for a specification matching the buffer displayed in the window, i.e. '`*scratch`'. There are none, so we proceed.
2. Then, we look for a specification matching the window itself. Again, there are none.
3. Then, we look for a specification matching the window's frame. The specification `(#<x-frame "emacs" 0x4ac> . "puke orange")` is found. We call the instantiation method for colors, passing it the locale we were searching over (i.e. the window, in this case) and the instantiator ('`"puke orange"`'). However, the particular device which this window is on (let's say it's an X connection) doesn't recognize the color '`"puke orange"`', so the specification is rejected.
4. So we continue looking for a specification matching the window's frame. We find '`(#<x-frame "emacs" 0x4ac> . "moccasin")`'. Again, we call the instantiation method for colors. This time, the X server our window is on recognizes the color '`moccasin`', and so the instantiation method succeeds and returns a color instance.

41.10 Creating New Specifier Objects

make-specifier *type* Function

This function creates a new specifier.

A specifier is an object that can be used to keep track of a property whose value can be per-buffer, per-window, per-frame, or per-device, and can further be restricted to a particular device-type or device-class. Specifiers are used, for example, for the various built-in properties of a face; this allows a face to have different values in different frames, buffers, etc. For more information, see 'specifier-instance', 'specifier-specs', and 'add-spec-to-specifier'; or, for a detailed description of specifiers, including how they are instantiated over a particular domain (i.e. how their value in that domain is determined), see the chapter on specifiers in the XEmacs Lisp Reference Manual.

type specifies the particular type of specifier, and should be one of the symbols `generic`, `integer`, `natnum`, `boolean`, `color`, `font`, `image`, `face-boolean`, or `toolbar`.

For more information on particular types of specifiers, see the functions `generic-specifier-p`, `integer-specifier-p`, `natnum-specifier-p`, `boolean-specifier-p`, `color-specifier-p`, `font-specifier-p`, `image-specifier-p`, `face-boolean-specifier-p`, and `toolbar-specifier-p`.

make-specifier-and-init *type spec-list* &optional *dont-canonicalize* — Function

This function creates and initialize a new specifier.

This is a front-end onto `make-specifier` that allows you to create a specifier and add specs to it at the same time. *type* specifies the specifier type. *spec-list* supplies the specification(s) to be added to the specifier. Normally, almost any reasonable abbreviation of the full spec-list form is accepted, and is converted to the full form; however, if optional argument *dont-canonicalize* is non-`nil`, this conversion is not performed, and the *spec-list* must already be in full form. See `canonicalize-spec-list`.

41.11 Functions for Checking the Validity of Specifier Components

valid-specifier-domain-p *domain* — Function

This function returns non-`nil` if *domain* is a valid specifier domain. A domain is used to instance a specifier (i.e. determine the specifier's value in that domain). Valid domains are a window, frame, or device. (`nil` is not valid.)

valid-specifier-locale-p *locale* — Function

This function returns non-`nil` if *locale* is a valid specifier locale. Valid locales are a device, a frame, a window, a buffer, and `global`. (`nil` is not valid.)

valid-specifier-locale-type-p *locale-type* — Function

Given a specifier *locale-type*, this function returns non-nil if it is valid. Valid locale types are the symbols `global`, `device`, `frame`, `window`, and `buffer`. (Note, however, that in functions that accept either a locale or a locale type, `global` is considered an individual locale.)

valid-specifier-type-p *specifier-type* — Function

Given a *specifier-type*, this function returns non-`nil` if it is valid. Valid types are `generic`, `integer`, `boolean`, `color`, `font`, `image`, `face-boolean`, and `toolbar`.

valid-specifier-tag-p *tag* — Function

This function returns non-`nil` if *tag* is a valid specifier tag.

valid-instantiator-p *instantiator specifier-type* — Function

This function returns non-`nil` if *instantiator* is valid for *specifier-type*.

valid-inst-list-p *inst-list type* — Function

This function returns non-`nil` if *inst-list* is valid for specifier type *type*.

valid-spec-list-p *spec-list type* — Function

This function returns non-`nil` if *spec-list* is valid for specifier type *type*.

check-valid-instantiator *instantiator specifier-type* — Function

This function signals an error if *instantiator* is invalid for *specifier-type*.

check-valid-inst-list *inst-list type* Function

This function signals an error if *inst-list* is invalid for specifier type *type*.

check-valid-spec-list *spec-list type* Function

This function signals an error if *spec-list* is invalid for specifier type *type*.

41.12 Other Functions for Working with Specifications in a Specifier

copy-specifier *specifier* &optional *dest locale tag-set exact-p how-to-add* Function

This function copies *specifier* to *dest*, or creates a new one if *dest* is `nil`.

If *dest* is `nil` or omitted, a new specifier will be created and the specifications copied into it. Otherwise, the specifications will be copied into the existing specifier in *dest*.

If *locale* is `nil` or the symbol `all`, all specifications will be copied. If *locale* is a particular locale, the specification for that particular locale will be copied. If *locale* is a locale type, the specifications for all locales of that type will be copied. *locale* can also be a list of locales, locale types, and/or `all`; this is equivalent to calling `copy-specifier` for each of the elements of the list. See `specifier-spec-list` for more information about *locale*.

Only instantiators where *tag-set* (a list of zero or more tags) is a subset of (or possibly equal to) the instantiator's tag set are copied. (The default value of `nil` is a subset of all tag sets, so in this case no instantiators will be screened out.) If *exact-p* is non-`nil`, however, *tag-set* must be equal to an instantiator's tag set for the instantiator to be copied.

Optional argument *how-to-add* specifies what to do with existing specifications in *dest*. If nil, then whichever locales or locale types are copied will first be completely erased in *dest*. Otherwise, it is the same as in `add-spec-to-specifier`.

remove-specifier *specifier* &optional *locale tag-set exact-p* Function

This function removes specification(s) for *specifier*.

If *locale* is a particular locale (a buffer, window, frame, device, or the symbol `global`), the specification for that locale will be removed.

If instead, *locale* is a locale type (i.e. a symbol `buffer`, `window`, `frame`, or `device`), the specifications for all locales of that type will be removed.

If *locale* is `nil` or the symbol `all`, all specifications will be removed.

locale can also be a list of locales, locale types, and/or `all`; this is equivalent to calling `remove-specifier` for each of the elements in the list.

Only instantiators where *tag-set* (a list of zero or more tags) is a subset of (or possibly equal to) the instantiator's tag set are removed. (The default value of `nil` is a subset of all tag sets, so in this case no instantiators will be screened out.) If *exact-p* is non-`nil`, however, *tag-set* must be equal to an instantiator's tag set for the instantiator to be removed.

map-specifier *specifier func* &optional *locale maparg* Function

This function applies *func* to the specification(s) for *locale* in *specifier*.

If *locale* is a locale, *func* will be called for that locale. If *locale* is a locale type, *func* will be mapped over all locales of that type. If *locale* is `nil` or the symbol `all`, *func* will be mapped over all locales in *specifier*.

func is called with four arguments: the *specifier*, the locale being mapped over, the inst-list for that locale, and the optional *maparg*. If any invocation of *func* returns non-`nil`, the mapping will stop and the returned value becomes the value returned from `map-specifier`. Otherwise, `map-specifier` returns `nil`.

specifier-locale-type-from-locale *locale* Function

Given a specifier *locale*, this function returns its type.

42 Faces and Window-System Objects

42.1 Faces

A *face* is a named collection of graphical properties: font, foreground color, background color, background pixmap, optional underlining, and (on TTY devices) whether the text is to be highlighted, dimmed, blinking, or displayed in reverse video. Faces control the display of text on the screen. Every face has a name, which is a symbol such as `default` or `modeline`.

Each built-in property of a face is controlled using a specifier, which allows it to have separate values in particular buffers, frames, windows, and devices and to further vary according to device type (X or TTY) and device class (color, mono, or grayscale). See Chapter 41 [Specifiers], page 599, for more information.

The face named `default` is used for ordinary text. The face named `modeline` is used for displaying the modeline. The face named `highlight` is used for highlighted extents (see Chapter 40 [Extents], page 585). The faces named `left-margin` and `right-margin` are used for the left and right margin areas, respectively (see Chapter 44 [Annotations], page 641). The face named `zmacs-region` is used for the highlighted region between point and mark.

42.1.1 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*, which is used for all text that doesn't somehow specify another face. The face named `default` applies to the text area, while the faces `left-margin` and `right-margin` apply to the left and right margin areas.
- With text properties. A character may have a `face` property; if so, it's displayed with that face. (Text properties are actually implemented in terms of extents.) See Section 36.18 [Text Properties], page 539.
- With extents. An extent may have a `face` property, which applies to all the text covered by the extent; in addition, if the `highlight` property is set, the `highlight` property applies when the mouse moves over the extent or if the extent is explicitly highlighted. See Chapter 40 [Extents], page 585.
- With annotations. Annotations that are inserted into a buffer can specify their own face. (Annotations are actually implemented in terms of extents.) See Chapter 44 [Annotations], page 641.

If these various sources together specify more than one face for a particular character, XEmacs merges the properties of the various faces specified. Extents, text properties, and annotations all use the same underlying representation (as extents). When multiple extents cover one character, an extent with higher priority overrides those with lower priority. See Chapter 40 [Extents], page 585. If no extent covers a particular character, the `default` face is used.

If a background pixmap is specified, it determines what will be displayed in the background of text characters. If the background pixmap is actually a pixmap, with its colors specified, those colors are used; if it is a bitmap, the face's foreground and background colors are used to color it.

42.1.2 Basic Functions for Working with Faces

The properties a face can specify include the font, the foreground color, the background color, the background pixmap, the underlining, the display table, and (for TTY devices) whether the text is to be highlighted, dimmed, blinking, or displayed in reverse video. The face can also leave these unspecified, causing them to assume the value of the corresponding property of the `default` face.

Here are the basic primitives for working with faces.

make-face *name* &optional *doc-string temporary* Function

This function defines and returns a new face named *name*, initially with all properties unspecified. It does nothing if there is already a face named *name*. Optional argument *doc-string* specifies an explanatory string used for descriptive purposes. If optional argument *temporary* is non-`nil`, the face will automatically disappear when there are no more references to it anywhere in text or Lisp code (otherwise, the face will continue to exist indefinitely even if it is not used).

face-list &optional *temporary* Function

This function returns a list of the names of all defined faces. If *temporary* is `nil`, only the permanent faces are included. If it is `t`, only the temporary faces are included. If it is any other non-`nil` value both permanent and temporary are included.

facep *object* Function

This function returns whether the given object is a face.

copy-face *old-face new-name* &optional *locale how-to-add* 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 properties as *old-face*. *locale* and *how-to-add* let you copy just parts of the old face rather than the whole face, and are as in `copy-specifier` (see Chapter 41 [Specifiers], page 599).

42.1.3 Face Properties

You can examine and modify the properties of an existing face with the following functions.

The following symbols have predefined meanings:

`foreground`
: The foreground color of the face.

`background`
: The background color of the face.

`font` The font used to display text covered by this face.

`display-table`
The display table of the face.

`background-pixmap`
The pixmap displayed in the background of the face. Only used by faces on X devices.

`underline`
Underline all text covered by this face.

`highlight`
Highlight all text covered by this face. Only used by faces on TTY devices.

`dim` Dim all text covered by this face. Only used by faces on TTY devices.

`blinking` Blink all text covered by this face. Only used by faces on TTY devices.

`reverse` Reverse the foreground and background colors. Only used by faces on TTY devices.

`doc-string`
Description of what the face's normal use is. NOTE: This is not a specifier, unlike all the other built-in properties, and cannot contain locale-specific values.

set-face-property *face property value* &optional *locale tag how-to-add* Function

This function changes a property of a *face*.

For built-in properties, the actual value of the property is a specifier and you cannot change this; but you can change the specifications within the specifier, and that is what this function will do. For user-defined properties, you can use this function to either change the actual value of the property or, if this value is a specifier, change the specifications within it.

If *property* is a built-in property, the specifications to be added to this property can be supplied in many different ways:

> If *value* is a simple instantiator (e.g. a string naming a font or color) or a list of instantiators, then the instantiator(s) will be added as a specification of the property for the given *locale* (which defaults to `global` if omitted).
>
> If *value* is a list of specifications (each of which is a cons of a locale and a list of instantiators), then *locale* must be `nil` (it does not make sense to explicitly specify a locale in this case), and specifications will be added as given.
>
> If *value* is a specifier (as would be returned by `face-property` if no *locale* argument is given), then some or all of the specifications in the specifier will be added to the property. In this case, the function is really equivalent to `copy-specifier` and *locale* has the same semantics (if it is a particular locale, the specification for the locale will be copied; if a locale type, specifications for all locales of that type will be copied; if `nil` or `all`, then all specifications will be copied).

how-to-add should be either `nil` or one of the symbols `prepend`, `append`, `remove-tag-set-prepend`, `remove-tag-set-append`, `remove-locale`, `remove-locale-type`, or `remove-all`. See `copy-specifier` and `add-spec-to-specifier` for a description of what each of these means. Most of the time, you do not need to worry about this argument; the default behavior usually is fine.

In general, it is OK to pass an instance object (e.g. as returned by `face-property-instance`) as an instantiator in place of an actual instantiator. In such a case, the instantiator used to create that instance object will be used (for example, if you set a font-instance object as the value of the `font` property, then the font name used to create that object will be used instead). If some cases, however, doing this conversion does not make sense, and this will be noted in the documentation for particular types of instance objects.

If *property* is not a built-in property, then this function will simply set its value if *locale* is `nil`. However, if *locale* is given, then this function will attempt to add *value* as the instantiator for the given *locale*, using `add-spec-to-specifier`. If the value of the property is not a specifier, it will automatically be converted into a `generic` specifier.

face-property *face property* &optional *locale* Function

This function returns *face*'s value of the given *property*.

If *locale* is omitted, the *face*'s actual value for *property* will be returned. For built-in properties, this will be a specifier object of a type appropriate to the property (e.g. a font or color specifier). For other properties, this could be anything.

If *locale* is supplied, then instead of returning the actual value, the specification(s) for the given locale or locale type will be returned. This will only work if the actual value of *property* is a specifier (this will always be the case for built-in properties, but not or not may apply to user-defined properties). If the actual value of *property* is not a specifier, this value will simply be returned regardless of *locale*.

The return value will be a list of instantiators (e.g. strings specifying a font or color name), or a list of specifications, each of which is a cons of a locale and a list of instantiators. Specifically, if *locale* is a particular locale (a buffer, window, frame, device, or `global`), a list of instantiators for that locale will be returned. Otherwise, if *locale* is a locale type (one of the symbols `buffer`, `window`, `frame`, or `device`), the specifications for all locales of that type will be returned. Finally, if *locale* is `all`, the specifications for all locales of all types will be returned.

The specifications in a specifier determine what the value of *property* will be in a particular *domain* or set of circumstances, which is typically a particular Emacs window along with the buffer it contains and the frame and device it lies within. The value is derived from the instantiator associated with the most specific locale (in the order buffer, window, frame, device, and `global`) that matches the domain in question. In other words, given a domain (i.e. an Emacs window, usually), the specifier for *property* will first be searched for a specification whose locale is the buffer contained within that window; then for a specification whose locale is the window itself; then for a specification whose locale is the frame that the window is contained within; etc. The first instantiator that is valid for the domain (usually this means that the instantiator is recognized by the device [i.e. the X server or TTY device] that the domain is on). The function `face-property-instance` actually does all this, and is used to determine how to display the face.

face-property-instance *face property* &optional *domain default no-fallback* Function

This function returns the instance of *face*'s *property* in the specified *domain*.

Under most circumstances, *domain* will be a particular window, and the returned instance describes how the specified property actually is displayed for that window and the particular buffer in it. Note that this may not be the same as how the property appears when the buffer is displayed in a different window or frame, or how the property appears in the same window if you switch to another buffer in that window; and in those cases, the returned instance would be different.

The returned instance will typically be a color-instance, font-instance, or pixmap-instance object, and you can query it using the appropriate object-specific functions. For example, you could use `color-instance-rgb-components` to find out the RGB (red, green, and blue) components of how the `background` property of the `highlight` face is displayed in a particular window. The results might be different from the results you would get for another window (perhaps the user specified a different color for the frame that window is on; or perhaps the same color was specified but the window is on a different X server, and that X server has different RGB values for the color from this one).

domain defaults to the selected window if omitted.

domain can be a frame or device, instead of a window. The value returned for a such a domain is used in special circumstances when a more specific domain does not apply; for example, a frame value might be used for coloring a toolbar, which is conceptually attached to a frame rather than a particular window. The value is also useful in determining what the value would be for a particular window within the frame or device, if it is not overridden by a more specific specification.

If *property* does not name a built-in property, its value will simply be returned unless it is a specifier object, in which case it will be instanced using `specifier-instance`.

Optional arguments *default* and *no-fallback* are the same as in `specifier-instance`. See Chapter 41 [Specifiers], page 599.

42.1.4 Face Convenience Functions

set-face-foreground *face color* &optional *locale tag how-to-add* Function
set-face-background *face color* &optional *locale tag how-to-add* Function

These functions set the foreground (respectively, background) color of face *face* to *color*. The argument *color* should be a string (the name of a color) or a color object as returned by `make-color` (see Section 42.3 [Colors], page 623).

set-face-background-pixmap *face pixmap* &optional *locale tag how-to-add* Function

This function sets the background pixmap of face *face* to *pixmap*. The argument *pixmap* should be a string (the name of a bitmap or pixmap file; the directories listed in the variable `x-bitmap-file-path` will be searched) or a glyph object as returned by `make-glyph` (see Chapter 43 [Glyphs], page 625). The argument may also be a list of the form `(`*width height data*`)` where *width* and *height* are the size in pixels, and *data* is a string, containing the raw bits of the bitmap.

set-face-font *face font* &optional *locale tag how-to-add* Function

This function sets the font of face *face*. The argument *font* should be a string or a font object as returned by `make-font` (see Section 42.2 [Fonts], page 621).

set-face-underline-p *face underline-p* &optional *locale tag how-to-add* Function

This function sets the underline property of face *face*.

face-foreground *face* &optional *locale* Function
face-background *face* &optional *locale* Function

These functions return the foreground (respectively, background) color specifier of face *face*. See Section 42.3 [Colors], page 623.

face-background-pixmap *face* &optional *locale* Function

This function return the background-pixmap glyph object of face *face*.

face-font *face* &optional *locale* Function

This function returns the font specifier of face *face*. (Note: This is not the same as the function `face-font` in FSF Emacs.) See Section 42.2 [Fonts], page 621.

face-font-name *face* &optional *domain* Function

This function returns the name of the font of face *face*, or `nil` if it is unspecified. This is basically equivalent to `(font-name (face-font` *face*`)` *domain*`)` except that it does not cause an error if *face*'s font is `nil`. (This function is named `face-font` in FSF Emacs.)

face-underline-p *face* &optional *locale* Function

This function returns the underline property of face *face*.

face-foreground-instance *face* &optional *domain* Function
face-background-instance *face* &optional *domain* Function

These functions return the foreground (respectively, background) color specifier of face *face*. See Section 42.3 [Colors], page 623.

face-background-pixmap-instance *face* &optional *domain* Function

This function return the background-pixmap glyph object of face *face*.

face-font-instance *face* &optional *domain* Function

This function returns the font specifier of face *face*. See Section 42.2 [Fonts], page 621.

42.1.5 Other Face Display Functions

invert-face *face* &optional *locale* 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.

face-equal *face1 face2* &optional *domain* Function

This returns `t` if the faces *face1* and *face2* will display in the same way. *domain* is as in `face-property-instance`.

face-differs-from-default-p *face* &optional *domain* Function

This returns `t` if the face *face* displays differently from the default face. *domain* is as in `face-property-instance`.

42.2 Fonts

This section describes how to work with font specifier and font instance objects, which encapsulate fonts in the window system.

42.2.1 Font Specifiers

font-specifier-p *object* Function

This predicate returns `t` if *object* is a font specifier, and `nil` otherwise.

42.2.2 Font Instances

font-instance-p *object* Function

This predicate returns `t` if *object* is a font instance, and `nil` otherwise.

make-font-instance *name* &optional *device noerror* Function

This function creates a new font-instance object of the specified name. *device* specifies the device this object applies to and defaults to the selected device. An error is signalled if the font is unknown or cannot be allocated; however, if *noerror* is non-`nil`, `nil` is simply returned in this case.

The returned object is a normal, first-class lisp object. The way you "deallocate" the font is the way you deallocate any other lisp object: you drop all pointers to it and allow it to be garbage collected. When these objects are GCed, the underlying X data is deallocated as well.

42.2.3 Font Instance Names

list-fonts *pattern* &optional *device* Function

This function returns a list of font names matching the given pattern. *device* specifies which device to search for names, and defaults to the currently selected device.

font-instance-name *font-instance* Function

This function returns the name used to allocate *font-instance*.

font-instance-truename *font-instance* Function

This function returns the canonical name of the given font instance. Font names are patterns which may match any number of fonts, of which the first found is used. This returns an unambiguous name for that font (but not necessarily its only unambiguous name).

42.2.4 Font Instance Size

x-font-size *font* Function

This function returns the nominal size of the given font. This is done by parsing its name, so it's likely to lose. X fonts can be specified (by the user) in either pixels or 10ths of points, and this returns the first one it finds, so you have to decide which units the returned value is measured in yourself ...

x-find-larger-font *font* &optional *device* Function

This function loads a new, slightly larger version of the given font (or font name). Returns the font if it succeeds, `nil` otherwise. If scalable fonts are available, this returns a font which is 1 point larger. Otherwise, it returns the next larger version of this font that is defined.

x-find-smaller-font *font* &optional *device* Function

This function loads a new, slightly smaller version of the given font (or font name). Returns the font if it succeeds, `nil` otherwise. If scalable fonts are available, this returns a font which is 1 point smaller. Otherwise, it returns the next smaller version of this font that is defined.

42.2.5 Font Instance Characteristics

font-instance-properties *font* Function

This function returns the properties (an alist or `nil`) of *font-instance*.

x-make-font-bold *font* &optional *device* Function

Given an X font specification, this attempts to make a "bold" font. If it fails, it returns `nil`.

x-make-font-unbold *font* &optional *device* Function

Given an X font specification, this attempts to make a non-bold font. If it fails, it returns `nil`.

x-make-font-italic *font* &optional *device* Function

Given an X font specification, this attempts to make an "italic" font. If it fails, it returns `nil`.

x-make-font-unitalic *font* &optional *device* Function

Given an X font specification, this attempts to make a non-italic font. If it fails, it returns `nil`.

x-make-font-bold-italic *font* &optional *device* Function

Given an X font specification, this attempts to make a "bold-italic" font. If it fails, it returns `nil`.

42.2.6 Font Convenience Functions

font-name *font* &optional *domain* Function

This function returns the name of the *font* in the specified *domain*, if any. *font* should be a font specifier object and *domain* is normally a window and defaults to the selected window if omitted. This is equivalent to using `specifier-instance` and applying `font-instance-name` to the result.

font-truename *font* &optional *domain* Function

This function returns the truename of the *font* in the specified *domain*, if any. *font* should be a font specifier object and *domain* is normally a window and defaults to the selected window if omitted. This is equivalent to using `specifier-instance` and applying `font-instance-truename` to the result.

font-properties *font* &optional *domain* Function

This function returns the properties of the *font* in the specified *domain*, if any. *font* should be a font specifier object and *domain* is normally a window and defaults to the selected window if omitted. This is equivalent to using `specifier-instance` and applying `font-instance-properties` to the result.

42.3 Colors

42.3.1 Color Specifiers

color-specifier-p *object* Function

This function returns non-`nil` if *object* is a color specifier.

42.3.2 Color Instances

A *color-instance object* is an object describing the way a color specifier is instanced in a particular domain. Functions such as `face-background-instance` return a color-instance object. For example,

```
(face-background-instance 'default (next-window))
     ⇒ #<color-instance moccasin 47=(FFFF,E4E4,B5B5) 0x678d>
```

The color-instance object returned describes the way the background color of the `default` face is displayed in the next window after the selected one.

color-instance-p *object* Function

This function returns non-`nil` if *object* is a color-instance.

42.3.3 Color Instance Properties

color-instance-name *color-instance* Function

This function returns the name used to allocate *color-instance*.

color-instance-rgb-components *color-instance* Function

This function returns a three element list containing the red, green, and blue color components of *color-instance*.

```
(color-instance-rgb-components
  (face-background-instance 'default (next-window)))
    ⇒ (65535 58596 46517)
```

42.3.4 Color Convenience Functions

color-name *color* &optional *domain* Function

This function returns the name of the *color* in the specified *domain*, if any. *color* should be a color specifier object and *domain* is normally a window and defaults to the selected window if omitted. This is equivalent to using `specifier-instance` and applying `color-instance-name` to the result.

color-rgb-components *color* &optional *domain* Function

This function returns the RGB components of the *color* in the specified *domain*, if any. *color* should be a color specifier object and *domain* is normally a window and defaults to the selected window if omitted. This is equivalent to using `specifier-instance` and applying `color-instance-rgb-components` to the result.

```
(color-rgb-components (face-background 'default (next-window)))
    ⇒ (65535 58596 46517)
```

43 Glyphs

A *glyph* is an object that is used for pixmaps and images of all sorts, as well as for things that "act" like pixmaps, such as non-textual strings (*annotations*) displayed in a buffer or in the margins. It is used in begin-glyphs and end-glyphs attached to extents, marginal and textual annotations, overlay arrows (`overlay-arrow-*` variables), toolbar buttons, mouse pointers, frame icons, truncation and continuation markers, and the like. (Basically, any place there is an image or something that acts like an image, there will be a glyph object representing it.)

The actual image that is displayed (as opposed to its position or clipping) is defined by an *image specifier* object contained within the glyph. The separation between an image specifier object and a glyph object is made because the glyph includes other properties than just the actual image: e.g. the face it is displayed in (for text images), the alignment of the image (when it is in a buffer), etc.

glyphp *object* Function

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

43.1 Glyph Functions

43.1.1 Creating Glyphs

make-glyph &optional *spec-list type* Function

This function creates a new glyph object of type *type*.

spec-list is used to initialize the glyph's image. It is typically an image instantiator (a string or a vector; Section 43.2.1 [Image Specifiers], page 630), but can also be a list of such instantiators (each one in turn is tried until an image is successfully produced), a cons of a locale (frame, buffer, etc.) and an instantiator, a list of such conses, or any other form accepted by `canonicalize-spec-list`. See Chapter 41 [Specifiers], page 599, for more information about specifiers.

type specifies the type of the glyph, which specifies in which contexts the glyph can be used, and controls the allowable image types into which the glyph's image can be instantiated. *type* should be one of `buffer` (used for glyphs in an extent, the modeline, the toolbar, or elsewhere in a buffer), `pointer` (used for the mouse-pointer), or `icon` (used for a frame's icon), and defaults to `buffer`. See Section 43.3 [Glyph Types], page 638.

make-glyph-internal &optional *type* Function

This function creates a new, uninitialized glyph of type *type*.

make-pointer-glyph &optional *spec-list* Function

This function is equivalent to calling `make-glyph` with a *type* of `pointer`.

make-icon-glyph &optional *spec-list* Function

This function is equivalent to calling `make-glyph` with a *type* of `icon`.

43.1.2 Glyph Properties

Each glyph has a list of properties, which control all of the aspects of the glyph's appearance. The following symbols have predefined meanings:

`image` The image used to display the glyph.

`baseline` Percent above baseline that glyph is to be displayed. Only for glyphs displayed inside of a buffer.

`contrib-p`
Whether the glyph contributes to the height of the line it's on. Only for glyphs displayed inside of a buffer.

`face` Face of this glyph (*not* a specifier).

set-glyph-property *glyph property value* &optional *locale tag-set how-to-add* Function

This function changes a property of a *glyph*.

For built-in properties, the actual value of the property is a specifier and you cannot change this; but you can change the specifications within the specifier, and that is what this function will do. For user-defined properties, you can use this function to either change the actual value of the property or, if this value is a specifier, change the specifications within it.

If *property* is a built-in property, the specifications to be added to this property can be supplied in many different ways:

- If *value* is a simple instantiator (e.g. a string naming a pixmap filename) or a list of instantiators, then the instantiator(s) will be added as a specification of the property for the given *locale* (which defaults to `global` if omitted).
- If *value* is a list of specifications (each of which is a cons of a locale and a list of instantiators), then *locale* must be `nil` (it does not make sense to explicitly specify a locale in this case), and specifications will be added as given.
- If *value* is a specifier (as would be returned by `glyph-property` if no *locale* argument is given), then some or all of the specifications in the specifier will be added to the property. In this case, the function is really equivalent to `copy-specifier` and *locale* has the same semantics (if it is a particular locale, the specification for the locale will be copied; if a locale type, specifications for all locales of that type will be copied; if `nil` or `all`, then all specifications will be copied).

how-to-add should be either `nil` or one of the symbols `prepend`, `append`, `remove-tag-set-prepend`, `remove-tag-set-append`, `remove-locale`, `remove-locale-type`, or `remove-all`. See `copy-specifier` and `add-spec-to-specifier` for a description of what each of these means. Most of the time, you do not need to worry about this argument; the default behavior usually is fine.

In general, it is OK to pass an instance object (e.g. as returned by `glyph-property-instance`) as an instantiator in place of an actual instantiator. In such a case, the instantiator used to create that instance object will be used (for example, if you set a font-instance object as the value of the `font` property, then the font name used to create that object will be used instead). If some cases, however, doing this conversion does not make sense, and this will be noted in the documentation for particular types of instance objects.

If *property* is not a built-in property, then this function will simply set its value if *locale* is `nil`. However, if *locale* is given, then this function will attempt to add *value* as the instantiator for the given *locale*, using `add-spec-to-specifier`. If the value of the property is not a specifier, it will automatically be converted into a `generic` specifier.

glyph-property *glyph property* &optional *locale* Function

This function returns *glyph*'s value of the given *property*.

If *locale* is omitted, the *glyph*'s actual value for *property* will be returned. For built-in properties, this will be a specifier object of a type appropriate to the property (e.g. a font or color specifier). For other properties, this could be anything.

If *locale* is supplied, then instead of returning the actual value, the specification(s) for the given locale or locale type will be returned. This will only work if the actual value of *property* is a specifier (this will always be the case for built-in properties, but may or may not apply to user-defined properties). If the actual value of *property* is not a specifier, this value will simply be returned regardless of *locale*.

The return value will be a list of instantiators (e.g. vectors specifying pixmap data), or a list of specifications, each of which is a cons of a locale and a list of instantiators. Specifically, if *locale* is a particular locale (a buffer, window, frame, device, or `global`), a list of instantiators for that locale will be returned. Otherwise, if *locale* is a locale type (one of the symbols `buffer`, `window`, `frame`, or `device`), the specifications for all locales of that type will be returned. Finally, if *locale* is `all`, the specifications for all locales of all types will be returned.

The specifications in a specifier determine what the value of *property* will be in a particular *domain* or set of circumstances, which is typically a particular Emacs window along with the buffer it contains and the frame and device it lies within. The value is derived from the instantiator associated with the most specific locale (in the order buffer, window, frame, device, and `global`) that matches the domain in question. In other words, given a domain (i.e. an Emacs window, usually), the specifier for *property* will first be searched for a specification whose locale is the buffer contained within that window; then for a specification whose locale is the window itself; then for a specification whose locale is the frame that the window is contained within; etc. The first instantiator that is valid for the domain (usually this means that the instantiator is recognized by the device [i.e. the X server or TTY device] that the domain is on). The function `glyph-property-instance` actually does all this, and is used to determine how to display the glyph.

glyph-property-instance *glyph property* &optional *domain default no-fallback* Function

This function returns the instance of *glyph*'s *property* in the specified *domain*.

Under most circumstances, *domain* will be a particular window, and the returned instance describes how the specified property actually is displayed for that window and the particular buffer in it. Note that this may not be the same as how the property appears when the buffer is displayed in a different window or frame, or how the property appears in the same window if you switch to another buffer in that window; and in those cases, the returned instance would be different.

The returned instance is an image-instance object, and you can query it using the appropriate image instance functions. For example, you could use `image-instance-depth` to find out the depth (number of color planes) of a pixmap displayed in a particular window. The results might be different from the results you would get for another window (perhaps the user specified a different image for the frame that window is on; or perhaps the same image was specified but the window is on a different X server, and that X server has different color capabilities from this one).

domain defaults to the selected window if omitted.

domain can be a frame or device, instead of a window. The value returned for such a domain is used in special circumstances when a more specific domain does not apply; for example, a frame value might be used for coloring a toolbar, which is conceptually attached to a frame rather than a particular window. The value is also useful in determining what the value would be for a particular window within the frame or device, if it is not overridden by a more specific specification.

If *property* does not name a built-in property, its value will simply be returned unless it is a specifier object, in which case it will be instanced using `specifier-instance`.

Optional arguments *default* and *no-fallback* are the same as in `specifier-instance`. See Chapter 41 [Specifiers], page 599.

remove-glyph-property *glyph property* &optional *locale tag-set exact-p* Function

This function removes a property from a glyph. For built-in properties, this is analogous to `remove-specifier`. See Chapter 41 [Specifiers], page 599, for the meaning of the *locale*, *tag-set*, and *exact-p* arguments.

43.1.3 Glyph Convenience Functions

The following functions are provided for working with specific properties of a glyph. Note that these are exactly like calling the general functions described above and passing in the appropriate value for *property*.

Remember that if you want to determine the "value" of a specific glyph property, you probably want to use the `*-instance` functions. For example, to determine whether a glyph contributes to its line height, use `glyph-contrib-p-instance`, not `glyph-contrib-p`. (The latter will return a boolean specifier or a list of specifications, and you probably aren't concerned with these.)

glyph-image *glyph* &optional *locale* — Function

This function is equivalent to calling `glyph-property` with a property of `image`. The return value will be an image specifier if *locale* is `nil` or omitted; otherwise, it will be a specification or list of specifications.

set-glyph-image *glyph spec* &optional *locale tag-set how-to-add* — Function

This function is equivalent to calling `set-glyph-property` with a property of `image`.

glyph-image-instance *glyph* &optional *domain default no-fallback* — Function

This function returns the instance of *glyph*'s image in the given *domain*, and is equivalent to calling `glyph-property-instance` with a property of `image`. The return value will be an image instance.

Normally *domain* will be a window or `nil` (meaning the selected window), and an instance object describing how the image appears in that particular window and buffer will be returned.

glyph-contrib-p *glyph* &optional *locale* — Function

This function is equivalent to calling `glyph-property` with a property of `contrib-p`. The return value will be a boolean specifier if *locale* is `nil` or omitted; otherwise, it will be a specification or list of specifications.

set-glyph-contrib-p *glyph spec* &optional *locale tag-set how-to-add* — Function

This function is equivalent to calling `set-glyph-property` with a property of `contrib-p`.

glyph-contrib-p-instance *glyph* &optional *domain default no-fallback* — Function

This function returns whether the glyph contributes to its line height in the given *domain*, and is equivalent to calling `glyph-property-instance` with a property of `contrib-p`. The return value will be either `nil` or `t`. (Normally *domain* will be a window or `nil`, meaning the selected window.)

glyph-baseline *glyph* &optional *locale* — Function

This function is equivalent to calling `glyph-property` with a property of `baseline`. The return value will be a specifier if *locale* is `nil` or omitted; otherwise, it will be a specification or list of specifications.

set-glyph-baseline *glyph spec* &optional *locale tag-set how-to-add* — Function

This function is equivalent to calling `set-glyph-property` with a property of `baseline`.

glyph-baseline-instance *glyph* &optional *domain default no-fallback* Function

This function returns the instance of *glyph*'s baseline value in the given *domain*, and is equivalent to calling `glyph-property-instance` with a property of `baseline`. The return value will be an integer or `nil`.

Normally *domain* will be a window or `nil` (meaning the selected window), and an instance object describing the baseline value appears in that particular window and buffer will be returned.

glyph-face *glyph* Function

This function returns the face of *glyph*. (Remember, this is not a specifier, but a simple property.)

set-glyph-face *glyph face* Function

This function changes the face of *glyph* to *face*.

43.1.4 Glyph Dimensions

glyph-width *glyph* &optional *window* Function

This function returns the width of *glyph* on *window*. This may not be exact as it does not take into account all of the context that redisplay will.

glyph-ascent *glyph* &optional *window* Function

This function returns the ascent value of *glyph* on *window*. This may not be exact as it does not take into account all of the context that redisplay will.

glyph-descent *glyph* &optional *window* Function

This function returns the descent value of *glyph* on *window*. This may not be exact as it does not take into account all of the context that redisplay will.

glyph-height *glyph* &optional *window* Function

This function returns the height of *glyph* on *window*. (This is equivalent to the sum of the ascent and descent values.) This may not be exact as it does not take into account all of the context that redisplay will.

43.2 Images

43.2.1 Image Specifiers

An image specifier is used to describe the actual image of a glyph. It works like other specifiers (see Chapter 41 [Specifiers], page 599), in that it contains a number of specifications describing how the image should appear in a variety of circumstances. These specifications are called *image instantiators*. When XEmacs wants to display the image, it instantiates the image into an *image instance*. Image instances are their own primitive object type (similar to font instances and color instances), describing how the image appears in a particular domain. (On the other hand, image instantiators, which are just descriptions of how the image should appear, are represented using strings or vectors.)

image-specifier-p *object* Function

This function returns non-`nil` if *object* is an image specifier. Usually, an image specifier results from calling `glyph-image` on a glyph.

make-image-specifier *spec-list* Function

This function creates a new image specifier object and initializes it according to *spec-list*. It is unlikely that you will ever want to do this, but this function is provided for completeness and for experimentation purposes. See Chapter 41 [Specifiers], page 599.

Image instantiators come in many formats: `xbm`, `xpm`, `gif`, `jpeg`, etc. This describes the format of the data describing the image. The resulting image instances also come in many types – `mono-pixmap`, `color-pixmap`, `text`, `pointer`, etc. This refers to the behavior of the image and the sorts of places it can appear. (For example, a color-pixmap image has fixed colors specified for it, while a mono-pixmap image comes in two unspecified shades "foreground" and "background" that are determined from the face of the glyph or surrounding text; a text image appears as a string of text and has an unspecified foreground, background, and font; a pointer image behaves like a mono-pixmap image but can only be used as a mouse pointer [mono-pixmap images cannot be used as mouse pointers]; etc.) It is important to keep the distinction between image instantiator format and image instance type in mind. Typically, a given image instantiator format can result in many different image instance types (for example, `xpm` can be instanced as `color-pixmap`, `mono-pixmap`, or `pointer`; whereas `cursor-font` can be instanced only as `pointer`), and a particular image instance type can be generated by many different image instantiator formats (e.g. `color-pixmap` can be generated by `xpm`, `gif`, `jpeg`, etc.).

See Section 43.2.3 [Image Instances], page 635, for a more detailed discussion of image instance types.

An image instantiator should be a string or a vector of the form

`[`*format* *:keyword* *value* `...]`

i.e. a format symbol followed by zero or more alternating keyword-value pairs. The *format* field should be a symbol, one of

`nothing`
: (Don't display anything; no keywords are valid for this. Can only be instanced as `nothing`.)

`string`
: (Display this image as a text string. Can only be instanced as `text`, although support for instancing as `mono-pixmap` should be added.)

`formatted-string`
: (Display this image as a text string with replaceable fields, similar to a modeline format string; not currently implemented.)

`xbm`
: (An X bitmap; only if X support was compiled into this XEmacs. Can be instanced as `mono-pixmap`, `color-pixmap`, or `pointer`.)

`xpm`
: (An XPM pixmap; only if XPM support was compiled into this XEmacs. Can be instanced as `color-pixmap`, `mono-pixmap`, or `pointer`. XPM is an add-on library for X that was designed to rectify the shortcomings of the XBM format. Most implementations of X include the XPM library as a standard part. If your

vendor does not, it is highly recommended that you download it and install it. You can get it from the standard XEmacs FTP site, among other places.)

`xface` (An X-Face bitmap, used to encode people's faces in e-mail messages; only if X-Face support was compiled into this XEmacs. Can be instanced as `mono-pixmap`, `color-pixmap`, or `pointer`.)

`gif` (A GIF87 or GIF89 image; only if GIF support was compiled into this XEmacs. Can be instanced as `color-pixmap`. Note that XEmacs includes GIF decoding functions as a standard part of it, so if you have X support, you will normally have GIF support, unless you explicitly disable it at configure time.)

`jpeg` (A JPEG-format image; only if JPEG support was compiled into this XEmacs. Can be instanced as `color-pixmap`. If you have the JPEG libraries present on your system when XEmacs is built, XEmacs will automatically detect this and use them, unless you explicitly disable it at configure time.)

`png` (A PNG/GIF24 image; only if PNG support was compiled into this XEmacs. Can be instanced as `color-pixmap`.)

`tiff` (A TIFF-format image; only if TIFF support was compiled into this XEmacs. Not currently implemented.)

`cursor-font`
(One of the standard cursor-font names, such as '`watch`' or '`right_ptr`' under X. Under X, this is, more specifically, any of the standard cursor names from appendix B of the Xlib manual [also known as the file '`<X11/cursorfont.h>`'] minus the '`XC_`' prefix. On other window systems, the valid names will be specific to the type of window system. Can only be instanced as `pointer`.)

`font` (A glyph from a font; i.e. the name of a font, and glyph index into it of the form '*font* `fontname index [[mask-font] mask-index]`'. Only if X support was compiled into this XEmacs. Currently can only be instanced as `pointer`, although this should probably be fixed.)

`subwindow`
(An embedded X window; not currently implemented.)

`autodetect`
(XEmacs tries to guess what format the data is in. If X support exists, the data string will be checked to see if it names a filename. If so, and this filename contains XBM or XPM data, the appropriate sort of pixmap or pointer will be created. [This includes picking up any specified hotspot or associated mask file.] Otherwise, if `pointer` is one of the allowable image-instance types and the string names a valid cursor-font name, the image will be created as a pointer. Otherwise, the image will be displayed as text. If no X support exists, the image will always be displayed as text.)

The valid keywords are:

`:data` (Inline data. For most formats above, this should be a string. For XBM images, this should be a list of three elements: width, height, and a string of bit data. This keyword is not valid for instantiator format `nothing`.)

:file (Data is contained in a file. The value is the name of this file. If both :data and :file are specified, the image is created from what is specified in :data and the string in :file becomes the value of the image-instance-file-name function when applied to the resulting image-instance. This keyword is not valid for instantiator formats nothing, string, formatted-string, cursor-font, font, and autodetect.)

:foreground
:background
(For xbm, xface, cursor-font, and font. These keywords allow you to explicitly specify foreground and background colors. The argument should be anything acceptable to make-color-instance. This will cause what would be a mono-pixmap to instead be colorized as a two-color color-pixmap, and specifies the foreground and/or background colors for a pointer instead of black and white.)

:mask-data
(For xbm and xface. This specifies a mask to be used with the bitmap. The format is a list of width, height, and bits, like for :data.)

:mask-file
(For xbm and xface. This specifies a file containing the mask data. If neither a mask file nor inline mask data is given for an XBM image, and the XBM image comes from a file, XEmacs will look for a mask file with the same name as the image file but with 'Mask' or 'msk' appended. For example, if you specify the XBM file 'left_ptr' [usually located in '/usr/include/X11/bitmaps'], the associated mask file 'left_ptrmsk' will automatically be picked up.)

:hotspot-x
:hotspot-y
(For xbm and xface. These keywords specify a hotspot if the image is instantiated as a pointer. Note that if the XBM image file specifies a hotspot, it will automatically be picked up if no explicit hotspot is given.)

:color-symbols
(Only for xpm. This specifies an alist that maps strings that specify symbolic color names to the actual color to be used for that symbolic color (in the form of a string or a color-specifier object). If this is not specified, the contents of xpm-color-symbols are used to generate the alist.)

If instead of a vector, the instantiator is a string, it will be converted into a vector by looking it up according to the specs in the console-type-image-conversion-list for the console type of the domain (usually a window; sometimes a frame or device) over which the image is being instantiated.

If the instantiator specifies data from a file, the data will be read in at the time that the instantiator is added to the image specifier (which may be well before the image is actually displayed), and the instantiator will be converted into one of the inline-data forms, with the filename retained using a :file keyword. This implies that the file must exist when the instantiator is added to the image, but does not need to exist at any other time (e.g. it may safely be a temporary file).

valid-image-instantiator-format-p *format* Function

This function returns non-`nil` if *format* is a valid image instantiator format. Note that the return value for many formats listed above depends on whether XEmacs was compiled with support for that format.

image-instantiator-format-list Function

This function return a list of valid image-instantiator formats.

xpm-color-symbols Variable

This variable holds definitions of logical color-names used when reading XPM files. Elements of this list should be of the form `(`*color-name form-to-evaluate*`)`. The *color-name* should be a string, which is the name of the color to define; the *form-to-evaluate* should evaluate to a color specifier object, or a string to be passed to `make-color-instance` (see Section 42.3 [Colors], page 623). If a loaded XPM file references a symbolic color called *color-name*, it will display as the computed color instead.

The default value of this variable defines the logical color names '`"foreground"`' and '`"background"`' to be the colors of the `default` face.

x-bitmap-file-path Variable

A list of the directories in which X bitmap files may be found. If nil, this is initialized from the '`"*bitmapFilePath"`' resource. This is used by the `make-image-instance` function (however, note that if the environment variable '`XBMLANGPATH`' is set, it is consulted first).

43.2.2 Image Instantiator Conversion

set-console-type-image-conversion-list *console-type list* Function

This function sets the image-conversion-list for consoles of the given *console-type*. The image-conversion-list specifies how image instantiators that are strings should be interpreted. Each element of the list should be a list of two elements (a regular expression string and a vector) or a list of three elements (the preceding two plus an integer index into the vector). The string is converted to the vector associated with the first matching regular expression. If a vector index is specified, the string itself is substituted into that position in the vector.

Note: The conversion above is applied when the image instantiator is added to an image specifier, not when the specifier is actually instantiated. Therefore, changing the image-conversion-list only affects newly-added instantiators. Existing instantiators in glyphs and image specifiers will not be affected.

console-type-image-conversion-list *console-type* Function

This function returns the image-conversion-list for consoles of the given *console-type*.

43.2.3 Image Instances

Image-instance objects encapsulate the way a particular image (pixmap, etc.) is displayed on a particular device.

In most circumstances, you do not need to directly create image instances; use a glyph instead. However, it may occasionally be useful to explicitly create image instances, if you want more control over the instantiation process.

image-instance-p *object* Function
  This function returns non-`nil` if *object* is an image instance.

43.2.3.1 Image Instance Types

Image instances come in a number of different types. The type of an image instance specifies the nature of the image: Whether it is a text string, a mono pixmap, a color pixmap, etc.

The valid image instance types are

`nothing` Nothing is displayed.

`text` Displayed as text. The foreground and background colors and the font of the text are specified independent of the pixmap. Typically these attributes will come from the face of the surrounding text, unless a face is specified for the glyph in which the image appears.

`mono-pixmap`
  Displayed as a mono pixmap (a pixmap with only two colors where the foreground and background can be specified independent of the pixmap; typically the pixmap assumes the foreground and background colors of the text around it, unless a face is specified for the glyph in which the image appears).

`color-pixmap`
  Displayed as a color pixmap.

`pointer` Used as the mouse pointer for a window.

`subwindow`
  A child window that is treated as an image. This allows (e.g.) another program to be responsible for drawing into the window. Not currently implemented.

valid-image-instance-type-p *type* Function
  This function returns non-`nil` if *type* is a valid image instance type.

image-instance-type-list Function
  This function returns a list of the valid image instance types.

image-instance-type *image-instance* Function
  This function returns the type of the given image instance. The return value will be one of `nothing`, `text`, `mono-pixmap`, `color-pixmap`, `pointer`, or `subwindow`.

text-image-instance-p *object* Function
  This function returns non-`nil` if *object* is an image instance of type `text`.

mono-pixmap-image-instance-p *object* Function

This function returns non-`nil` if *object* is an image instance of type `mono-pixmap`.

color-pixmap-image-instance-p *object* Function

This function returns non-`nil` if *object* is an image instance of type `color-pixmap`.

pointer-image-instance-p *object* Function

This function returns non-`nil` if *object* is an image instance of type `pointer`.

subwindow-image-instance-p *object* Function

This function returns non-`nil` if *object* is an image instance of type `subwindow`.

nothing-image-instance-p *object* Function

This function returns non-`nil` if *object* is an image instance of type `nothing`.

43.2.3.2 Image Instance Functions

make-image-instance *data* &optional *device dest-types no-error* Function

This function creates a new image-instance object.

data is an image instantiator, which describes the image (see Section 43.2.1 [Image Specifiers], page 630).

dest-types should be a list of allowed image instance types that can be generated. The *dest-types* list is unordered. If multiple destination types are possible for a given instantiator, the "most natural" type for the instantiator's format is chosen. (For XBM, the most natural types are `mono-pixmap`, followed by `color-pixmap`, followed by `pointer`. For the other normal image formats, the most natural types are `color-pixmap`, followed by `mono-pixmap`, followed by `pointer`. For the string and formatted-string formats, the most natural types are `text`, followed by `mono-pixmap` (not currently implemented), followed by `color-pixmap` (not currently implemented). The other formats can only be instantiated as one type. (If you want to control more specifically the order of the types into which an image is instantiated, just call `make-image-instance` repeatedly until it succeeds, passing less and less preferred destination types each time.

If *dest-types* is omitted, all possible types are allowed.

no-error controls what happens when the image cannot be generated. If *nil*, an error message is generated. If *t*, no messages are generated and this function returns *nil*. If anything else, a warning message is generated and this function returns *nil*.

colorize-image-instance *image-instance foreground background* Function

This function makes the image instance be displayed in the given colors. Image instances come in two varieties: bitmaps, which are 1 bit deep which are rendered in the prevailing foreground and background colors; and pixmaps, which are of arbitrary depth (including 1) and which have the colors explicitly specified. This function converts a bitmap to a pixmap. If the image instance was a pixmap already, nothing is done (and `nil` is returned). Otherwise `t` is returned.

image-instance-name *image-instance* Function
This function returns the name of the given image instance.

image-instance-string *image-instance* Function
This function returns the string of the given image instance. This will only be non-`nil` for text image instances.

image-instance-file-name *image-instance* Function
This function returns the file name from which *image-instance* was read, if known.

image-instance-mask-file-name *image-instance* Function
This function returns the file name from which *image-instance*'s mask was read, if known.

image-instance-depth *image-instance* Function
This function returns the depth of the image instance. This is 0 for a mono pixmap, or a positive integer for a color pixmap.

image-instance-height *image-instance* Function
This function returns the height of the image instance, in pixels.

image-instance-width *image-instance* Function
This function returns the width of the image instance, in pixels.

image-instance-hotspot-x *image-instance* Function
This function returns the X coordinate of the image instance's hotspot, if known. This is a point relative to the origin of the pixmap. When an image is used as a mouse pointer, the hotspot is the point on the image that sits over the location that the pointer points to. This is, for example, the tip of the arrow or the center of the crosshairs.

This will always be `nil` for a non-pointer image instance.

image-instance-hotspot-y *image-instance* Function
This function returns the Y coordinate of the image instance's hotspot, if known.

image-instance-foreground *image-instance* Function
This function returns the foreground color of *image-instance*, if applicable. This will be a color instance or `nil`. (It will only be non-`nil` for colorized mono pixmaps and for pointers.)

image-instance-background *image-instance* Function
This function returns the background color of *image-instance*, if applicable. This will be a color instance or `nil`. (It will only be non-`nil` for colorized mono pixmaps and for pointers.)

43.3 Glyph Types

Each glyph has a particular type, which controls how the glyph's image is generated. Each glyph type has a corresponding list of allowable image instance types that can be generated. When you call `glyph-image-instance` to retrieve the image instance of a glyph, XEmacs does the equivalent of calling `make-image-instance` and passing in *dest-types* the list of allowable image instance types for the glyph's type.

- `buffer` glyphs can be used as the begin-glyph or end-glyph of an extent, in the modeline, and in the toolbar. Their image can be instantiated as `nothing`, `mono-pixmap`, `color-pixmap`, `text`, and `subwindow`.
- `pointer` glyphs can be used to specify the mouse pointer. Their image can be instantiated as `pointer`.
- `icon` glyphs can be used to specify the icon used when a frame is iconified. Their image can be instantiated as `mono-pixmap` and `color-pixmap`.

glyph-type *glyph* — Function

This function returns the type of the given glyph. The return value will be a symbol, one of `buffer`, `pointer`, or `icon`.

valid-glyph-type-p *glyph-type* — Function

Given a *glyph-type*, this function returns non-`nil` if it is valid.

glyph-type-list — Function

This function returns a list of valid glyph types.

buffer-glyph-p *object* — Function

This function returns non-`nil` if *object* is a glyph of type `buffer`.

icon-glyph-p *object* — Function

This function returns non-`nil` if *object* is a glyph of type `icon`.

pointer-glyph-p *object* — Function

This function returns non-`nil` if *object* is a glyph of type `pointer`.

43.4 Mouse Pointer

The shape of the mouse pointer when over a particular section of a frame is controlled using various glyph variables. Since the image of a glyph is a specifier, it can be controlled on a per-buffer, per-frame, per-window, or per-device basis.

You should use `set-glyph-image` to set the following variables, *not* `setq`.

text-pointer-glyph — Glyph

This variable specifies the shape of the mouse pointer when over text.

nontext-pointer-glyph — Glyph

This variable specifies the shape of the mouse pointer when over a buffer, but not over text. If unspecified in a particular domain, `text-pointer-glyph` is used.

modeline-pointer-glyph Glyph

This variable specifies the shape of the mouse pointer when over the modeline. If unspecified in a particular domain, `nontext-pointer-glyph` is used.

selection-pointer-glyph Glyph

This variable specifies the shape of the mouse pointer when over a selectable text region. If unspecified in a particular domain, `text-pointer-glyph` is used.

gc-pointer-glyph Glyph

This variable specifies the shape of the mouse pointer when a garbage collection is in progress. If the selected window is on a window system and this glyph specifies a value (i.e. a pointer image instance) in the domain of the selected window, the pointer will be changed as specified during garbage collection. Otherwise, a message will be printed in the echo area, as controlled by `gc-message`.

busy-pointer-glyph Glyph

This variable specifies the shape of the mouse pointer when XEmacs is busy. If unspecified in a particular domain, the pointer is not changed when XEmacs is busy.

menubar-pointer-glyph Glyph

This variable specifies the shape of the mouse pointer when over the menubar. If unspecified in a particular domain, the window-system-provided default pointer is used.

scrollbar-pointer-glyph Glyph

This variable specifies the shape of the mouse pointer when over a scrollbar. If unspecified in a particular domain, the window-system-provided default pointer is used.

toolbar-pointer-glyph Glyph

This variable specifies the shape of the mouse pointer when over a toolbar. If unspecified in a particular domain, `nontext-pointer-glyph` is used.

Internally, these variables are implemented in `default-mouse-motion-handler`, and thus only take effect when the mouse moves. That function calls `set-frame-pointer`, which sets the current mouse pointer for a frame.

set-frame-pointer *frame image-instance* Function

This function sets the mouse pointer of *frame* to the given pointer image instance. You should not call this function directly. (If you do, the pointer will change again the next time the mouse moves.)

43.5 Redisplay Glyphs

truncation-glyph Glyph

This variable specifies what is displayed at the end of truncated lines.

continuation-glyph Glyph

This variable specifies what is displayed at the end of wrapped lines.

octal-escape-glyph Glyph

This variable specifies what to prefix character codes displayed in octal with.

hscroll-glyph Glyph

This variable specifies what to display at the beginning of horizontally scrolled lines.

invisible-text-glyph Glyph

This variable specifies what to use to indicate the presence of invisible text. This is the glyph that is displayed when an ellipsis is called for, according to `selective-display-ellipses` or `buffer-invisibility-spec`). Normally this is three dots ("...").

control-arrow-glyph Glyph

This variable specifies what to use as an arrow for control characters.

43.6 Subwindows

Subwindows are not currently implemented.

subwindowp *object* Function

This function returns non-`nil` if *object* is a subwindow.

44 Annotations

An *annotation* is a pixmap or string that is not part of a buffer's text but is displayed next to a particular location in a buffer. Annotations can be displayed intermixed with text, in any whitespace at the beginning or end of a line, or in a special area at the left or right side of the frame called a *margin*, whose size is controllable. Annotations are implemented using extents (see Chapter 40 [Extents], page 585); but you can work with annotations without knowing how extents work.

44.1 Annotation Basics

Marginal annotations are notes associated with a particular location in a buffer. They may be displayed in a margin created on the left-hand or right-hand side of the frame, in any whitespace at the beginning or end of a line, or inside of the text itself. Every annotation may have an associated action to be performed when the annotation is selected. The term *annotation* is used to refer to an individual note. The term *margin* is generically used to refer to the whitespace before the first character on a line or after the last character on a line.

Each annotation has the following characteristics:

glyph This is a glyph object and is used as the displayed representation of the annotation.

down-glyph
If given, this glyph is used as the displayed representation of the annotation when the mouse is pressed down over the annotation.

face The face with which to display the glyph.

side Which side of the text (left or right) the annotation is displayed at.

action If non-`nil`, this field must contain a function capable of being the first argument to `funcall`. This function is normally evaluated with a single argument, the value of the *data* field, each time the annotation is selected. However, if the *with-event* parameter to `make-annotation` is non-`nil`, the function is called with two arguments. The first argument is the same as before, and the second argument is the event (a button-up event, usually) that activated the annotation.

data Not used internally. This field can contain any E-Lisp object. It is passed as the first argument to *action* described above.

menu A menu displayed when the right mouse button is pressed over the annotation.

The margin is divided into *outside* and *inside*. The outside margin is space on the left or right side of the frame which normal text cannot be displayed in. The inside margin is that space between the leftmost or rightmost point at which text can be displayed and where the first or last character actually is.

There are four different *layout types* which affect the exact location an annotation appears.

`outside-margin`
: The annotation is placed in the outside margin area. as close as possible to the edge of the frame. If the outside margin is not wide enough for an annotation to fit, it is not displayed.

`inside-margin`
: The annotation is placed in the inside margin area, as close as possible to the edge of the frame. If the inside margin is not wide enough for the annotation to fit, it will be displayed using any available outside margin space if and only if the specifier `use-left-overflow` or `use-right-overflow` (depending on which side the annotation appears in) is non-`nil`.

`whitespace`
: The annotation is placed in the inside margin area, as close as possible to the first or last non-whitespace character on a line. If the inside margin is not wide enough for the annotation to fit, it will be displayed if and only if the specifier `use-left-overflow` or `use-right-overflow` (depending on which side the annotation appears in) is non-`nil`.

`text`
: The annotation is placed at the position it is inserted. It will create enough space for itself inside of the text area. It does not take up a place in the logical buffer, only in the display of the buffer.

The current layout policy is that all `whitespace` annotations are displayed first. Next, all `inside-margin` annotations are displayed using any remaining space. Finally as many `outside-margin` annotations are displayed as possible. The `text` annotations will always display as they create their own space to display in.

44.2 Annotation Primitives

make-annotation *glyph* &optional *position layout buffer with-event d-glyph rightp* — Function

This function creates a marginal annotation at position *pos* in *buffer*. The annotation is displayed using *glyph*, which should be a glyph object or a string, and is positioned using layout policy *layout*. If *pos* is `nil`, point is used. If *layout* is `nil`, `whitespace` is used. If *buffer* is `nil`, the current buffer is used.

If *with-event* is non-`nil`, then when an annotation is activated, the triggering event is passed as the second arg to the annotation function. If *d-glyph* is non-`nil` then it is used as the glyph that will be displayed when button1 is down. If *rightp* is non-`nil` then the glyph will be displayed on the right side of the buffer instead of the left.

The newly created annotation is returned.

delete-annotation *annotation* — Function

This function removes *annotation* from its buffer. This does not modify the buffer text.

annotationp *annotation* — Function

This function returns `t` if *annotation* is an annotation, `nil` otherwise.

44.3 Annotation Properties

annotation-glyph *annotation* Function
This function returns the glyph object used to display *annotation*.

set-annotation-glyph *annotation glyph* &optional *layout side* Function
This function sets the glyph of *annotation* to *glyph*, which should be a glyph object. If *layout* is non-`nil`, set the layout policy of *annotation* to *layout*. If *side* is `left` or `right`, change the side of the buffer at which the annotation is displayed to the given side. The new value of `annotation-glyph` is returned.

annotation-down-glyph *annotation* Function
This function returns the glyph used to display *annotation* when the left mouse button is depressed on the annotation.

set-annotation-down-glyph *annotation glyph* Function
This function returns the glyph used to display *annotation* when the left mouse button is depressed on the annotation to *glyph*, which should be a glyph object.

annotation-face *annotation* Function
This function returns the face associated with *annotation*.

set-annotation-face *annotation face* Function
This function sets the face associated with *annotation* to *face*.

annotation-layout *annotation* Function
This function returns the layout policy of *annotation*.

set-annotation-layout *annotation layout* Function
This function sets the layout policy of *annotation* to *layout*.

annotation-side *annotation* Function
This function returns the side of the buffer that *annotation* is displayed on. Return value is a symbol, either `left` or `right`.

annotation-data *annotation* Function
This function returns the data associated with *annotation*.

set-annotation-data *annotation data* Function
This function sets the data field of *annotation* to *data*. *data* is returned.

annotation-action *annotation* Function
This function returns the action associated with *annotation*.

set-annotation-action *annotation action* Function
This function sets the action field of *annotation* to *action*. *action* is returned..

annotation-menu *annotation* Function
This function returns the menu associated with *annotation*.

set-annotation-menu *annotation menu* Function

This function sets the menu associated with *annotation* to *menu*. This menu will be displayed when the right mouse button is pressed over the annotation.

annotation-visible *annotation* Function

This function returns `t` if there is enough available space to display *annotation*, `nil` otherwise.

annotation-width *annotation* Function

This function returns the width of *annotation* in pixels.

hide-annotation *annotation* Function

This function removes *annotation*'s glyph, making it invisible.

reveal-annotation *annotation* Function

This function restores *annotation*'s glyph, making it visible.

44.4 Locating Annotations

annotations-in-region *start end buffer* Function

This function returns a list of all annotations in *buffer* which are between *start* and *end* inclusively.

annotations-at &optional *position buffer* Function

This function returns a list of all annotations at *position* in *buffer*. If *position* is `nil` point is used. If *buffer* is `nil` the current buffer is used.

annotation-list &optional *buffer* Function

This function returns a list of all annotations in *buffer*. If *buffer* is `nil`, the current buffer is used.

all-annotations Function

This function returns a list of all annotations in all buffers in existence.

44.5 Margin Primitives

The margin widths are controllable on a buffer-local, window-local, frame-local, device-local, or device-type-local basis through the use of specifiers. See Chapter 41 [Specifiers], page 599.

left-margin-width Specifier

This is a specifier variable controlling the width of the left outside margin, in characters. Use `set-specifier` to change its value.

right-margin-width Specifier

This is a specifier variable controlling the width of the right outside margin, in characters. Use `set-specifier` to change its value.

use-left-overflow Specifier

If non-`nil`, use the left outside margin as extra whitespace when displaying `whitespace` and `inside-margin` annotations. Defaults to `nil`. This is a specifier variable; use `set-specifier` to change its value.

use-right-overflow Specifier

If non-`nil`, use the right outside margin as extra whitespace when displaying `whitespace` and `inside-margin` annotations. Defaults to `nil`. This is a specifier variable; use `set-specifier` to change its value.

window-left-margin-pixel-width &optional *window* Function

This function returns the width in pixels of the left outside margin of *window*. If *window* is `nil`, the selected window is assumed.

window-right-margin-pixel-width &optional *window* Function

This function returns the width in pixels of the right outside margin of *window*. If *window* is `nil`, the selected window is assumed.

The margin colors are controlled by the faces `left-margin` and `right-margin`. These can be set using the X resources `Emacs.left-margin.background` and `Emacs.left-margin.foreground`; likewise for the right margin.

44.6 Annotation Hooks

The following three hooks are provided for use with the marginal annotations:

`before-delete-annotation-hook`
: This hook is called immediately before an annotation is destroyed. It is passed a single argument, the annotation being destroyed.

`after-delete-annotation-hook`
: This normal hook is called immediately after an annotation is destroyed.

`make-annotation-hook`
: This hook is called immediately after an annotation is created. It is passed a single argument, the newly created annotation.

45 Emacs Display

This chapter describes a number of other features related to the display that XEmacs presents to the user.

45.1 Refreshing the Screen

The function `redraw-frame` redisplays the entire contents of a given frame. See Chapter 32 [Frames], page 471.

redraw-frame *frame* Function

This function clears and redisplays frame *frame*.

Even more powerful is `redraw-display`:

redraw-display &optional *device* Command

This function redraws all frames on *device* marked as having their image garbled. *device* defaults to the selected device. If *device* is `t`, all devices will have their frames checked.

Processing user input takes absolute priority over redisplay. If you call these functions when input is available, they do nothing immediately, but a full redisplay does happen eventually—after all the input has been processed.

Normally, suspending and resuming XEmacs also refreshes the screen. Some terminal emulators record separate contents for display-oriented programs such as XEmacs and for ordinary sequential display. If you are using such a terminal, you might want to inhibit the redisplay on resumption. See Section 50.2.2 [Suspending XEmacs], page 696.

no-redraw-on-reenter Variable

This variable controls whether XEmacs redraws the entire screen after it has been suspended and resumed. Non-`nil` means yes, `nil` means no.

The above functions do not actually cause the display to be updated; rather, they clear out the internal display records that XEmacs maintains, so that the next time the display is updated it will be redrawn from scratch. Normally this occurs the next time that `next-event` or `sit-for` is called; however, a display update will not occur if there is input pending. See Chapter 19 [Command Loop], page 281.

force-cursor-redisplay Function

This function causes an immediate update of the cursor on the selected frame. (This function does not exist in FSF Emacs.)

45.2 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 normally shown with a '\' in the rightmost column of the window on X displays, and with a '`$`' on TTY devices. When a line is continued or "wrapped" onto the next line, this is shown with a curved arrow in the rightmost column of the window (or with a '\' on TTY devices). The additional screen lines used to display a long text line are called *continuation* lines.

Normally, whenever line truncation is in effect for a particular window, a horizontal scrollbar is displayed in that window if the device supports scrollbars. See Chapter 24 [Scrollbars], page 357.

Note that continuation is different from filling; continuation happens on the screen only, not in the buffer contents, and it breaks a line precisely at the right margin, not at a word boundary. See Section 36.11 [Filling], page 526.

truncate-lines User Option

This buffer-local variable controls how XEmacs displays lines that extend beyond the right edge of the window. If it is non-`nil`, then XEmacs does not display continuation lines; rather each line of text occupies exactly one screen line, and a backslash 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 always used for side-by-side windows (within one frame) regardless of the value of `truncate-lines`.

default-truncate-lines User Option

This variable is the default value for `truncate-lines`, for buffers that do not have local values for it.

truncate-partial-width-windows User Option

This variable controls display of lines that extend beyond the right edge of the window, in side-by-side windows (see Section 31.2 [Splitting Windows], page 446). If it is non-`nil`, these lines are truncated; otherwise, `truncate-lines` says what to do with them.

The backslash and curved arrow used to indicate truncated or continued lines are only defaults, and can be changed. These images are actually glyphs (see Chapter 43 [Glyphs], page 625). XEmacs provides a great deal of flexibility in how glyphs can be controlled. (This differs from FSF Emacs, which uses display tables to control these images.)

For details, Section 43.5 [Redisplay Glyphs], page 639.

45.3 The Echo Area

The *echo area* is used for displaying messages made with the `message` primitive, and for echoing keystrokes. It is not the same as the minibuffer, despite the fact that the minibuffer appears (when active) in the same place on the screen as the echo area. The *XEmacs Reference 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 XEmacs Reference Manual*). Error messages appear in the echo area; see Section 9.5.3 [Errors], page 136.

You can write output in the echo area by using the Lisp printing functions with `t` as the stream (see Section 17.5 [Output Functions], page 256), or as follows:

message *string* &rest *arguments* Function

This function displays a one-line message in the echo area. The argument *string* is similar to a C language `printf` control string. See `format` in Section 4.7 [String Conversion], page 65, for the details on the conversion specifications. `message` returns the constructed string.

In batch mode, `message` prints the message text on the standard error stream, followed by a newline.

If *string* is `nil`, `message` clears the echo area. If the minibuffer is active, this brings the minibuffer contents back onto the screen immediately.

```
(message "Minibuffer depth is %d."
         (minibuffer-depth))
  ⊣ Minibuffer depth is 0.
⇒ "Minibuffer depth is 0."

---------- Echo Area ----------
Minibuffer depth is 0.
---------- Echo Area ----------
```

In addition to only displaying a message, XEmacs allows you to *label* your messages, giving you fine-grained control of their display. Message label is a symbol denoting the message type. Some standard labels are:

- `message`—default label used by the `message` function;
- `error`—default label used for reporting errors;
- `progress`—progress indicators like ‘`Converting... 45%`’ (not logged by default);
- `prompt`—prompt-like messages like ‘`Isearch: foo`’ (not logged by default);
- `command`—helper command messages like ‘`Mark set`’ (not logged by default);
- `no-log`—messages that should never be logged

Several messages may be stacked in the echo area at once. Lisp programs may access these messages, or remove them as appropriate, via the message stack.

display-message *label message* &optional *frame stdout-p* Function

This function displays *message* (a string) labeled as *label*, as described above.

The *frame* argument specifies the frame to whose minibuffer the message should be printed. This is currently unimplemented. The *stdout-p* argument is used internally.

```
(display-message 'command "Mark set")
```

lmessage *label string* &rest *arguments* Function

This function displays a message *string* with label *label*. It is similar to `message` in that it accepts a `printf`-like strings and any number of arguments.

```
;; Display a command message.
(lmessage 'command "Comment column set to %d" comment-column)

;; Display a progress message.
(lmessage 'progress "Fontifying %s... (%d)" buffer percentage)

;; Display a message that should not be logged.
(lmessage 'no-log "Done")
```

clear-message &optional *label frame stdout-p no-restore* — Function

This function remove any message with the given *label* from the message-stack, erasing it from the echo area if it's currently displayed there.

If a message remains at the head of the message-stack and *no-restore* is `nil`, it will be displayed. The string which remains in the echo area will be returned, or `nil` if the message-stack is now empty. If *label* is nil, the entire message-stack is cleared.

```
;; Show a message, wait for 2 seconds, and restore old minibuffer
;; contents.
(message "A message")
 ⊣ A message
⇒ "A Message"
(lmessage 'my-label "Newsflash!  Newsflash!")
 ⊣ Newsflash!  Newsflash!
⇒ "Newsflash!  Newsflash!"
(sit-for 2)
(clear-message 'my-label)
 ⊣ A message
⇒ "A message"
```

Unless you need the return value or you need to specify a label, you should just use `(message nil)`.

current-message &optional *frame* — Function

This function returns the current message in the echo area, or `nil`. The *frame* argument is currently unused.

Some of the messages displayed in the echo area are also recorded in the '`*Message-Log*`' buffer. Exactly which messages will be recorded can be tuned using the following variables.

log-message-max-size — User Option

This variable specifies the maximum size of the '`*Message-log*`' buffer.

log-message-ignore-labels — Variable

This variable specifies the labels whose messages will not be logged. It should be a list of symbols.

log-message-ignore-regexps — Variable

This variable specifies the regular expressions matching messages that will not be logged. It should be a list of regular expressions.

Normally, packages that generate messages that might need to be ignored should label them with `progress`, `prompt`, or `no-log`, so they can be filtered by `log-message-ignore-labels`.

echo-keystrokes Variable

This variable determines how much time should elapse before command characters echo. Its value must be a number, which specifies the number of seconds to wait before echoing. If the user types a prefix key (such as `C-x`) and then delays this many seconds before continuing, the prefix key is echoed in the echo area. Any subsequent characters in the same command will be echoed as well.

If the value is zero, then command input is not echoed.

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.

45.4 Warnings

XEmacs contains a facility for unified display of various warnings. Unlike errors, warnings are displayed in the situations when XEmacs encounters a problem that is recoverable, but which should be fixed for safe future operation.

For example, warnings are printed by the startup code when it encounters problems with X keysyms, when there is an error in '`.emacs`', and in other problematic situations. Unlike messages, warnings are displayed in a separate buffer, and include an explanatory message that may span across several lines. Here is an example of how a warning is displayed:

```
(1) (initialization/error) An error has occurred while loading ~/.emacs:

Symbol's value as variable is void: bogus-variable

To ensure normal operation, you should investigate the cause of the error
in your initialization file and remove it.  Use the '-debug-init' option
to XEmacs to view a complete error backtrace.
```

Each warning has a *class* and a *priority level*. The class is a symbol describing what sort of warning this is, such as `initialization`, `resource` or `key-mapping`.

The warning priority level specifies how important the warning is. The recognized warning levels, in increased order of priority, are: `debug`, `info`, `notice`, `warning`, `error`, `critical`, `alert` and `emergency`.

display-warning *class message* &optional *level* Function

This function displays a warning message *message* (a string). *class* should be a warning class symbol, as described above, or a list of such symbols. *level* describes the warning priority level. If unspecified, it default to `warning`.

```
(display-warning 'resource
  "Bad resource specification encountered:
something like

    Emacs*foo: bar

You should replace the * with a . in order to get proper behavior when
you use the specifier and/or `set-face-*' functions.")

---------- Warning buffer ----------
(1) (resource/warning) Bad resource specification encountered:
something like

    Emacs*foo: bar

You should replace the * with a . in order to get proper behavior when
you use the specifier and/or `set-face-*' functions.
---------- Warning buffer ----------
```

lwarn *class level message* &rest *args* — Function

This function displays a formatted labeled warning message. As above, *class* should be the warning class symbol, or a list of such symbols, and *level* should specify the warning priority level (`warning` by default).

Unlike in `display-warning`, *message* may be a formatted message, which will be, together with the rest of the arguments, passed to `format`.

```
(lwarn 'message-log 'warning
  "Error caught in `remove-message-hook': %s"
  (error-message-string e))
```

log-warning-minimum-level — Variable

This variable specifies the minimum level of warnings that should be generated. Warnings with level lower than defined by this variable are completely ignored, as if they never happened.

display-warning-minimum-level — Variable

This variable specifies the minimum level of warnings that should be displayed. Unlike `log-warning-minimum-level`, setting this function does not suppress warnings entirely—they are still generated in the '`*Warnings*`' buffer, only they are not displayed by default.

log-warning-suppressed-classes — Variable

This variable specifies a list of classes that should not be logged or displayed. If any of the class symbols associated with a warning is the same as any of the symbols listed here, the warning will be completely ignored, as it they never happened.

display-warning-suppressed-classes — Variable

This variable specifies a list of classes that should not be logged or displayed. If any of the class symbols associated with a warning is the same as any of the symbols listed here, the warning will not be displayed. The warning will still logged in the *Warnings* buffer (unless also contained in 'log-warning-suppressed-classes'), but the buffer will not be automatically popped up.

45.5 Invisible Text

You can make characters *invisible*, so that they do not appear on the screen, with the `invisible` property. This can be either a text property or a property of an overlay.

In the simplest case, any non-`nil` `invisible` property makes a character invisible. This is the default case—if you don't alter the default value of `buffer-invisibility-spec`, this is how the `invisibility` property works. This feature is much like selective display (see Section 45.6 [Selective Display], page 654), but more general and cleaner.

More generally, you can use the variable `buffer-invisibility-spec` to control which values of the `invisible` property make text invisible. This permits you to classify the text into different subsets in advance, by giving them different `invisible` values, and subsequently make various subsets visible or invisible by changing the value of `buffer-invisibility-spec`.

Controlling visibility with `buffer-invisibility-spec` is especially useful in a program to display the list of entries in a data base. It permits the implementation of convenient filtering commands to view just a part of the entries in the data base. Setting this variable is very fast, much faster than scanning all the text in the buffer looking for properties to change.

buffer-invisibility-spec Variable

This variable specifies which kinds of `invisible` properties actually make a character invisible.

`t` A character is invisible if its `invisible` property is non-`nil`. This is the default.

a list Each element of the list makes certain characters invisible. Ultimately, a character is invisible if any of the elements of this list applies to it. The list can have two kinds of elements:

atom A character is invisible if its `invisible` property value is *atom* or if it is a list with *atom* as a member.

(*atom* . `t`)
A character is invisible if its `invisible` property value is *atom* or if it is a list with *atom* as a member. Moreover, if this character is at the end of a line and is followed by a visible newline, it displays an ellipsis.

Ordinarily, commands that operate on text or move point do not care whether the text is invisible. However, the user-level line motion commands explicitly ignore invisible newlines.

45.6 Selective Display

Selective display is a pair of features that hide certain lines on the screen.

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 has traditionally used this variant. It has been partially replaced by the invisible text feature (see Section 45.5 [Invisible Text], page 653); there is a new version of Outline mode which uses that instead.

In the second variant, the choice of lines to hide is made automatically based on indentation. This variant is designed to be a user-level feature.

The way you control explicit selective display is by replacing a newline (control-j) with a carriage return (control-m). The text that was formerly a line following that newline is now invisible. Strictly speaking, it is temporarily no longer a line at all, since only newlines can separate lines; it is now part of the previous line.

Selective display does not directly affect editing commands. For example, `C-f` (`forward-char`) moves point unhesitatingly into invisible text. However, the replacement of newline characters with carriage return characters affects some editing commands. For example, `next-line` skips invisible lines, since it searches only for newlines. Modes that use selective display can also define commands that take account of the newlines, or that make parts of the text visible or invisible.

When you write a selectively displayed buffer into a file, all the control-m's are output as newlines. This means that when you next read in the file, it looks OK, with nothing invisible. The selective display effect is seen only within XEmacs.

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 that many columns of indentation are not displayed.

When some portion of a buffer is invisible, the vertical movement commands operate as if that portion did not exist, allowing a single `next-line` command to skip any number of invisible lines. However, character movement commands (such as `forward-char`) do not skip the invisible portion, and it is possible (if tricky) to insert or delete text in an invisible portion.

In the examples below, we show the *display appearance* of the buffer `foo`, which changes with the value of `selective-display`. The *contents* of the buffer do not change.

```
(setq selective-display nil)
     ⇒ nil

---------- Buffer: foo ----------
1 on this column
 2on this column
  3n this column
  3n this column
 2on this column
1 on this column
---------- Buffer: foo ----------

(setq selective-display 2)
     ⇒ 2

---------- Buffer: foo ----------
1 on this column
 2on this column
 2on this column
1 on this column
---------- Buffer: foo ----------
```

selective-display-ellipses Variable

If this buffer-local variable is non-`nil`, then XEmacs 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 45.11 [Display Tables], page 659.

45.7 The Overlay Arrow

The *overlay arrow* is useful for directing the user's attention to a particular line in a buffer. For example, in the modes used for interface to debuggers, the overlay arrow indicates the line of code about to be executed.

overlay-arrow-string Variable

This variable holds the string to display to call attention to a particular line, or `nil` if the arrow feature is not in use. Despite its name, the value of this variable can be either a string or a glyph (see Chapter 43 [Glyphs], page 625).

overlay-arrow-position Variable

This variable holds a marker that indicates where to display the overlay arrow. It should point at the beginning of a line. The arrow text appears at the beginning of that line, overlaying any text that would otherwise appear. Since the arrow is usually short, and the line usually begins with indentation, normally nothing significant is overwritten.

The overlay string is displayed only in the buffer that this marker points into. Thus, only one buffer can have an overlay arrow at any given time.

You can do the same job by creating an extent with a `begin-glyph` property. See Section 40.6 [Extent Properties], page 591.

45.8 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 they are "output" in the general sense of the word, are not affected). See Section 17.5 [Output Functions], page 256.

The value of the last form in *forms* is returned.

```
---------- Buffer: foo ----------
 This is the contents of foo.
---------- Buffer: foo ----------

(with-output-to-temp-buffer "foo"
    (print 20)
    (print standard-output))
⇒ #<buffer foo>

---------- Buffer: foo ----------
20

#<buffer foo>

---------- Buffer: foo ----------
```

temp-buffer-show-function Variable

If this variable is non-`nil`, `with-output-to-temp-buffer` calls it as a function to do the job of displaying a help buffer. The function gets one argument, which is the buffer it should display.

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*. It has no effect on the undo list or on the buffer's modification status.

The momentary display remains until the next input event. If the next input event is *char*, `momentary-string-display` ignores it and returns. Otherwise, that event remains buffered for subsequent use as input. Thus, typing *char* will simply remove the string from the display, while typing (say) *C-f* will remove the string from the display and later (presumably) move point forward. The argument *char* is a space by default.

The return value of `momentary-string-display` is not meaningful.

You can do the same job in a more general way by creating an extent with a begin-glyph property. See Section 40.6 [Extent Properties], page 591.

If *message* is non-`nil`, it is displayed in the echo area while *string* is displayed in the buffer. If it is `nil`, a default message says to type *char* to continue.

In this example, point is initially located at the beginning of the second line:

```
---------- Buffer: foo ----------
This is the contents of foo.
⋆Second line.
---------- Buffer: foo ----------

(momentary-string-display
  "**** Important Message! ****"
  (point) ?\r
  "Type RET when done reading")
⇒ t

---------- Buffer: foo ----------
This is the contents of foo.
**** Important Message! ****Second line.
---------- Buffer: foo ----------

---------- Echo Area ----------
Type RET when done reading
---------- Echo Area ----------
```

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.

45.9 Blinking Parentheses

This section describes the mechanism by which XEmacs shows a matching open parenthesis when the user inserts a close parenthesis.

blink-paren-function Variable

The value of this variable should be a function (of no arguments) to be called whenever a character with close parenthesis syntax is inserted. The value of `blink-paren-function` may be `nil`, in which case nothing is done.

> **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-paren-delay Variable

This variable specifies the number of seconds for the cursor to remain at the matching parenthesis. A fraction of a second often gives good results, but the default is 1, which works on all systems.

blink-matching-open 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, it displays the character's context in the echo area. To avoid long delays, this function does not search farther than `blink-matching-paren-distance` characters.

Here is an example of calling this function explicitly.

```
(defun interactive-blink-matching-open ()
  "Indicate momentarily the start of sexp before point."
  (interactive)
  (let ((blink-matching-paren-distance
         (buffer-size))
        (blink-matching-paren t))
    (blink-matching-open)))
```

45.10 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 45.11 [Display Tables], page 659). Here are the usual display conventions:

- Character codes 32 through 126 map to glyph codes 32 through 126. Normally this means they display as themselves.
- Character code 9 is a horizontal tab. It displays as whitespace up to a position determined by `tab-width`.
- Character code 10 is a newline.
- All other codes in the range 0 through 31, and code 127, display in one of two ways according to the value of `ctl-arrow`. If it is non-`nil`, these codes map to sequences of two glyphs, where the first glyph is the ASCII code for '^'. (A display table can specify a glyph to use instead of '^'.) Otherwise, these codes map just like the codes in the range 128 to 255.
- Character codes 128 through 255 map to sequences of four glyphs, where the first glyph is the ASCII code for '\', and the others are digit characters representing the code in octal. (A display table can specify a glyph to use instead of '\'.)

The usual display conventions apply even when there is a display table, for any character whose entry in the active display table is `nil`. Thus, when you set up a display table, you need only specify the characters for which you want 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. See Section 10.9.3 [Default Value], page 159.

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 36.16.5 [Indent Tabs], page 537.

45.11 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 *runes*, each rune being an image that takes up one character position on the screen. You can also define how to display each rune on your terminal, using the *rune table*.

45.11.1 Display Table Format

A display table is an array of 256 elements. (In FSF Emacs, a display table is 262 elements. The six extra elements specify the truncation and continuation glyphs, etc. This method is very kludgey, and in XEmacs the variables `truncation-glyph`, `continuation-glyph`, etc. are used. See Section 45.2 [Truncation], page 648.)

make-display-table Function

This creates and returns a display table. The table initially has `nil` in all elements.

The 256 elements correspond to character codes; the *n*th element says how to display the character code *n*. The value should be `nil`, a string, a glyph, or a vector of strings and glyphs (see Section 45.11.3 [Character Descriptors], page 660). If an element is `nil`, it says to display that character according to the usual display conventions (see Section 45.10 [Usual Display], page 658).

If you use the display table to change the display of newline characters, the whole buffer will be displayed as one long "line."

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 (concat "^" (char-to-string (+ i 64)))))
    (setq i (1+ i)))
  (aset disptab 127 "^?"))
```

45.11.2 Active Display Table

The active display table is controlled by the variable `current-display-table`. This is a specifier, which means that you can specify separate values for it in individual buffers, windows, frames, and devices, as well as a global value. It also means that you cannot set this variable using `setq`; use `set-specifier` instead. See Chapter 41 [Specifiers], page 599. (FSF Emacs uses `window-display-table`, `buffer-display-table`, `standard-display-table`, etc. to control the display table. However, specifiers are a cleaner and more powerful way of doing the same thing. FSF Emacs also uses a different format for the contents of a display table, using additional indirection to a "glyph table" and such. Note that "glyph" has a different meaning in XEmacs.)

Individual faces can also specify an overriding display table; this is set using `set-face-display-table`. See Section 42.1 [Faces], page 615.

If no display table can be determined for a particular window, then XEmacs uses the usual display conventions. See Section 45.10 [Usual Display], page 658.

45.11.3 Character Descriptors

Each element of the display-table vector describes how to display a particular character and is called a *character descriptor*. A character descriptor can be:

a string Display this particular string wherever the character is to be displayed.

a glyph Display this particular glyph wherever the character is to be displayed.

a vector The vector may contain strings and/or glyphs. Display the elements of the vector one after another wherever the character is to be displayed.

`nil` Display according to the standard interpretation (see Section 45.10 [Usual Display], page 658).

45.12 Beeping

You can make XEmacs ring a bell, play a sound, 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 136.)

ding &optional *dont-terminate sound device* 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`. If *sound* is specified, it should be a symbol specifying which sound to make. This sound will be played if `visible-bell` is `nil`. (This only works if sound support was compiled into the executable and you are running on the console of a Sun SparcStation, SGI, HP9000s700, or Linux PC. Otherwise you just get a beep.) The optional third argument specifies what device to make the sound on, and defaults to the selected device.

beep &optional *dont-terminate sound device* Function

This is a synonym for `ding`.

visible-bell User Option

This variable determines whether XEmacs should flash the screen to represent a bell. Non-`nil` means yes, `nil` means no. On TTY devices, this is effective only if the Termcap entry for the terminal type has the visible bell flag ('`vb`') set.

sound-alist Variable

This variable holds an alist associating names with sounds. When `beep` or `ding` is called with one of the name symbols, the associated sound will be generated instead of the standard beep.

Each element of `sound-alist` is a list describing a sound. The first element of the list is the name of the sound being defined. Subsequent elements of the list are alternating keyword/value pairs:

`sound` A string of raw sound data, or the name of another sound to play. The symbol `t` here means use the default X beep.

`volume` An integer from 0-100, defaulting to `bell-volume`.

`pitch` If using the default X beep, the pitch (Hz) to generate.

`duration` If using the default X beep, the duration (milliseconds).

For compatibility, elements of 'sound-alist' may also be:

- `( sound-name . <sound> )`
- `( sound-name <volume> <sound> )`

You should probably add things to this list by calling the function `load-sound-file`.

Caveats:

- You can only play audio data if running on the console screen of a Sun SparcStation, SGI, or HP9000s700.
- The pitch, duration, and volume options are available everywhere, but many X servers ignore the 'pitch' option.

The following beep-types are used by XEmacs itself:

`auto-save-error`
: when an auto-save does not succeed

`command-error`
: when the XEmacs command loop catches an error

`undefined-key`
: when you type a key that is undefined

`undefined-click`
: when you use an undefined mouse-click combination

`no-completion`
: during completing-read

`y-or-n-p` when you type something other than 'y' or 'n'

`yes-or-no-p`
: when you type something other than 'yes' or 'no'

`default` used when nothing else is appropriate.

Other lisp packages may use other beep types, but these are the ones that the C kernel of XEmacs uses.

bell-volume User Option

This variable specifies the default volume for sounds, from 0 to 100.

load-default-sounds Command

This function loads and installs some sound files as beep-types.

load-sound-file *filename sound-name* &optional *volume* Command

This function reads in an audio file and adds it to `sound-alist`. The sound file must be in the Sun/NeXT U-LAW format. *sound-name* should be a symbol, specifying the name of the sound. If *volume* is specified, the sound will be played at that volume; otherwise, the value of *bell-volume* will be used.

play-sound *sound* &optional *volume device* Function

This function plays sound *sound*, which should be a symbol mentioned in `sound-alist`. If *volume* is specified, it overrides the value (if any) specified in `sound-alist`. *device* specifies the device to play the sound on, and defaults to the selected device.

play-sound-file *file* &optional *volume device* Command

This function plays the named sound file at volume *volume*, which defaults to `bell-volume`. *device* specifies the device to play the sound on, and defaults to the selected device.

46 Hash Tables

hashtablep *object* Function

This function returns non-`nil` if *object* is a hash table.

46.1 Introduction to Hash Tables

A hash table is a data structure that provides mappings from arbitrary Lisp objects (called *keys*) to other arbitrary Lisp objects (called *values*). There are many ways other than hash tables of implementing the same sort of mapping, e.g. association lists (see Section 5.8 [Association Lists], page 92) and property lists (see Section 5.9 [Property Lists], page 96), but hash tables provide much faster lookup.

When you create a hash table, you specify a size, which indicates the expected number of elements that the table will hold. You are not bound by this size, however; hash tables automatically resize themselves if the number of elements becomes too large.

(Internally, hash tables are hashed using a modification of the *linear probing* hash table method. This method hashes each key to a particular spot in the hash table, and then scans forward sequentially until a blank entry is found. To look up a key, hash to the appropriate spot, then search forward for the key until either a key is found or a blank entry stops the search. The modification actually used is called *double hashing* and involves moving forward by a fixed increment, whose value is computed from the original hash value, rather than always moving forward by one. This eliminates problems with clustering that can arise from the simple linear probing method. For more information, see *Algorithms* (second edition) by Robert Sedgewick, pp. 236-241.)

make-hashtable *size* &optional *test-fun* Function

This function makes a hash table of initial size *size*. Comparison between keys is normally done with `eql`; i.e. two keys must be the same object to be considered equivalent. However, you can explicitly specify the comparison function using *test-fun*, which must be one of `eq`, `eql`, or `equal`.

Note that currently, `eq` and `eql` are the same. This will change when bignums are implemented.

copy-hashtable *old-table* Function

This function makes a new hash table which contains the same keys and values as the given table. The keys and values will not themselves be copied.

hashtable-fullness *table* Function

This function returns number of entries in *table*.

46.2 Working With Hash Tables

puthash *key val table* Function

This function hashes *key* to *val* in *table*.

gethash *key table* &optional *default* Function

This function finds the hash value for *key* in *table*. If there is no corresponding value, *default* is returned (defaults to `nil`).

remhash *key table* Function

This function removes the hash value for *key* in *table*.

clrhash *table* Function

This function flushes *table*. Afterwards, the hash table will contain no entries.

maphash *function table* Function

This function maps *function* over entries in *table*, calling it with two args, each key and value in the table.

46.3 Weak Hash Tables

A *weak hash table* is a special variety of hash table whose elements do not count as GC referents. For any key-value pair in such a hash table, if either the key or value (or in some cases, if one particular one of the two) has no references to it outside of weak hash tables (and similar structures such as weak lists), the pair will be removed from the table, and the key and value collected. A non-weak hash table (or any other pointer) would prevent the objects from being collected.

Weak hash tables are useful for keeping track of information in a non-obtrusive way, for example to implement caching. If the cache contains objects such as buffers, markers, image instances, etc. that will eventually disappear and get garbage-collected, using a weak hash table ensures that these objects are collected normally rather than remaining around forever, long past their actual period of use. (Otherwise, you'd have to explicitly map over the hash table every so often and remove unnecessary elements.)

There are three types of weak hash tables:

fully weak hash tables
: In these hash tables, a pair disappears if either the key or the value is unreferenced outside of the table.

key-weak hash tables
: In these hash tables, a pair disappears if the key is unreferenced outside of the table, regardless of how the value is referenced.

value-weak hash tables
: In these hash tables, a pair disappears if the value is unreferenced outside of the table, regardless of how the key is referenced.

Also see Section 5.10 [Weak Lists], page 98.

make-weak-hashtable *size* &optional *test-fun* Function

This function makes a fully weak hash table of initial size *size*. *test-fun* is as in `make-hashtable`.

make-key-weak-hashtable *size* &optional *test-fun* Function

This function makes a key-weak hash table of initial size *size*. *test-fun* is as in `make-hashtable`.

make-value-weak-hashtable *size* &optional *test-fun* Function

This function makes a value-weak hash table of initial size *size*. *test-fun* is as in `make-hashtable`.

47 Range Tables

A range table is a table that efficiently associated values with ranges of integers.

Note that range tables have a read syntax, like this:

```
#s(range-table data ((-3 2) foo (5 20) bar))
```

This maps integers in the range (-3, 2) to `foo` and integers in the range (5, 20) to `bar`.

range-table-p *object* Function

Return non-`nil` if *object* is a range table.

47.1 Introduction to Range Tables

make-range-table Function

Make a new, empty range table.

copy-range-table *old-table* Function

Make a new range table which contains the same values for the same ranges as the given table. The values will not themselves be copied.

47.2 Working With Range Tables

get-range-table *pos table* &optional *default* Function

This function finds value for position *pos* in *table*. If there is no corresponding value, return *default* (defaults to `nil`).

put-range-table *start end val table* Function

This function sets the value for range (*start*, *end*) to be *val* in *table*.

remove-range-table *start end table* Function

This function removes the value for range (*start*, *end*) in *table*.

clear-range-table *table* Function

This function flushes *table*.

map-range-table *function table* Function

This function maps *function* over entries in *table*, calling it with three args, the beginning and end of the range and the corresponding value.

48 Databases

databasep *object* Function

This function returns non-`nil` if *object* is a database.

48.1 Connecting to a Database

open-database *file* &optional *type subtype access mode* Function

This function opens database *file*, using database method *type* and *subtype*, with access rights *access* and permissions *mode*. *access* can be any combination of `r` `w` and `+`, for read, write, and creation flags.

type can have the value `'dbm` or `'berkeley_db` to select the type of database file to use. (Note: XEmacs may not support both of these types.)

For a *type* of `'dbm`, there are no subtypes, so *subtype* should by `nil`.

For a *type* of `'berkeley_db`, the following subtypes are available: `'hash`, `'btree`, and `'recno`. See the manpages for the Berkeley DB functions to more information about these types.

close-database *obj* Function

This function closes database *obj*.

database-live-p *obj* Function

This function returns `t` iff *obj* is an active database, else `nil`.

48.2 Working With a Database

get-database *key dbase* &optional *default* Function

This function finds the value for *key* in *database*. If there is no corresponding value, *default* is returned (`nil` if *default* is omitted).

map-database *function dbase* Function

This function maps *function* over entries in *database*, calling it with two args, each key and value in the database.

put-database *key val dbase* &optional *replace* Function

This function stores *key* and *val* in *database*. If optional fourth arg *replace* is non-`nil`, replace any existing entry in the database.

remove-database *key dbase* Function

This function removes *key* from *database*.

48.3 Other Database Functions

database-file-name *obj* Function

This function returns the filename associated with the database *obj*.

database-last-error &optional *obj* Function

This function returns the last error associated with database *obj*.

database-subtype *obj* Function

This function returns the subtype of database *obj*, if any.

database-type *obj* Function

This function returns the type of database *obj*.

49 Processes

In the terminology of operating systems, a *process* is a space in which a program can execute. XEmacs runs in a process. XEmacs Lisp programs can invoke other programs in processes of their own. These are called *subprocesses* or *child processes* of the XEmacs process, which is their *parent process*.

A subprocess of XEmacs 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 XEmacs 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.

49.1 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 49.4 [Asynchronous Processes], page 677). The other two, `call-process` and `call-process-region`, create a synchronous process and do not return a process object (see Section 49.2 [Synchronous Processes], page 674).

Synchronous and asynchronous processes are explained in following sections. Since the three functions are all called in a similar fashion, their common arguments are described here.

In all cases, the function's *program* argument specifies the program to be run. An error is signaled if the file is not found or cannot be executed. If the file name is relative, the variable `exec-path` contains a list of directories to search. Emacs initializes `exec-path` when it starts up, based on the value of the environment variable `PATH`. The standard file name constructs, '`~`', '`.`', and '`..`', are interpreted as usual in `exec-path`, but environment variable substitutions ('`$HOME`', etc.) are not recognized; use `substitute-in-file-name` to perform them (see Section 28.8.4 [File Name Expansion], page 409).

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 49.9.2 [Filter Functions], page 684, and Chapter 17 [Read and Print], page 251.) 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. You must use *args* to provide those.

The subprocess gets its current directory from the value of `default-directory` (see Section 28.8.4 [File Name Expansion], page 409).

The subprocess inherits its environment from XEmacs; but you can specify overrides for it with `process-environment`. See Section 50.3 [System Environment], page 697.

exec-directory Variable

The value of this variable is the name of a directory (a string) that contains programs that come with XEmacs, that are intended for XEmacs 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.

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.

49.2 Creating a Synchronous Process

After a *synchronous process* is created, XEmacs 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 XEmacs tries to do anything with it.

While Emacs waits for the synchronous subprocess to terminate, the user can quit by typing *C-g*. The first *C-g* tries to kill the subprocess with a `SIGINT` signal; but it waits until the subprocess actually terminates before quitting. If during that time the user types another *C-g*, that kills the subprocess instantly with `SIGKILL` and quits immediately. See Section 19.8 [Quitting], page 306.

The synchronous subprocess functions returned `nil` in version 18. In version 19, they return an indication of how the process terminated.

call-process *program* &optional *infile destination display* &rest *args* Function

This function calls *program* in a separate process and waits for it to finish.

The standard input for the process comes from file *infile* if *infile* is not `nil` and from '`/dev/null`' otherwise. The argument *destination* says where to put the process output. Here are the possibilities:

a buffer
: Insert the output in that buffer, before point. This includes both the standard output stream and the standard error stream of the process.

a string Find or create a buffer with that name, then insert the output in that buffer, before point.

`t` Insert the output in the current buffer, before point.

`nil` Discard the output.

0 Discard the output, and return immediately without waiting for the subprocess to finish.

In this case, the process is not truly synchronous, since it can run in parallel with Emacs; but you can think of it as synchronous in that Emacs is essentially finished with the subprocess as soon as this function returns.

(*real-destination error-destination*)
Keep the standard output stream separate from the standard error stream; deal with the ordinary output as specified by *real-destination*, and dispose of the error output according to *error-destination*. The value `nil` means discard it, `t` means mix it with the ordinary output, and a string specifies a file name to redirect error output into.

You can't directly specify a buffer to put the error output in; that is too difficult to implement. But you can achieve this result by sending the error output to a temporary file and then inserting the file into a buffer.

If *display* is non-`nil`, then `call-process` redisplays the buffer as output is inserted. Otherwise the function does no redisplay, and the results become visible on the screen only when XEmacs redisplays that buffer in the normal course of events.

The remaining arguments, *args*, are strings that specify command line arguments for the program.

The value returned by `call-process` (unless you told it not to wait) indicates the reason for process termination. A number gives the exit status of the subprocess; 0 means success, and any other value means failure. If the process terminated with a signal, `call-process` returns a string describing the signal.

In the examples below, the buffer '`foo`' is current.

```
(call-process "pwd" nil t)
     ⇒ nil

---------- Buffer: foo ----------
/usr/user/lewis/manual
---------- Buffer: foo ----------
```

```
(call-process "grep" nil "bar" nil "lewis" "/etc/passwd")
     ⇒ nil

---------- Buffer: bar ----------
lewis:5LTsHm66CSWKg:398:21:Bil Lewis:/user/lewis:/bin/csh

---------- Buffer: bar ----------
```

The `insert-directory` function contains a good example of the use of `call-process`:

```
(call-process insert-directory-program nil t nil switches
              (if full-directory-p
                  (concat (file-name-as-directory file) ".")
                file))
```

call-process-region *start end program* &optional *delete destination display* &rest *args* Function

This function sends the text between *start* to *end* as standard input to a process running *program*. It deletes the text sent if *delete* is non-`nil`; this is useful when *buffer* is `t`, to insert the output in the current buffer.

The arguments *destination* and *display* control what to do with the output from the subprocess, and whether to update the display as it comes in. For details, see the description of `call-process`, above. If *destination* is the integer 0, `call-process-region` discards the output and returns `nil` immediately, without waiting for the subprocess to finish.

The remaining arguments, *args*, are strings that specify command line arguments for the program.

The return value of `call-process-region` is just like that of `call-process`: `nil` if you told it to return without waiting; otherwise, a number or string which indicates how the subprocess terminated.

In the following example, we use `call-process-region` to run the `cat` utility, with standard input being the first five characters in buffer '`foo`' (the word '`input`'). `cat` copies its standard input into its standard output. Since the argument *destination* is `t`, this output is inserted in the current buffer.

```
---------- Buffer: foo ----------
input⋆
---------- Buffer: foo ----------
(call-process-region 1 6 "cat" nil t)
     ⇒ nil

---------- Buffer: foo ----------
inputinput⋆
---------- Buffer: foo ----------
```

The `shell-command-on-region` command uses `call-process-region` like this:

```
(call-process-region
 start end
 shell-file-name      ; Name of program.
 nil                  ; Do not delete region.
 buffer               ; Send output to buffer.
 nil                  ; No redisplay during output.
 "-c" command)        ; Arguments for the shell.
```

49.3 MS-DOS Subprocesses

On MS-DOS, you must indicate whether the data going to and from a synchronous subprocess are text or binary. Text data requires translation between the end-of-line convention used within Emacs (a single newline character) and the convention used outside Emacs (the two-character sequence, CRLF).

The variable `binary-process-input` applies to input sent to the subprocess, and `binary-process-output` applies to output received from it. A non-`nil` value means the data is non-text; `nil` means the data is text, and calls for conversion.

binary-process-input Variable

If this variable is `nil`, convert newlines to CRLF sequences in the input to a synchronous subprocess.

binary-process-output Variable

If this variable is `nil`, convert CRLF sequences to newlines in the output from a synchronous subprocess.

See Section 28.14 [Files and MS-DOS], page 419, for related information.

49.4 Creating an Asynchronous Process

After an *asynchronous process* is created, Emacs and the Lisp program both 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 in Lisp. The argument *name* specifies the name for the process object; if a process with this name already exists, then *name* is modified (by 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 specify command line arguments for the program.

In the example below, the first process is started and runs (rather, sleeps) for 100 seconds. Meanwhile, the second process is started, and given the name '`my-process<1>`' for the sake of uniqueness. It inserts the directory listing at the end of the buffer '`foo`', before the first process finishes. Then it finishes, and a message to that effect is inserted in the buffer. Much later, the first process finishes, and another message is inserted in the buffer for it.

```
(start-process "my-process" "foo" "sleep" "100")
     ⇒ #<process my-process>

(start-process "my-process" "foo" "ls" "-l" "/user/lewis/bin")
     ⇒ #<process my-process<1>>

---------- Buffer: foo ----------
total 2
lrwxrwxrwx  1 lewis     14 Jul 22 10:12 gnuemacs --> /emacs
-rwxrwxrwx  1 lewis     19 Jul 30 21:02 lemon

Process my-process<1> finished

Process my-process finished
---------- Buffer: foo ----------
```

start-process-shell-command *name buffer-or-name command* &rest *command-args* — Function

This function is like `start-process` except that it uses a shell to execute the specified command. The argument *command* is a shell command name, and *command-args* are the arguments for the shell command.

process-connection-type — Variable

This variable controls the type of device used to communicate with asynchronous subprocesses. If it is non-`nil`, then PTYs are used, when available. Otherwise, pipes are used.

PTYs are usually preferable for processes visible to the user, as in Shell mode, because they allow job control (*C-c*, *C-z*, etc.) to work between the process and its children whereas pipes do not. For subprocesses used for internal purposes by programs, it is often better to use a pipe, because they are more efficient. In addition, the total number of PTYs is limited on many systems and it is good not to waste them.

The value `process-connection-type` is used when `start-process` is called. So you can specify how to communicate with one subprocess by binding the variable around the call to `start-process`.

```
(let ((process-connection-type nil))  ; Use a pipe.
  (start-process ...))
```

To determine whether a given subprocess actually got a pipe or a PTY, use the function `process-tty-name` (see Section 49.6 [Process Information], page 679).

49.5 Deleting Processes

Deleting a process disconnects XEmacs 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*, killing it with a `SIGHUP` signal. The argument *name* may be a process, the name of a process, a buffer, or the name of a buffer.

```
(delete-process "*shell*")
     ⇒ nil
```

process-kill-without-query *process* &optional *require-query-p* Function

This function declares that XEmacs need not query the user if *process* is still running when XEmacs is exited. The process will be deleted silently. If *require-query-p* is non-`nil`, then XEmacs *will* query the user (this is the default). The return value is `t` if a query was formerly required, and `nil` otherwise.

```
(process-kill-without-query (get-process "shell"))
     ⇒ t
```

49.6 Process Information

Several functions return information about processes. `list-processes` is provided for interactive use.

list-processes Command

This command displays a listing of all living processes. In addition, it finally deletes any process whose status was '`Exited`' or '`Signaled`'. It returns `nil`.

process-list Function

This function returns a list of all processes that have not been deleted.

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

get-process *name* Function

This function returns the process named *name*, or `nil` if there is none. An error is signaled if *name* is not a string.

```
(get-process "shell")
     ⇒ #<process shell>
```

process-command *process* Function

This function returns the command that was executed to start *process*. This is a list of strings, the first string being the program executed and the rest of the strings being the arguments that were given to the program.

```
(process-command (get-process "shell"))
     ⇒ ("/bin/csh" "-i")
```

process-id *process* Function

This function returns the PID of *process*. This is an integer that distinguishes the process *process* from all other processes running on the same computer at the current time. The PID of a process is chosen by the operating system kernel when the process is started and remains constant as long as the process exists.

process-name *process* Function

This function returns the name of *process*.

process-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 XEmacs 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.

process-kill-without-query-p *process* Function

This function returns whether *process* will be killed without querying the user, if it is running when XEmacs is exited. The default value is `nil`.

process-exit-status *process* Function

This function returns the exit status of *process* or the signal number that killed it. (Use the result of `process-status` to determine which of those it is.) If *process* has not yet terminated, the value is 0.

process-tty-name *process* Function

This function returns the terminal name that *process* is using for its communication with Emacs—or `nil` if it is using pipes instead of a terminal (see `process-connection-type` in Section 49.4 [Asynchronous Processes], page 677).

49.7 Sending Input to Processes

Asynchronous subprocesses receive input when it is sent to them by XEmacs, which is done with the functions in this section. You must specify the process to send input to, and the input data to send. The data appears on the "standard input" of the subprocess.

Some operating systems have limited space for buffered input in a PTY. On these systems, Emacs sends an EOF periodically amidst the other characters, to force them through. For most programs, these EOFs do no harm.

process-send-string *process-name string* Function

This function sends *process-name* the contents of *string* as standard input. The argument *process-name* must be a process or the name of a process. If it is `nil`, the current buffer's process is used.

The function returns `nil`.

```
(process-send-string "shell<1>" "ls\n")
     ⇒ nil


---------- Buffer: *shell* ----------
...
introduction.texi             syntax-tables.texi~
introduction.texi~            text.texi
introduction.txt              text.texi~
...
---------- Buffer: *shell* ----------
```

process-send-region *process-name start end* Command

This function sends the text in the region defined by *start* and *end* as standard input to *process-name*, which is a process or a process name. (If it is `nil`, the current buffer's process is used.)

An error is signaled unless both *start* and *end* are integers or markers that indicate positions in the current buffer. (It is unimportant which number is larger.)

process-send-eof &optional *process-name* Function

This function makes *process-name* see an end-of-file in its input. The EOF comes after any text already sent to it.

If *process-name* is not supplied, or if it is `nil`, then this function sends the EOF to the current buffer's process. An error is signaled if the current buffer has no process.

The function returns *process-name*.

```
(process-send-eof "shell")
     ⇒ "shell"
```

49.8 Sending Signals to Processes

Sending a signal to a subprocess is a way of interrupting its activities. There are several different signals, each with its own meaning. The set of signals and their names is defined by the operating system. For example, the signal `SIGINT` means that the user has typed `C-c`, or that some analogous thing has happened.

Each signal has a standard effect on the subprocess. Most signals kill the subprocess, but some stop or resume execution instead. Most signals can optionally be handled by programs; if the program handles the signal, then we can say nothing in general about its effects.

The set of signals and their names is defined by the operating system; XEmacs has facilities for sending only a few of the signals that are defined. XEmacs can send signals only to its own subprocesses.

You can send signals explicitly by calling the functions in this section. XEmacs also sends signals automatically at certain times: killing a buffer sends a `SIGHUP` signal to all its associated processes; killing XEmacs 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 XEmacs subprocess. If it is non-`nil`, then the signal is sent to the current process-group of the terminal that XEmacs 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 XEmacs. 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 49.4 [Asynchronous Processes], page 677.

interrupt-process &optional *process-name current-group* Function

This function interrupts the process *process-name* by sending the signal `SIGINT`. Outside of XEmacs, 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 XEmacs 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 XEmacs.

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 XEmacs). When *current-group* is non-`nil`, you can think of this function as "typing `C-z`" on the terminal XEmacs 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 XEmacs. The argument *signal* specifies which signal to send; it should be an integer.

49.9 Receiving Output from Processes

There are two ways to receive the output that a subprocess writes to its standard output stream. The output can be inserted in a buffer, which is called the associated buffer of the process, or a function called the *filter function* can be called to act on the output. If the process has no buffer and no filter function, its output is discarded.

49.9.1 Process Buffers

A process can (and usually does) have an *associated buffer*, which is an ordinary Emacs buffer that is used for two purposes: storing the output from the process, and deciding when to kill the process. You can also use the buffer to identify a process to operate on, since in normal practice only one process is associated with any given buffer. Many applications of processes also use the buffer for editing input to be sent to the process, but this is not built into XEmacs Lisp.

Unless the process has a filter function (see Section 49.9.2 [Filter Functions], page 684), its output is inserted in the associated buffer. The position to insert the output is determined by the `process-mark`, which is then updated to point to the end of the text just inserted. Usually, but not always, the `process-mark` is at the end of the buffer.

process-buffer *process* Function

This function returns the associated buffer of the process *process*.

```
(process-buffer (get-process "shell"))
     ⇒ #<buffer *shell*>
```

process-mark *process* Function

This function returns the process marker for *process*, which is the marker that says where to insert output from the process.

If *process* does not have a buffer, `process-mark` returns a marker that points nowhere.

Insertion of process output in a buffer uses this marker to decide where to insert, and updates it to point after the inserted text. That is why successive batches of output are inserted consecutively.

Filter functions normally should use this marker in the same fashion as is done by direct insertion of output in the buffer. A good example of a filter function that uses `process-mark` is found at the end of the following section.

When the user is expected to enter input in the process buffer for transmission to the process, the process marker 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 becomes associated with no buffer.

get-buffer-process *buffer-or-name* Function

This function returns the process associated with *buffer-or-name*. If there are several processes associated with it, then one is chosen. (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>
```

Killing the process's buffer deletes the process, which kills the subprocess with a `SIGHUP` signal (see Section 49.8 [Signals to Processes], page 682).

49.9.2 Process Filter Functions

A process *filter function* is a function that receives the standard output from the associated process. If a process has a filter, then *all* output from that process is passed to the filter. The process buffer is used directly for output from the process only when there is no filter.

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 XEmacs 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` or `sleep-for` (see Section 19.7 [Waiting], page 305), or `accept-process-output` (see Section 49.9.3 [Accepting Output], page 686). 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 19.8 [Quitting], page 306.

If an error happens during execution of a filter function, it is caught automatically, so that it doesn't stop the execution of whatever program was running when the filter function was started. However, if `debug-on-error` is non-`nil`, the error-catching is turned off. This makes it possible to use the Lisp debugger to debug the filter function. See Section 16.1 [Debugger], page 217.

Many filter functions sometimes or always insert the text in the process's buffer, mimicking the actions of XEmacs 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.

In earlier Emacs versions, every filter function that did regexp searching or matching had to explicitly save and restore the match data. Now Emacs does this automatically; filter functions never need to do it explicitly. See Section 37.6 [Match Data], page 561.

A filter function that writes the output into the buffer of the process should check whether the buffer is still alive. If it tries to insert into a dead buffer, it will get an error. If the buffer is dead, `(buffer-name (process-buffer process))` returns `nil`.

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. If *filter* is `t`, then no output from the process will be accepted until the filter is changed. (Output received during this time is not discarded, but is queued, and will be processed as soon as the filter is changed.)

process-filter *process* Function

This function returns the filter function of *process*, or `nil` if it has none. `t` means that output processing has been stopped.

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
")
```

49.9.3 Accepting Output from Processes

Output from asynchronous subprocesses normally arrives only while XEmacs 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 XEmacs 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. Note that *seconds* is allowed to be a floating-point number; thus, there is no need to ever use *millisec*. (It is retained for compatibility purposes.)

The function `accept-process-output` returns non-`nil` if it did get some output, or `nil` if the timeout expired before output arrived.

49.10 Sentinels: Detecting Process Status Changes

A *process sentinel* is a function that is called whenever the associated process changes status for any reason, including signals (whether sent by XEmacs 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 XEmacs is waiting (e.g., for terminal input, or for time to elapse, or for process output). This avoids the timing errors that could result from running them at random places in the middle of other Lisp programs. A program can wait, so that sentinels will run, by calling `sit-for` or `sleep-for` (see Section 19.7 [Waiting], page 305), or `accept-process-output` (see Section 49.9.3 [Accepting Output], page 686). 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 19.8 [Quitting], page 306.

A sentinel that writes the output into the buffer of the process should check whether the buffer is still alive. If it tries to insert into a dead buffer, it will get an error. If the buffer is dead, `(buffer-name (process-buffer` *process*`))` returns `nil`.

If an error happens during execution of a sentinel, it is caught automatically, so that it doesn't stop the execution of whatever programs was running when the sentinel was started. However, if `debug-on-error` is non-`nil`, the error-catching is turned off. This makes it possible to use the Lisp debugger to debug the sentinel. See Section 16.1 [Debugger], page 217.

In earlier Emacs versions, every sentinel that did regexp searching or matching had to explicitly save and restore the match data. Now Emacs does this automatically; sentinels never need to do it explicitly. See Section 37.6 [Match Data], page 561.

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 XEmacs was waiting for keyboard input from the user at the time the sentinel or filter function was called, `nil` if it was not.

49.11 Process Window Size

set-process-window-size *process height width* Function

This function tells *process* that its logical window size is *height* by *width* characters. This is principally useful with pty's.

49.12 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 that starts the transaction. The argument *fn* is the function to call when the corresponding answer comes back; it is called with two arguments: *closure*, and the answer received.

The argument *regexp* is a regular expression that should match the entire answer, but nothing less; that's how `tq-enqueue` determines where the answer ends.

The return value of `tq-enqueue` itself is not meaningful.

tq-close *queue* Function

Shut down transaction queue *queue*, waiting for all pending transactions to complete, and then terminate the connection or child process.

Transaction queues are implemented by means of a filter function. See Section 49.9.2 [Filter Functions], page 684.

49.13 Network Connections

XEmacs Lisp programs can open TCP network connections to other processes on the same machine or other machines. A network connection is handled by Lisp much like a subprocess, and is represented by a process object. However, the process you are communicating with is not a child of the XEmacs process, so you can't kill it or send it signals. All you can do is send and receive data. `delete-process` closes the connection, but does not kill the process at the other end; that process must decide what to do about closure of the connection.

You can distinguish process objects representing network connections from those representing subprocesses with the `process-status` function. It always returns either `open` or `closed` for a network connection, and it never returns either of those values for a real subprocess. See Section 49.6 [Process Information], page 679.

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 or IP address (a string), and *service* is the name of a defined network service (a string) or a port number (an integer).

50 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 XEmacs], page 765, for related information. See also Chapter 45 [Display], page 647, for additional operating system status information pertaining to the terminal and the screen.

50.1 Starting Up XEmacs

This section describes what XEmacs does when it is started, and how you can customize these actions.

50.1.1 Summary: Sequence of Actions at Start Up

The order of operations performed (in '`startup.el`') by XEmacs when it is started up is as follows:

1. It loads the initialization library for the window system, if you are using a window system. This library's name is '`term/`*windowsystem*`-win.el`'.
2. It processes the initial options. (Some of them are handled even earlier than this.)
3. It initializes the X window frame and faces, if appropriate.
4. It runs the normal hook `before-init-hook`.
5. It loads the library '`site-start`', unless the option '`-no-site-file`' was specified. The library's file name is usually '`site-start.el`'.
6. It loads the file '`~/.emacs`' unless '`-q`' was specified on the command line. (This is not done in '`-batch`' mode.) The '`-u`' option can specify the user name whose home directory should be used instead of '`~`'.
7. It loads the library '`default`' unless `inhibit-default-init` is non-`nil`. (This is not done in '`-batch`' mode or if '`-q`' was specified on the command line.) The library's file name is usually '`default.el`'.
8. It runs the normal hook `after-init-hook`.
9. It sets the major mode according to `initial-major-mode`, provided the buffer '`*scratch*`' is still current and still in Fundamental mode.
10. It loads the terminal-specific Lisp file, if any, except when in batch mode or using a window system.
11. It displays the initial echo area message, unless you have suppressed that with `inhibit-startup-echo-area-message`.
12. It processes the action arguments from the command line.
13. It runs `term-setup-hook`.
14. It calls `frame-notice-user-settings`, which modifies the parameters of the selected frame according to whatever the init files specify.
15. It runs `window-setup-hook`. See Section 50.1.3 [Terminal-Specific], page 693.

16. It displays copyleft, nonwarranty, and basic use information, provided there were no remaining command line arguments (a few steps above) and the value of `inhibit-startup-message` is `nil`.

inhibit-startup-message User Option

This variable inhibits the initial startup messages (the nonwarranty, etc.). If it is non-`nil`, then the messages are not printed.

This variable exists so you can set it in your personal init file, once you are familiar with the contents of the startup message. Do not set this variable in the init file of a new user, or in a way that affects more than one user, because that would prevent new users from receiving the information they are supposed to see.

inhibit-startup-echo-area-message User Option

This variable controls the display of the startup echo area message. You can suppress the startup echo area message by adding text with this form to your '`.emacs`' file:

```
(setq inhibit-startup-echo-area-message
      "your-login-name")
```

Simply setting `inhibit-startup-echo-area-message` to your login name is not sufficient to inhibit the message; Emacs explicitly checks whether '`.emacs`' contains an expression as shown above. Your login name must appear in the expression as a Lisp string constant.

This way, you can easily inhibit the message for yourself if you wish, but thoughtless copying of your '`.emacs`' file will not inhibit the message for someone else.

50.1.2 The Init File: '`.emacs`'

When you start XEmacs, it normally attempts to load the file '`.emacs`' from your home directory. This file, if it exists, must contain Lisp code. It is called your *init file*. The command line switches '`-q`' and '`-u`' affect the use of the init file; '`-q`' says not to load an init file, and '`-u`' says to load a specified user's init file instead of yours. See section "Entering XEmacs" in *The XEmacs User's Manual*.

A site may have a *default init file*, which is the library named '`default.el`'. XEmacs finds the '`default.el`' file through the standard search path for libraries (see Section 14.1 [How Programs Do Loading], page 195). The XEmacs distribution does not come with this file; sites may provide one for local customizations. If the default init file exists, it is loaded whenever you start Emacs, except in batch mode or if '`-q`' is specified. But your own personal init file, if any, is loaded first; if it sets `inhibit-default-init` to a non-`nil` value, then XEmacs does not subsequently load the '`default.el`' file.

Another file for site-customization is '`site-start.el`'. Emacs loads this *before* the user's init file. You can inhibit the loading of this file with the option '`-no-site-file`'.

site-run-file Variable

This variable specifies the site-customization file to load before the user's init file. Its normal value is `"site-start"`.

If there is a great deal of code in your '.emacs' file, you should move it into another file named '*something*.el', byte-compile it (see Chapter 15 [Byte Compilation], page 205), and make your '.emacs' file load the other file using `load` (see Chapter 14 [Loading], page 195).

See section "Init File Examples" in *The XEmacs User's Manual*, for examples of how to make various commonly desired customizations in your '.emacs' file.

inhibit-default-init User Option

This variable prevents XEmacs from loading the default initialization library file for your session of XEmacs. 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, 'default.el', and/or 'site-start.el'.

50.1.3 Terminal-Specific Initialization

Each terminal type can have its own Lisp library that XEmacs loads when run on that type of terminal. For a terminal type named *termtype*, the library is called 'term/*termtype*'. XEmacs 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 XEmacs 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 XEmacs can recognize. It may also need to set or add to `function-key-map` if the Termcap entry does not specify all the terminal's function keys. See Section 50.8 [Terminal Input], page 706.

When the name of the terminal type contains a hyphen, only the part of the name before the first hyphen is significant in choosing the library name. Thus, terminal types 'aaa-48' and 'aaa-30-rv' both use the 'term/aaa' library. If necessary, the library can evaluate `(getenv "TERM")` to find the full name of the terminal type.

Your '.emacs' file can prevent the loading of the terminal-specific library by setting the variable `term-file-prefix` to `nil`. This feature is useful when experimenting with your own peculiar customizations.

You can also arrange to override some of the actions of the terminal-specific library by setting the variable `term-setup-hook`. This is a normal hook which XEmacs runs using `run-hooks` at the end of XEmacs 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 26.4 [Hooks], page 378.

term-file-prefix Variable

If the `term-file-prefix` variable is non-`nil`, XEmacs loads a terminal-specific initialization file as follows:

```
(load (concat term-file-prefix (getenv "TERM")))
```

You may set the `term-file-prefix` variable to `nil` in your '`.emacs`' file if you do not wish to load the terminal-initialization file. To do this, put the following in your '`.emacs`' file: `(setq term-file-prefix nil)`.

term-setup-hook Variable

This variable is a normal hook that XEmacs runs after loading your '`.emacs`' file, the default initialization file (if any) and the terminal-specific Lisp file.

You can use `term-setup-hook` to override the definitions made by a terminal-specific file.

window-setup-hook Variable

This variable is a normal hook which XEmacs 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`.

50.1.4 Command Line Arguments

You can use command line arguments to request various actions when you start XEmacs. Since you do not need to start XEmacs more than once per day, and will often leave your XEmacs 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 XEmacs unnecessarily often. These options exist for two reasons: to be compatible with other editors (for invocation by other programs) and to enable shell scripts to run specific Lisp programs.

This section describes how Emacs processes command line arguments, and how you can customize them.

command-line Function

This function parses the command line that XEmacs was called with, processes it, loads the user's '`.emacs`' file and displays the startup messages.

command-line-processed Variable

The value of this variable is `t` once the command line has been processed.

If you redump XEmacs by calling `dump-emacs`, you may wish to set this variable to `nil` first in order to cause the new dumped XEmacs 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 XEmacs User's Manual*.

command-line-args Variable

The value of this variable is the list of command line arguments passed to XEmacs.

command-line-functions Variable

This variable's value is a list of functions for handling an unrecognized command-line argument. Each time the next argument to be processed has no special meaning, the functions in this list are called, in order of appearance, until one of them returns a non-`nil` value.

These functions are called with no arguments. They can access the command-line argument under consideration through the variable `argi`. The remaining arguments (not including the current one) are in the variable `command-line-args-left`.

When a function recognizes and processes the argument in `argi`, it should return a non-`nil` value to say it has dealt with that argument. If it has also dealt with some of the following arguments, it can indicate that by deleting them from `command-line-args-left`.

If all of these functions return `nil`, then the argument is used as a file name to visit.

50.2 Getting out of XEmacs

There are two ways to get out of XEmacs: you can kill the XEmacs job, which exits permanently, or you can suspend it, which permits you to reenter the XEmacs process later. As a practical matter, you seldom kill XEmacs—only when you are about to log out. Suspending is much more common.

50.2.1 Killing XEmacs

Killing XEmacs means ending the execution of the XEmacs process. The parent process normally resumes control. The low-level primitive for killing XEmacs is `kill-emacs`.

kill-emacs &optional *exit-data* Function

This function exits the XEmacs process and kills it.

If *exit-data* is an integer, then it is used as the exit status of the XEmacs process. (This is useful primarily in batch operation; see Section 50.11 [Batch Mode], page 711.)

If *exit-data* is a string, its contents are stuffed into the terminal input buffer so that the shell (or whatever program next reads input) can read them.

All the information in the XEmacs process, aside from files that have been saved, is lost when the XEmacs is killed. Because killing XEmacs inadvertently can lose a lot of work, XEmacs queries for confirmation before actually terminating if you have buffers that need saving or subprocesses that are running. This is done in the function `save-buffers-kill-emacs`.

kill-emacs-query-functions Variable

After asking the standard questions, `save-buffers-kill-emacs` calls the functions in the list `kill-buffer-query-functions`, in order of appearance, with no arguments. These functions can ask for additional confirmation from the user. If any of them returns non-`nil`, XEmacs is not killed.

kill-emacs-hook Variable

This variable is a normal hook; once `save-buffers-kill-emacs` is finished with all file saving and confirmation, it runs the functions in this hook.

50.2.2 Suspending XEmacs

Suspending XEmacs means stopping XEmacs temporarily and returning control to its superior process, which is usually the shell. This allows you to resume editing later in the same XEmacs process, with the same buffers, the same kill ring, the same undo history, and so on. To resume XEmacs, 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 XEmacs. Then you would exit the shell to return to XEmacs.

Suspension is not useful with window systems such as X, because the XEmacs 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 XEmacs is an X client.

suspend-emacs *string* Function

This function stops XEmacs and returns control to the superior process. If and when the superior process resumes XEmacs, `suspend-emacs` returns `nil` to its caller in Lisp.

If *string* is non-`nil`, its characters are sent to be read as terminal input by XEmacs'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 XEmacs, `suspend-emacs` runs the normal hook `suspend-resume-hook`. See Section 26.4 [Hooks], page 378.

The next redisplay after resumption will redraw the entire screen, unless the variable `no-redraw-on-reenter` is non-`nil` (see Section 45.1 [Refresh Screen], page 647).

In the following example, note that '`pwd`' is not echoed after XEmacs 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.

50.3 Operating System Environment

XEmacs 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 XEmacs is operating on. Here is a table of the possible values:

`aix-v3` AIX.

`berkeley-unix`
Berkeley BSD.

`dgux` Data General DGUX operating system.

`gnu` A GNU system using the GNU HURD and Mach.

`hpux` Hewlett-Packard HPUX operating system.

`irix` Silicon Graphics Irix system.

`linux` A GNU system using the Linux kernel.

`ms-dos` Microsoft MS-DOS "operating system."

`next-mach`
NeXT Mach-based system.

`rtu` Masscomp RTU, UCB universe.

`unisoft-unix`
UniSoft UniPlus.

`usg-unix-v`
AT&T System V.

`vax-vms` VAX VMS.

`windows-nt`
Microsoft windows NT.

`xenix` SCO Xenix 386.

We do not wish to add new symbols to make finer distinctions unless it is absolutely necessary! In fact, we hope to eliminate some of these alternatives in the future. We recommend using `system-configuration` to distinguish between different operating systems.

system-configuration — Variable

This variable holds the three-part configuration name for the hardware/software configuration of your system, as a string. The convenient way to test parts of this string is with `string-match`.

system-name — Function

This function returns the name of the machine you are running on.

```
(system-name)
     ⇒ "prep.ai.mit.edu"
```

The symbol `system-name` is a variable as well as a function. In fact, the function returns whatever value the variable `system-name` currently holds. Thus, you can set the variable `system-name` in case Emacs is confused about the name of your system. The variable is also useful for constructing frame titles (see Section 32.3 [Frame Titles], page 476).

mail-host-address Variable

If this variable is non-`nil`, it is used instead of `system-name` for purposes of generating email addresses. For example, it is used when constructing the default value of `user-mail-address`. See Section 50.4 [User Identification], page 700. (Since this is done when XEmacs starts up, the value actually used is the one saved when XEmacs was dumped. See Section B.1 [Building XEmacs], page 765.)

getenv *var* Function

This function returns the value of the environment variable *var*, as a string. Within XEmacs, the environment variable values are kept in the Lisp variable `process-environment`.

```
(getenv "USER")
     ⇒ "lewis"

lewis@slug[10] % printenv
PATH=.:/user/lewis/bin:/usr/bin:/usr/local/bin
USER=lewis
TERM=ibmapa16
SHELL=/bin/csh
HOME=/user/lewis
```

setenv *variable value* Command

This command sets the value of the environment variable named *variable* to *value*. Both arguments should be strings. This function works by modifying `process-environment`; binding that variable with `let` is also reasonable practice.

process-environment Variable

This variable is a list of strings, each describing one environment variable. The functions `getenv` and `setenv` work by means of this variable.

```
process-environment
⇒ ("l=/usr/stanford/lib/gnuemacs/lisp"
    "PATH=.:/user/lewis/bin:/usr/class:/nfsusr/local/bin"
    "USER=lewis"
    "TERM=ibmapa16"
    "SHELL=/bin/csh"
    "HOME=/user/lewis")
```

path-separator Variable

This variable holds a string which says which character separates directories in a search path (as found in an environment variable). Its value is `":"` for Unix and GNU systems, and `";"` for MS-DOS and Windows NT.

invocation-name Variable

This variable holds the program name under which Emacs was invoked. The value is a string, and does not include a directory name.

invocation-directory Variable

This variable holds the directory from which the Emacs executable was invoked, or perhaps `nil` if that directory cannot be determined.

installation-directory Variable

If non-`nil`, this is a directory within which to look for the '`lib-src`' and '`etc`' subdirectories. This is non-`nil` when Emacs can't find those directories in their standard installed locations, but can find them in a directory related somehow to the one containing the Emacs executable.

load-average &optional *use-floats* Function

This function returns a list of the current 1-minute, 5-minute and 15-minute load averages. The values are integers that are 100 times the system load averages. (The load averages indicate the number of processes trying to run.)

When *use-floats* is non-`nil`, floats will be returned instead of integers. These floats are not multiplied by 100.

```
(load-average)
     ⇒ (169 158 164)
(load-average t)
     ⇒ (1.69921875 1.58984375 1.640625)

lewis@rocky[5] % uptime
  8:06pm  up 16 day(s), 21:57,  40 users,
 load average: 1.68, 1.59, 1.64
```

If the 5-minute or 15-minute load averages are not available, return a shortened list, containing only those averages which are available.

On some systems, this function may require special privileges to run, or it may be unimplemented for the particular system type. In that case, the function will signal an error.

emacs-pid Function

This function returns the process ID of the Emacs process.

setprv *privilege-name* &optional *setp getprv* Function

This function sets or resets a VMS privilege. (It does not exist on Unix.) The first arg is the privilege name, as a string. The second argument, *setp*, is `t` or `nil`, indicating whether the privilege is to be turned on or off. Its default is `nil`. The function returns `t` if successful, `nil` otherwise.

If the third argument, *getprv*, is non-`nil`, `setprv` does not change the privilege, but returns `t` or `nil` indicating whether the privilege is currently enabled.

50.4 User Identification

user-mail-address Variable

This holds the nominal email address of the user who is using Emacs. When Emacs starts up, it computes a default value that is usually right, but users often set this themselves when the default value is not right.

user-login-name &optional *uid* Function

If you don't specify *uid*, this function returns the name under which the user is logged in. If the environment variable `LOGNAME` is set, that value is used. Otherwise, if the environment variable `USER` is set, that value is used. Otherwise, the value is based on the effective UID, not the real UID.

If you specify *uid*, the value is the user name that corresponds to *uid* (which should be an integer).

```
(user-login-name)
     ⇒ "lewis"
```

user-real-login-name Function

This function returns the user name corresponding to Emacs's real UID. This ignores the effective UID and ignores the environment variables `LOGNAME` and `USER`.

user-full-name Variable

This variable holds the name of the user running this Emacs. It is initialized at startup time from the value of `NAME` environment variable. You can change the value of this variable to alter the result of the `user-full-name` function.

user-full-name &optional *user* Function

This function returns the full name of *user*. If *user* is `nil`, it defaults to the user running this Emacs. In that case, the value of `user-full-name` variable, if non-`nil`, will be used.

If *user* is specified explicitly, `user-full-name` variable is ignored.

```
(user-full-name)
     ⇒ "Hrvoje Niksic"
(setq user-full-name "Hrvoje \"Niksa\" Niksic")
(user-full-name)
     ⇒ "Hrvoje \"Niksa\" Niksic"
(user-full-name "hniksic")
     ⇒ "Hrvoje Niksic"
```

The symbols `user-login-name`, `user-real-login-name` and `user-full-name` are variables as well as functions. The functions return the same values that the variables hold. These variables allow you to "fake out" Emacs by telling the functions what to return. The variables are also useful for constructing frame titles (see Section 32.3 [Frame Titles], page 476).

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.

user-home-directory Function

This function returns the "`HOME`" directory of the user, and is intended to replace occurrences of "`(getenv "HOME")`". Under Unix systems, the following is done:

1. Return the value of "`(getenv "HOME")`", if set.
2. Return "/", as a fallback, but issue a warning. (Future versions of XEmacs will also attempt to lookup the `HOME` directory via `getpwent()`, but this has not yet been implemented.)

Under MS Windows, this is done:

1. Return the value of "`(getenv "HOME")`", if set.
2. If the environment variables `HOMEDRIVE` and `HOMEDIR` are both set, return the concatenation (the following description uses MS Windows environment variable substitution syntax): `%HOMEDRIVE%%HOMEDIR%`.
3. Return "C:\", as a fallback, but issue a warning.

50.5 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. It is 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 list whose first two elements are integers. Thus, you can use times obtained from `current-time` (see below) and from `file-attributes` (see Section 28.6.4 [File Attributes], page 401).

```
(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 28.6.4 [File Attributes], page 401.

current-time-zone &optional *time-value* Function

This function returns a list describing the time zone that the user is in.

The value has the form `(`*offset name*`)`. Here *offset* is an integer giving the number of seconds ahead of UTC (east of Greenwich). A negative value means west of Greenwich. The second element, *name* is a string giving the name of the time zone. Both elements change when daylight savings time begins or ends; if the user has specified a time zone that does not use a seasonal time adjustment, then the value is constant through time.

If the operating system doesn't supply all the information necessary to compute the value, both elements of the list are `nil`.

The argument *time-value*, if given, specifies a time to analyze instead of the current time. The argument should be a cons cell containing two integers, or a list whose first two elements are integers. Thus, you can use times obtained from `current-time` (see above) and from `file-attributes` (see Section 28.6.4 [File Attributes], page 401).

50.6 Time Conversion

These functions convert time values (lists of two or three integers) to strings or to calendrical information. There is also a function to convert calendrical information to a time value. You can get time values from the functions `current-time` (see Section 50.5 [Time of Day], page 702) and `file-attributes` (see Section 28.6.4 [File Attributes], page 401).

format-time-string *format-string* &optional *time* Function

This function converts *time* to a string according to *format-string*. If *time* is omitted, it defaults to the current time. The argument *format-string* may contain '`%`'-sequences which say to substitute parts of the time. Here is a table of what the '`%`'-sequences mean:

'`%a`' This stands for the abbreviated name of the day of week.

'`%A`' This stands for the full name of the day of week.

'`%b`' This stands for the abbreviated name of the month.

'`%B`' This stands for the full name of the month.

'`%c`' This is a synonym for '`%x %X`'.

'`%C`' This has a locale-specific meaning. In the default locale (named C), it is equivalent to '`%A, %B %e, %Y`'.

'`%d`' This stands for the day of month, zero-padded.

'`%D`' This is a synonym for '`%m/%d/%y`'.

'`%e`' This stands for the day of month, blank-padded.

'`%h`' This is a synonym for '`%b`'.

'`%H`' This stands for the hour (00-23).

'`%I`' This stands for the hour (00-12).

'`%j`' This stands for the day of the year (001-366).

'`%k`' This stands for the hour (0-23), blank padded.

'`%l`' This stands for the hour (1-12), blank padded.

'%m' This stands for the month (01-12).

'%M' This stands for the minute (00-59).

'%n' This stands for a newline.

'%p' This stands for 'AM' or 'PM', as appropriate.

'%r' This is a synonym for '%I:%M:%S %p'.

'%R' This is a synonym for '%H:%M'.

'%S' This stands for the seconds (00-60).

'%t' This stands for a tab character.

'%T' This is a synonym for '%H:%M:%S'.

'%U' This stands for the week of the year (01-52), assuming that weeks start on Sunday.

'%w' This stands for the numeric day of week (0-6). Sunday is day 0.

'%W' This stands for the week of the year (01-52), assuming that weeks start on Monday.

'%x' This has a locale-specific meaning. In the default locale (named C), it is equivalent to '%D'.

'%X' This has a locale-specific meaning. In the default locale (named C), it is equivalent to '%T'.

'%y' This stands for the year without century (00-99).

'%Y' This stands for the year with century.

'%Z' This stands for the time zone abbreviation.

decode-time *time* Function

This function converts a time value into calendrical information. The return value is a list of nine elements, as follows:

(*seconds minutes hour day month year dow dst zone*)

Here is what the elements mean:

sec The number of seconds past the minute, as an integer between 0 and 59.

minute The number of minutes past the hour, as an integer between 0 and 59.

hour The hour of the day, as an integer between 0 and 23.

day The day of the month, as an integer between 1 and 31.

month The month of the year, as an integer between 1 and 12.

year The year, an integer typically greater than 1900.

dow The day of week, as an integer between 0 and 6, where 0 stands for Sunday.

dst `t` if daylight savings time is effect, otherwise `nil`.

zone An integer indicating the time zone, as the number of seconds east of Greenwich.

Note that Common Lisp has different meanings for *dow* and *zone*.

encode-time *seconds minutes hour day month year* &optional *zone* — Function

This function is the inverse of `decode-time`. It converts seven items of calendrical data into a time value. For the meanings of the arguments, see the table above under `decode-time`.

Year numbers less than 100 are treated just like other year numbers. If you want them to stand for years above 1900, you must alter them yourself before you call `encode-time`.

The optional argument *zone* defaults to the current time zone and its daylight savings time rules. If specified, it can be either a list (as you would get from `current-time-zone`) or an integer (as you would get from `decode-time`). The specified zone is used without any further alteration for daylight savings time.

50.7 Timers for Delayed Execution

You can set up a timer to call a function at a specified future time.

add-timeout *secs function object* &optional *resignal* — Function

This function adds a timeout, to be signaled after the timeout period has elapsed. *secs* is a number of seconds, expressed as an integer or a float. *function* will be called after that many seconds have elapsed, with one argument, the given *object*. If the optional *resignal* argument is provided, then after this timeout expires, 'add-timeout' will automatically be called again with *resignal* as the first argument.

This function returns an object which is the *id* of this particular timeout. You can pass that object to `disable-timeout` to turn off the timeout before it has been signalled.

The number of seconds may be expressed as a floating-point number, in which case some fractional part of a second will be used. Caveat: the usable timeout granularity will vary from system to system.

Adding a timeout causes a timeout event to be returned by `next-event`, and the function will be invoked by `dispatch-event`, so if XEmacs is in a tight loop, the function will not be invoked until the next call to sit-for or until the return to top-level (the same is true of process filters).

WARNING: if you are thinking of calling add-timeout from inside of a callback function as a way of resignalling a timeout, think again. There is a race condition. That's why the *resignal* argument exists.

(NOTE: In FSF Emacs, this function is called `run-at-time` and has different semantics.)

disable-timeout *id* Function

Cancel the requested action for *id*, which should be a value previously returned by `add-timeout`. This cancels the effect of that call to `add-timeout`; the arrival of the specified time will not cause anything special to happen. (NOTE: In FSF Emacs, this function is called `cancel-timer`.)

50.8 Terminal Input

This section describes functions and variables for recording or manipulating terminal input. See Chapter 45 [Display], page 647, for related functions.

50.8.1 Input Modes

set-input-mode *interrupt flow meta quit-char* Function

This function sets the mode for reading keyboard input. If *interrupt* is non-null, then XEmacs uses input interrupts. If it is `nil`, then it uses CBREAK mode. When XEmacs communicates directly with X, it ignores this argument and uses interrupts if that is the way it knows how to communicate.

If *flow* is non-`nil`, then XEmacs uses XON/XOFF (*C-q*, *C-s*) flow control for output to the terminal. This has no effect except in CBREAK mode. See Section 50.10 [Flow Control], page 710.

The default 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`, XEmacs converts characters with the 8th bit set into Meta characters. If *meta* is `nil`, XEmacs disregards the 8th bit; this is necessary when the terminal uses it as a parity bit. If *meta* is neither `t` nor `nil`, XEmacs 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 19.8 [Quitting], page 306.

The `current-input-mode` function returns the input mode settings XEmacs 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 XEmacs is using interrupt-driven input. If `nil`, Emacs is using CBREAK mode.

flow
: is non-`nil` if XEmacs 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 `t` if XEmacs treats the eighth bit of input characters as the meta bit; `nil` means XEmacs clears the eighth bit of every input character; any other value means XEmacs uses all eight bits as the basic character code.

quit is the character XEmacs currently uses for quitting, usually `C-g`.

50.8.2 Translating Input Events

This section describes features for translating input events into other input events before they become part of key sequences.

function-key-map Variable

This variable holds a keymap that describes the character sequences sent by function keys on an ordinary character terminal. This keymap uses the same 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 XEmacs 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. XEmacs comes with terminal-specific files for many common terminals; their main purpose is to make entries in `function-key-map` beyond those that can be deduced from Termcap and Terminfo. See Section 50.1.3 [Terminal-Specific], page 693.

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`.

You can use `function-key-map` or `key-translation-map` for more than simple aliases, by using a function, instead of a key sequence, as the "translation" of a key. Then this function is called to compute the translation of that key.

The key translation function receives one argument, which is the prompt that was specified in `read-key-sequence`—or `nil` if the key sequence is being read by the editor command loop. In most cases you can ignore the prompt value.

If the function reads input itself, it can have the effect of altering the event that follows. For example, here's how to define *C-c h* to turn the character that follows into a Hyper character:

```
(defun hyperify (prompt)
  (let ((e (read-event)))
    (vector (if (numberp e)
                (logior (lsh 1 20) e)
              (if (memq 'hyper (event-modifiers e))
                  e
                (add-event-modifier "H-" e))))))

(defun add-event-modifier (string e)
  (let ((symbol (if (symbolp e) e (car e))))
    (setq symbol (intern (concat string
                                 (symbol-name symbol))))
    (if (symbolp e)
        symbol
      (cons symbol (cdr e)))))

(define-key function-key-map "\C-ch" 'hyperify)
```

The '`iso-transl`' library uses this feature to provide a way of inputting non-ASCII Latin-1 characters.

50.8.3 Recording Input

recent-keys &optional *number* Function

This function returns a vector containing recent input events from the keyboard or mouse. By default, 100 events are recorded, which is how many `recent-keys` returns.

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 that invoked the macros.)

If *number* is specified, not more than *number* events will be returned. You may change the number of stored events using `set-recent-keys-ring-size`.

recent-keys-ring-size Function

This function returns the number of recent events stored internally. This is also the maximum number of events `recent-keys` can return. By default, 100 events are stored.

set-recent-keys-ring-size *size* Function

This function changes the number of events stored by XEmacs and returned by `recent-keys`.

For example, `(set-recent-keys-ring-size 250)` will make XEmacs remember last 250 events and will make `recent-keys` return last 250 events by default.

open-dribble-file *filename* Command

This function opens a *dribble file* named *filename*. When a dribble file is open, each input event from the keyboard or mouse (but not those from keyboard macros) is written in that file. A non-character event is expressed using its printed representation surrounded by '`<...>`'.

You close the dribble file by calling this function with an argument of `nil`.

This function is normally used to record the input necessary to trigger an XEmacs bug, for the sake of a bug report.

```
(open-dribble-file "~/dribble")
     ⇒ nil
```

See also the `open-termscript` function (see Section 50.9 [Terminal Output], page 709).

50.9 Terminal Output

The terminal output functions send output to the terminal or keep track of output sent to the terminal. The function `device-baud-rate` tells you what XEmacs thinks is the output speed of the terminal.

device-baud-rate &optional *device* Function

This function's value is the output speed of the terminal associated with *device*, as far as XEmacs knows. *device* defaults to the selected device (usually the only device) if omitted. Changing this value does not change the speed of actual data transmission, but the value is used for calculations such as padding. This value has no effect for window-system devices. (This is different in FSF Emacs, where the baud rate also affects decisions about whether to scroll part of the screen or repaint, even when using a window system.)

The value is measured in bits per second.

XEmacs attempts to automatically initialize the baud rate by querying the terminal. If you are running across a network, however, and different parts of the network work are at different baud rates, the value returned by XEmacs 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 XEmacs and other programs can get the proper value, but others do not. If XEmacs has the wrong value, it makes decisions that are less than optimal. To fix the problem, use `set-device-baud-rate`.

set-device-baud-rate &optional *device* Function

This function sets the output speed of *device*. See `device-baud-rate`. *device* defaults to the selected device (usually the only device) if omitted.

send-string-to-terminal *char-or-string* &optional *stdout-p* *device* Function

This function sends *char-or-string* to the terminal without alteration. Control characters in *char-or-string* have terminal-dependent effects.

If *device* is `nil`, this function writes to XEmacs's stderr, or to stdout if *stdout-p* is non-`nil`. Otherwise, *device* should be a tty or stream device, and the function writes to the device's normal or error output, according to *stdout-p*.

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 XEmacs to the terminal. (If there are multiple tty or stream devices, all characters sent to all such devices are recorded.) The function returns `nil`. Termscript files are useful for investigating problems where XEmacs garbles the screen, problems that are due to incorrect Termcap entries or to undesirable settings of terminal options more often than to actual XEmacs bugs. Once you are certain which characters were actually output, you can determine reliably whether they correspond to the Termcap specifications in use.

A `nil` value for *filename* stops recording terminal output.

See also `open-dribble-file` in Section 50.8 [Terminal Input], page 706.

```
(open-termscript "../junk/termscript")
     ⇒ nil
```

50.10 Flow Control

This section attempts to answer the question "Why does XEmacs 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 uncontroversial. XEmacs, 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 XEmacs 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", XEmacs's or that of some terminal and concentrator manufacturers? This question has no simple answer.

One reason why we are reluctant to cater to the problems caused by `C-s` and `C-q` is that they are gratuitous. There are other techniques (albeit less common in practice) for flow control that preserve transparency of the character stream. Note also that their use for flow control is not an official standard. Interestingly, on the model 33 teletype with a paper tape punch (which is very old), `C-s` and `C-q` were sent by the computer to turn the punch on and off!

As X servers and other window systems replace character-only terminals, this problem is gradually being cured. For the mean time, XEmacs 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 50.8.2 [Translating Input], page 707).

You can use the function `enable-flow-control-on` in your '`.emacs`' file to enable flow control automatically on certain terminal types.

enable-flow-control-on &rest *termtypes* Function

> This function enables flow control, and the aliases `C-\` and `C-^`, if the terminal type is one of *termtypes*. For example:

```
(enable-flow-control-on "vt200" "vt300" "vt101" "vt131")
```

Here is how `enable-flow-control` does its job:

1. It sets CBREAK mode for terminal input, and tells the operating system to handle flow control, with `(set-input-mode nil t)`.
2. It sets up `keyboard-translate-table` to translate `C-\` and `C-^` into `C-s` and `C-q`. Except at its very lowest level, XEmacs never knows that the characters typed were anything but `C-s` and `C-q`, so you can in effect type them as `C-\` and `C-^` even when they are input for other commands. See Section 50.8.2 [Translating Input], page 707.

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 XEmacs uses a smaller speed when calculating the padding needed. See Section 50.9 [Terminal Output], page 709.

50.11 Batch Mode

The command line option '`-batch`' causes XEmacs to run noninteractively. In this mode, XEmacs 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, XEmacs 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 XEmacs's standard error descriptor when in batch mode. Thus, XEmacs behaves much like a noninteractive application program. (The echo area output that XEmacs itself normally generates, such as command echoing, is suppressed entirely.)

noninteractive Function

This function returns non-`nil` when XEmacs is running in batch mode.

noninteractive Variable

This variable is non-`nil` when XEmacs is running in batch mode. Setting this variable to `nil`, however, will not change whether XEmacs is running in batch mode, and will not change the return value of the `noninteractive` function.

51 Functions Specific to the X Window System

XEmacs provides the concept of *devices*, which generalizes connections to an X server, a TTY device, etc. Most information about an X server that XEmacs is connected to can be determined through general console and device functions. See Chapter 33 [Consoles and Devices], page 483. However, there are some features of the X Window System that do not generalize well, and they are covered specially here.

51.1 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 XEmacs by symbols. X clients including XEmacs can read or set the selection for any given type.

x-own-selection *data* &optional *type* Function

This function sets a "selection" in the X server. It takes two arguments: a value, *data*, and the selection type *type* to assign it to. *data* may be a string, a cons of two markers, or an extent. In the latter cases, the selection is considered to be the text between the markers, or between the extent's endpoints.

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`.

(In FSF Emacs, this function is called `x-set-selection` and takes different arguments.)

x-get-selection Function

This function accesses selections set up by XEmacs or by other X clients. It returns the value of the current primary selection.

x-disown-selection &optional *secondary-p* Function

Assuming we own the selection, this function disowns it. If *secondary-p* is non-`nil`, the secondary selection instead of the primary selection is discarded.

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 XEmacs supports them for the sake of X clients that still use them.

x-get-cutbuffer &optional *n* Function

This function returns the contents of cut buffer number *n*. (This function is called `x-get-cut-buffer` in FSF Emacs.)

x-store-cutbuffer *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. (This function is called `x-set-cut-buffer` in FSF Emacs.)

51.2 X Server

This section describes how to access and change the overall status of the X server XEmacs is using.

51.2.1 Resources

default-x-device Function

This function return the default X device for resourcing. This is the first-created X device that still exists.

x-get-resource *name class type* &optional *locale device noerror* Function

This function retrieves a resource value from the X resource manager.

- The first arg is the name of the resource to retrieve, such as '`"font"`'.
- The second arg is the class of the resource to retrieve, like '`"Font"`'.
- The third arg should be one of the symbols `string`, `integer`, `natnum`, or `boolean`, specifying the type of object that the database is searched for.
- The fourth arg is the locale to search for the resources on, and can currently be a a buffer, a frame, a device, or the symbol `global`. If omitted, it defaults to `global`.
- The fifth arg is the device to search for the resources on. (The resource database for a particular device is constructed by combining non-device-specific resources such any command-line resources specified and any app-defaults files found [or the fallback resources supplied by XEmacs, if no app-defaults file is found] with device-specific resources such as those supplied using '`xrdb`'.) If omitted, it defaults to the device of *locale*, if a device can be derived (i.e. if *locale* is a frame or device), and otherwise defaults to the value of `default-x-device`.
- The sixth arg *noerror*, if non-`nil`, means do not signal an error if a bogus resource specification was retrieved (e.g. if a non-integer was given when an integer was requested). In this case, a warning is issued instead.

The resource names passed to this function are looked up relative to the locale.

If you want to search for a subresource, you just need to specify the resource levels in *name* and *class*. For example, *name* could be '`"modeline.attributeFont"`', and *class* '`"Face.AttributeFont"`'.

Specifically,

1. If *locale* is a buffer, a call

   ```
   (x-get-resource "foreground" "Foreground" 'string some-buffer)
   ```

   is an interface to a C call something like

   ```
   XrmGetResource (db, "xemacs.buffer.buffer-name.foreground",
   "Emacs.EmacsLocaleType.EmacsBuffer.Foreground",
   "String");
   ```

2. If *locale* is a frame, a call

```
(x-get-resource "foreground" "Foreground" 'string some-frame)
```

is an interface to a C call something like

```
XrmGetResource (db, "xemacs.frame.frame-name.foreground",
"Emacs.EmacsLocaleType.EmacsFrame.Foreground",
"String");
```

3. If *locale* is a device, a call

```
(x-get-resource "foreground" "Foreground" 'string some-device)
```

is an interface to a C call something like

```
XrmGetResource (db, "xemacs.device.device-name.foreground",
"Emacs.EmacsLocaleType.EmacsDevice.Foreground",
"String");
```

4. If *locale* is the symbol `global`, a call

```
(x-get-resource "foreground" "Foreground" 'string 'global)
```

is an interface to a C call something like

```
XrmGetResource (db, "xemacs.foreground",
"Emacs.Foreground",
"String");
```

Note that for `global`, no prefix is added other than that of the application itself; thus, you can use this locale to retrieve arbitrary application resources, if you really want to.

The returned value of this function is `nil` if the queried resource is not found. If *type* is `string`, a string is returned, and if it is `integer`, an integer is returned. If *type* is `boolean`, then the returned value is the list `(t)` for true, `(nil)` for false, and is `nil` to mean "unspecified".

x-put-resource *resource-line* &optional *device* — Function

This function adds a resource to the resource database for *device*. *resource-line* specifies the resource to add and should be a standard resource specification.

x-emacs-application-class — Variable

This variable holds The X application class of the XEmacs process. This controls, among other things, the name of the "app-defaults" file that XEmacs will use. For changes to this variable to take effect, they must be made before the connection to the X server is initialized, that is, this variable may only be changed before XEmacs is dumped, or by setting it in the file '`lisp/term/x-win.el`'.

By default, this variable is nil at startup. When the connection to the X server is first initialized, the X resource database will be consulted and the value will be set according to whether any resources are found for the application class "XEmacs".

51.2.2 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 corresponding to a particular device. The device argument is generally optional and defaults to the selected device.

x-server-version &optional *device* Function

This function returns the list of version numbers of the X server *device* is on. The returned value is a list of three integers: the major and minor version numbers of the X protocol in use, and the vendor-specific release number.

x-server-vendor &optional *device* Function

This function returns the vendor supporting the X server *device* is on.

x-display-visual-class &optional *device* Function

This function returns the visual class of the display *device* is on. The value is one of the symbols `static-gray`, `gray-scale`, `static-color`, `pseudo-color`, `true-color`, and `direct-color`. (Note that this is different from previous versions of XEmacs, which returned `StaticGray`, `GrayScale`, etc.)

51.2.3 Restricting Access to the Server by Other Apps

x-grab-keyboard &optional *device* Function

This function grabs the keyboard on the given device (defaulting to the selected one). So long as the keyboard is grabbed, all keyboard events will be delivered to XEmacs – it is not possible for other X clients to eavesdrop on them. Ungrab the keyboard with `x-ungrab-keyboard` (use an `unwind-protect`). Returns `t` if the grab was successful; `nil` otherwise.

x-ungrab-keyboard &optional *device* Function

This function releases a keyboard grab made with `x-grab-keyboard`.

x-grab-pointer &optional *device cursor ignore-keyboard* Function

This function grabs the pointer and restricts it to its current window. If optional *device* argument is `nil`, the selected device will be used. If optional *cursor* argument is non-`nil`, change the pointer shape to that until `x-ungrab-pointer` is called (it should be an object returned by the `make-cursor` function). If the second optional argument *ignore-keyboard* is non-`nil`, ignore all keyboard events during the grab. Returns `t` if the grab is successful, `nil` otherwise.

x-ungrab-pointer &optional *device* Function

This function releases a pointer grab made with `x-grab-pointer`. If optional first arg *device* is `nil` the selected device is used. If it is `t` the pointer will be released on all X devices.

51.3 Miscellaneous X Functions and Variables

x-bitmap-file-path Variable

This variable holds a list of the directories in which X bitmap files may be found. If `nil`, this is initialized from the '`"*bitmapFilePath"`' resource. This is used by the `make-image-instance` function (however, note that if the environment variable '`XBMLANGPATH`' is set, it is consulted first).

x-library-search-path Variable

This variable holds the search path used by `read-color` to find '`rgb.txt`'.

x-valid-keysym-name-p *keysym* Function

This function returns true if *keysym* names a keysym that the X library knows about. Valid keysyms are listed in the files '`/usr/include/X11/keysymdef.h`' and in '`/usr/lib/X11/XKeysymDB`', or whatever the equivalents are on your system.

x-window-id &optional *frame* Function

This function returns the ID of the X11 window. This gives us a chance to manipulate the Emacs window from within a different program. Since the ID is an unsigned long, we return it as a string.

x-allow-sendevents Variable

If non-`nil`, synthetic events are allowed. `nil` means they are ignored. Beware: allowing XEmacs to process SendEvents opens a big security hole.

x-debug-mode *arg* &optional *device* Function

With a true arg, make the connection to the X server synchronous. With false, make it asynchronous. Synchronous connections are much slower, but are useful for debugging. (If you get X errors, make the connection synchronous, and use a debugger to set a breakpoint on `x_error_handler`. Your backtrace of the C stack will now be useful. In asynchronous mode, the stack above `x_error_handler` isn't helpful because of buffering.) If *device* is not specified, the selected device is assumed.

Calling this function is the same as calling the C function `XSynchronize`, or starting the program with the '`-sync`' command line argument.

x-debug-events Variable

If non-zero, debug information about events that XEmacs sees is displayed. Information is displayed on stderr. Currently defined values are:

- 1 == non-verbose output
- 2 == verbose output

52 ToolTalk Support

52.1 XEmacs ToolTalk API Summary

The XEmacs Lisp interface to ToolTalk is similar, at least in spirit, to the standard C ToolTalk API. Only the message and pattern parts of the API are supported at present; more of the API could be added if needed. The Lisp interface departs from the C API in a few ways:

- ToolTalk is initialized automatically at XEmacs startup-time. Messages can only be sent other ToolTalk applications connected to the same X11 server that XEmacs is running on.
- There are fewer entry points; polymorphic functions with keyword arguments are used instead.
- The callback interface is simpler and marginally less functional. A single callback may be associated with a message or a pattern; the callback is specified with a Lisp symbol (the symbol should have a function binding).
- The session attribute for messages and patterns is always initialized to the default session.
- Anywhere a ToolTalk enum constant, e.g. '`TT_SESSION`', is valid, one can substitute the corresponding symbol, e.g. `'TT_SESSION`. This simplifies building lists that represent messages and patterns.

52.2 Sending Messages

52.2.1 Example of Sending Messages

Here's a simple example that sends a query to another application and then displays its reply. Both the query and the reply are stored in the first argument of the message.

```
(defun tooltalk-random-query-handler (msg)
  (let ((state (get-tooltalk-message-attribute msg 'state)))
    (cond
      ((eq state 'TT_HANDLED)
       (message (get-tooltalk-message-attribute msg arg_val 0)))
      ((memq state '(TT_FAILED TT_REJECTED))
       (message "Random query turns up nothing")))))

(defvar random-query-message
  '(   class TT_REQUEST
       scope TT_SESSION
     address TT_PROCEDURE
          op "random-query"
        args '((TT_INOUT "?" "string"))
    callback tooltalk-random-query-handler))
```

```
(let ((m (make-tooltalk-message random-query-message)))
  (send-tooltalk-message m))
```

52.2.2 Elisp Interface for Sending Messages

make-tooltalk-message *attributes* Function

Create a ToolTalk message and initialize its attributes. The value of *attributes* must be a list of alternating keyword/values, where keywords are symbols that name valid message attributes. For example:

```
(make-tooltalk-message
  '(class TT_NOTICE
    scope TT_SESSION
    address TT_PROCEDURE
    op "do-something"
    args ("arg1" 12345 (TT_INOUT "arg3" "string"))))
```

Values must always be strings, integers, or symbols that represent ToolTalk constants. Attribute names are the same as those supported by `set-tooltalk-message-attribute`, plus `args`.

The value of `args` should be a list of message arguments where each message argument has the following form:

> '`(mode [value [type]])`' or just '`value`'

Where *mode* is one of `TT_IN`, `TT_OUT`, or `TT_INOUT` and *type* is a string. If *type* isn't specified then `int` is used if *value* is a number; otherwise `string` is used. If *type* is `string` then *value* is converted to a string (if it isn't a string already) with `prin1-to-string`. If only a value is specified then *mode* defaults to `TT_IN`. If *mode* is `TT_OUT` then *value* and *type* don't need to be specified. You can find out more about the semantics and uses of ToolTalk message arguments in chapter 4 of the *ToolTalk Programmer's Guide*.

send-tooltalk-message *msg* Function

Send the message on its way. Once the message has been sent it's almost always a good idea to get rid of it with `destroy-tooltalk-message`.

return-tooltalk-message *msg* &optional *mode* Function

Send a reply to this message. The second argument can be `reply`, `reject` or `fail`; the default is `reply`. Before sending a reply, all message arguments whose mode is `TT_INOUT` or `TT_OUT` should have been filled in – see `set-tooltalk-message-attribute`.

get-tooltalk-message-attribute *msg attribute* &optional *argn* Function

Returns the indicated ToolTalk message attribute. Attributes are identified by symbols with the same name (underscores and all) as the suffix of the ToolTalk '`tt_message_<attribute>`' function that extracts the value. String attribute values are copied and enumerated type values (except disposition) are converted to symbols; e.g. '`TT_HANDLER`' is `'TT_HANDLER`, '`uid`' and '`gid`' are represented by fixnums (small integers), '`opnum`' is converted to a string, and '`disposition`' is converted to a fixnum. We convert '`opnum`' (a C int) to a string (e.g. `123` ⇒ `"123"`) because there's no guarantee that opnums will fit within the range of XEmacs Lisp integers.

[TBD] Use the `plist` attribute instead of C API `user` attribute for user-defined message data. To retrieve the value of a message property, specify the indicator for *argn*. For example, to get the value of a property called `rflag`, use

```
(get-tooltalk-message-attribute msg 'plist 'rflag)
```

To get the value of a message argument use one of the `arg_val` (strings), `arg_ival` (integers), or `arg_bval` (strings with embedded nulls), attributes. For example, to get the integer value of the third argument:

```
(get-tooltalk-message-attribute msg 'arg_ival 2)
```

As you can see, argument numbers are zero-based. The type of each arguments can be retrieved with the `arg_type` attribute; however ToolTalk doesn't define any semantics for the string value of `arg_type`. Conventionally `string` is used for strings and `int` for 32 bit integers. Note that XEmacs Lisp stores the lengths of strings explicitly (unlike C) so treating the value returned by `arg_bval` like a string is fine.

set-tooltalk-message-attribute *value msg attribute* &optional *argn* — Function

Initialize one ToolTalk message attribute.

Attribute names and values are the same as for `get-tooltalk-message-attribute`. A property list is provided for user data (instead of the `user` message attribute); see `get-tooltalk-message-attribute`.

Callbacks are handled slightly differently than in the C ToolTalk API. The value of *callback* should be the name of a function of one argument. It will be called each time the state of the message changes. This is usually used to notice when the message's state has changed to `TT_HANDLED` (or `TT_FAILED`), so that reply argument values can be used.

If one of the argument attributes is specified as `arg_val`, `arg_ival`, or `arg_bval`, then *argn* must be the number of an already created argument. Arguments can be added to a message with `add-tooltalk-message-arg`.

add-tooltalk-message-arg *msg mode type* &optional *value* — Function

Append one new argument to the message. *mode* must be one of `TT_IN`, `TT_INOUT`, or `TT_OUT`, *type* must be a string, and *value* can be a string or an integer. ToolTalk doesn't define any semantics for *type*, so only the participants in the protocol you're using need to agree what types mean (if anything). Conventionally `string` is used for strings and `int` for 32 bit integers. Arguments can initialized by providing a value or with `set-tooltalk-message-attribute`; the latter is necessary if you want to initialize the argument with a string that can contain embedded nulls (use `arg_bval`).

create-tooltalk-message — Function

Create a new ToolTalk message. The message's session attribute is initialized to the default session. Other attributes can be initialized with `set-tooltalk-`

message-attribute. make-tooltalk-message is the preferred way to create and initialize a message.

destroy-tooltalk-message *msg* Function

Apply 'tt_message_destroy' to the message. It's not necessary to destroy messages after they've been processed by a message or pattern callback, the Lisp/ToolTalk callback machinery does this for you.

52.3 Receiving Messages

52.3.1 Example of Receiving Messages

Here's a simple example of a handler for a message that tells XEmacs to display a string in the mini-buffer area. The message operation is called 'emacs-display-string'. Its first (0th) argument is the string to display.

```
(defun tooltalk-display-string-handler (msg)
  (message (get-tooltalk-message-attribute msg 'arg_val 0)))

(defvar display-string-pattern
  '(category TT_HANDLE
       scope TT_SESSION
          op "emacs-display-string"
    callback tooltalk-display-string-handler))

(let ((p (make-tooltalk-pattern display-string-pattern)))
  (register-tooltalk-pattern p))
```

52.3.2 Elisp Interface for Receiving Messages

make-tooltalk-pattern *attributes* Function

Create a ToolTalk pattern and initialize its attributes. The value of attributes must be a list of alternating keyword/values, where keywords are symbols that name valid pattern attributes or lists of valid attributes. For example:

```
(make-tooltalk-pattern
  '(category TT_OBSERVE
       scope TT_SESSION
          op ("operation1" "operation2")
        args ("arg1" 12345 (TT_INOUT "arg3" "string"))))
```

Attribute names are the same as those supported by add-tooltalk-pattern-attribute, plus 'args.

Values must always be strings, integers, or symbols that represent ToolTalk constants or lists of same. When a list of values is provided all of the list elements are added to the attribute. In the example above, messages whose 'op' attribute is '"operation1"' or '"operation2"' would match the pattern.

The value of *args* should be a list of pattern arguments where each pattern argument has the following form:

'`(mode [value [type]])`' or just '`value`'

Where *mode* is one of `TT_IN`, `TT_OUT`, or `TT_INOUT` and *type* is a string. If *type* isn't specified then `int` is used if *value* is a number; otherwise `string` is used. If *type* is `string` then *value* is converted to a string (if it isn't a string already) with `prin1-to-string`. If only a value is specified then *mode* defaults to `TT_IN`. If *mode* is `TT_OUT` then *value* and *type* don't need to be specified. You can find out more about the semantics and uses of ToolTalk pattern arguments in chapter 3 of the *ToolTalk Programmer's Guide*.

register-tooltalk-pattern *pat* — Function

XEmacs will begin receiving messages that match this pattern.

unregister-tooltalk-pattern *pat* — Function

XEmacs will stop receiving messages that match this pattern.

add-tooltalk-pattern-attribute *value pat indicator* — Function

Add one value to the indicated pattern attribute. The names of attributes are the same as the ToolTalk accessors used to set them less the '`tooltalk_pattern_`' prefix and the '`_add`' suffix. For example, the name of the attribute for the '`tt_pattern_disposition_add`' attribute is `disposition`. The `category` attribute is handled specially, since a pattern can only be a member of one category (`TT_OBSERVE` or `TT_HANDLE`).

Callbacks are handled slightly differently than in the C ToolTalk API. The value of *callback* should be the name of a function of one argument. It will be called each time the pattern matches an incoming message.

add-tooltalk-pattern-arg *pat mode type value* — Function

Add one fully-specified argument to a ToolTalk pattern. *mode* must be one of `TT_IN`, `TT_INOUT`, or `TT_OUT`. *type* must be a string. *value* can be an integer, string or `nil`. If *value* is an integer then an integer argument ('`tt_pattern_iarg_add`') is added; otherwise a string argument is added. At present there's no way to add a binary data argument.

create-tooltalk-pattern — Function

Create a new ToolTalk pattern and initialize its session attribute to be the default session.

destroy-tooltalk-pattern *pat* — Function

Apply '`tt_pattern_destroy`' to the pattern. This effectively unregisters the pattern.

describe-tooltalk-message *msg* &optional *stream* — Function

Print the message's attributes and arguments to *stream*. This is often useful for debugging.

53 LDAP Support

XEmacs can be linked with a LDAP client library to provide Elisp primitives to access directory servers using the Lightweight Directory Access Protocol.

53.1 Building XEmacs with LDAP support

LDAP support must be added to XEmacs at build time since it requires linking to an external LDAP client library. As of 21.0, XEmacs has been successfully built and tested with

- University of Michigan's LDAP 3.3 (`http://www.umich.edu/~dirsvcs/ldap/`)
- LDAP SDK 1.0 from Netscape Corp. (`http://developer.netscape.com/`)

Other libraries conforming to RFC 1823 will probably work also but may require some minor tweaking at C level.

The standard XEmacs configure script autodetects an installed LDAP library provided the library itself and the corresponding header files can be found in the library and include paths. A successful detection will be signalled in the final output of the configure script.

53.2 XEmacs LDAP API

XEmacs LDAP API consists of two layers: a low-level layer which tries to stay as close as possible to the C API (where practical) and a higher-level layer which provides more convenient primitives to effectively use LDAP.

As of XEmacs 21.0, only interfaces to basic LDAP search functions are provided, broader support is planned in future versions.

53.2.1 LDAP Variables

ldap-default-host Variable

The default LDAP server

ldap-default-port Variable

Default TCP port for LDAP connections. Initialized from the LDAP library. Default value is 389.

ldap-default-base Variable

Default base for LDAP searches. This is a string using the syntax of RFC 1779. For instance, "oME, c" limits the search to the Acme organization in the United States.

ldap-host-parameters-alist Variable

An alist of per host options for LDAP transactions. The list elements look like `(HOST PROP1 VAL1 PROP2 VAL2 ...)` *host* is the name of an LDAP server. *propn* and *valn* are property/value pairs describing parameters for the server. Valid properties:

`binddn` The distinguished name of the user to bind as. This may look like '`c, ome, cnnny Bugs`', see RFC 1779 for details.

`passwd` The password to use for authentication.

`auth` The authentication method to use, possible values depend on the LDAP library XEmacs was compiled with, they may include `simple`, `krbv41` and `krbv42`.

`base` The base for the search. This may look like '`c, ome`', see RFC 1779 for syntax details.

`scope` One of the symbols `base`, `onelevel` or `subtree` indicating the scope of the search limited to a base object, to a single level or to the whole subtree.

`deref` The dereference policy is one of the symbols `never`, `always`, `search` or `find` and defines how aliases are dereferenced.

- `never` Aliases are never dereferenced
- `always` Aliases are always dereferenced
- `search` Aliases are dereferenced when searching
- `find` Aliases are dereferenced when locating the base object for the search

`timelimit`
The timeout limit for the connection in seconds.

`sizelimit`
The maximum number of matches to return for searches performed on this connection.

53.2.2 The High-Level LDAP API

As of this writing the high-level Lisp LDAP API only provides for LDAP searches. Further support is planned in the future.

The `ldap-search` function provides the most convenient interface to perform LDAP searches. It opens a connection to a host, performs the query and cleanly closes the connection thus insulating the user from all the details of the low-level interface such as LDAP Lisp objects see Section 53.2.3 [The Low-Level LDAP API], page 726

ldap-search *filter* &optional *host attributes attrsonly* Function

Perform an LDAP search. *filter* is the search filter see Section 53.3 [Syntax of Search Filters], page 728 *host* is the LDAP host on which to perform the search *attributes* is the specific attributes to retrieve, `nil` means retrieve all *attrsonly* if non-`nil` retrieves the attributes only without their associated values. Additional search parameters can be specified through `ldap-host-parameters-alist`.

53.2.3 The Low-Level LDAP API

53.2.3.1 The LDAP Lisp Object

An internal built-in `ldap` lisp object represents a LDAP connection.

ldapp *object* Function
: This function returns non-`nil` if *object* is a `ldap` object.

ldap-host *ldap* Function
: Return the server host of the connection represented by *ldap*

ldap-live-p *ldap* Function
: Return non-`nil` if *ldap* is an active LDAP connection

53.2.3.2 Opening and Closing a LDAP Connection

ldap-open *host* &optional *plist* Function
: Open a LDAP connection to *host*. *plist* is a property list containing additional parameters for the connection. Valid keys in that list are:

`port` The TCP port to use for the connection if different from `ldap-default-port` or the library builtin value

`auth` The authentication method to use, possible values depend on the LDAP library XEmacs was compiled with, they may include `simple`, `krbv41` and `krbv42`.

`binddn` The distinguished name of the user to bind as. This may look like '`c, ome, cnnny Bugs`', see RFC 1779 for details.

`passwd` The password to use for authentication.

`deref` The dereference policy is one of the symbols `never`, `always`, `search` or `find` and defines how aliases are dereferenced.

`never` Aliases are never dereferenced

`always` Aliases are always dereferenced

`search` Aliases are dereferenced when searching

`find` Aliases are dereferenced when locating the base object for the search

The default is `never`.

`timelimit`
: The timeout limit for the connection in seconds.

`sizelimit`
: The maximum number of matches to return for searches performed on this connection.

ldap-close *ldap* Function
: Close the connection represented by *ldap*

53.2.3.3 Searching on a LDAP Server (Low-level)

ldap-search-internal is the low-level primitive to perform a search on a LDAP server. It works directly on an open LDAP connection thus requiring a preliminary call to ldap-open. Multiple searches can be made on the same connection, then the session must be closed with ldap-close.

ldap-search-internal *ldap filter base scope attrs attrsonly* Function

Perform a search on an open connection *ldap* created with ldap-open. *filter* is a filter string for the search see Section 53.3 [Syntax of Search Filters], page 728 *base* is the distinguished name at which to start the search. *scope* is one of the symbols base, onelevel or subtree indicating the scope of the search limited to a base object, to a single level or to the whole subtree. The default is subtree. attrs is a list of strings indicating which attributes to retrieve for each matching entry. If nil all available attributes are returned. If attrsonly is non-nil then only the attributes are retrieved, not their associated values The function returns a list of matching entries. Each entry being itself an alist of attribute/values.

53.3 Syntax of Search Filters

LDAP search functions use RFC1558 syntax to describe the search filter. In that syntax simple filters have the form:

```
(<attr> <filtertype> <value>)
```

<attr> is an attribute name such as cn for Common Name, o for Organization, etc...

<value> is the corresponding value. This is generally an exact string but may also contain * characters as wildcards

filtertype is one = ~=, <=, >= which respectively describe equality, approximate equality, inferiority and superiority.

Thus (cn=John Smith) matches all records having a canonical name equal to John Smith.

A special case is the presence filter (<attr>=* which matches records containing a particular attribute. For instance (mail=*) matches all records containing a mail attribute.

Simple filters can be connected together with the logical operators &, | and ! which stand for the usual and, or and not operators.

(&(objectClass=Person)(mail=*)(|(sn=Smith)(givenname=John))) matches records of class Person containing a mail attribute and corresponding to people whose last name is Smith or whose first name is John.

54 Internationalization

54.1 I18N Levels 1 and 2

XEmacs is now compliant with I18N levels 1 and 2. Specifically, this means that it is 8-bit clean and correctly handles time and date functions. XEmacs will correctly display the entire ISO-Latin 1 character set.

The compose key may now be used to create any character in the ISO-Latin 1 character set not directly available via the keyboard.. In order for the compose key to work it is necessary to load the file '`x-compose.el`'. At any time while composing a character, `C-h` will display all valid completions and the character which would be produced.

54.2 I18N Level 3

54.2.1 Level 3 Basics

XEmacs now provides alpha-level functionality for I18N Level 3. This means that everything necessary for full messaging is available, but not every file has been converted.

The two message files which have been created are '`src/emacs.po`' and '`lisp/packages/mh-e.po`'. Both files need to be converted using `msgfmt`, and the resulting '`.mo`' files placed in some locale's `LC_MESSAGES` directory. The test "translations" in these files are the original messages prefixed by `TRNSLT_`.

The domain for a variable is stored on the variable's property list under the property name *variable-domain*. The function `documentation-property` uses this information when translating a variable's documentation.

54.2.2 Level 3 Primitives

gettext *string* — Function

This function looks up *string* in the default message domain and returns its translation. If `I18N3` was not enabled when XEmacs was compiled, it just returns *string*.

dgettext *domain string* — Function

This function looks up *string* in the specified message domain and returns its translation. If `I18N3` was not enabled when XEmacs was compiled, it just returns *string*.

bind-text-domain *domain pathname* — Function

This function associates a pathname with a message domain. Here's how the path to message file is constructed under SunOS 5.x:

```
{pathname}/{LANG}/LC_MESSAGES/{domain}.mo
```

If `I18N3` was not enabled when XEmacs was compiled, this function does nothing.

domain *string* Special Form

This function specifies the text domain used for translating documentation strings and interactive prompts of a function. For example, write:

```
(defun foo (arg) "Doc string" (domain "emacs-foo") ...)
```

to specify `emacs-foo` as the text domain of the function `foo`. The "call" to `domain` is actually a declaration rather than a function; when actually called, `domain` just returns `nil`.

domain-of *function* Function

This function returns the text domain of *function*; it returns `nil` if it is the default domain. If `I18N3` was not enabled when XEmacs was compiled, it always returns `nil`.

54.2.3 Dynamic Messaging

The `format` function has been extended to permit you to change the order of parameter insertion. For example, the conversion format `%1$s` inserts parameter one as a string, while `%2$s` inserts parameter two. This is useful when creating translations which require you to change the word order.

54.2.4 Domain Specification

The default message domain of XEmacs is 'emacs'. For add-on packages, it is best to use a different domain. For example, let us say we want to convert the "gorilla" package to use the domain 'emacs-gorilla'. To translate the message "What gorilla?", use `dgettext` as follows:

```
(dgettext "emacs-gorilla" "What gorilla?")
```

A function (or macro) which has a documentation string or an interactive prompt needs to be associated with the domain in order for the documentation or prompt to be translated. This is done with the `domain` special form as follows:

```
(defun scratch (location)
  "Scratch the specified location."
  (domain "emacs-gorilla")
  (interactive "sScratch: ")
  ... )
```

It is most efficient to specify the domain in the first line of the function body, before the `interactive` form.

For variables and constants which have documentation strings, specify the domain after the documentation.

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

Example:

```
(defvar weight 250 "Weight of gorilla, in pounds." "emacs-gorilla")
```

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

Example:

```
(defconst limbs 4 "Number of limbs" "emacs-gorilla")
```

Autoloaded functions which are specified in '`loaddefs.el`' do not need to have a domain specification, because their documentation strings are extracted into the main message base. However, for autoloaded functions which are specified in a separate package, use following syntax:

autoload *symbol filename* &optional *docstring interactive macro domain* Function

Example:

```
(autoload 'explore "jungle" "Explore the jungle." nil nil "emacs-gorilla")
```

54.2.5 Documentation String Extraction

The utility '`etc/make-po`' scans the file `DOC` to extract documentation strings and creates a message file `doc.po`. This file may then be inserted within `emacs.po`.

Currently, `make-po` is hard-coded to read from `DOC` and write to `doc.po`. In order to extract documentation strings from an add-on package, first run `make-docfile` on the package to produce the `DOC` file. Then run `make-po -p` with the `-p` argument to indicate that we are extracting documentation for an add-on package.

(The `-p` argument is a kludge to make up for a subtle difference between pre-loaded documentation and add-on documentation: For add-on packages, the final carriage returns in the strings produced by `make-docfile` must be ignored.)

54.3 I18N Level 4

The Asian-language support in XEmacs is called "MULE". See Chapter 55 [MULE], page 733.

55 MULE

MULE is the name originally given to the version of GNU Emacs extended for multi-lingual (and in particular Asian-language) support. "MULE" is short for "MUlti-Lingual Emacs". It was originally called Nemacs ("Nihon Emacs" where "Nihon" is the Japanese word for "Japan"), when it only provided support for Japanese. XEmacs refers to its multi-lingual support as *MULE support* since it is based on *MULE*.

55.1 Internationalization Terminology

In internationalization terminology, a string of text is divided up into *characters*, which are the printable units that make up the text. A single character is (for example) a capital '`A`', the number '`2`', a Katakana character, a Kanji ideograph (an *ideograph* is a "picture" character, such as is used in Japanese Kanji, Chinese Hanzi, and Korean Hangul; typically there are thousands of such ideographs in each language), etc. The basic property of a character is its shape. Note that the same character may be drawn by two different people (or in two different fonts) in slightly different ways, although the basic shape will be the same.

In some cases, the differences will be significant enough that it is actually possible to identify two or more distinct shapes that both represent the same character. For example, the lowercase letters '`a`' and '`g`' each have two distinct possible shapes – the '`a`' can optionally have a curved tail projecting off the top, and the '`g`' can be formed either of two loops, or of one loop and a tail hanging off the bottom. Such distinct possible shapes of a character are called *glyphs*. The important characteristic of two glyphs making up the same character is that the choice between one or the other is purely stylistic and has no linguistic effect on a word (this is the reason why a capital '`A`' and lowercase '`a`' are different characters rather than different glyphs – e.g. '`Aspen`' is a city while '`aspen`' is a kind of tree).

Note that *character* and *glyph* are used differently here than elsewhere in XEmacs.

A *character set* is simply a set of related characters. ASCII, for example, is a set of 94 characters (or 128, if you count non-printing characters). Other character sets are ISO8859-1 (ASCII plus various accented characters and other international symbols), JISX0201 (ASCII, more or less, plus half-width Katakana), JISX0208 (Japanese Kanji), JISX0212 (a second set of less-used Japanese Kanji), GB2312 (Mainland Chinese Hanzi), etc.

Every character set has one or more *orderings*, which can be viewed as a way of assigning a number (or set of numbers) to each character in the set. For most character sets, there is a standard ordering, and in fact all of the character sets mentioned above define a particular ordering. ASCII, for example, places letters in their "natural" order, puts uppercase letters before lowercase letters, numbers before letters, etc. Note that for many of the Asian character sets, there is no natural ordering of the characters. The actual orderings are based on one or more salient characteristic, of which there are many to choose from – e.g. number of strokes, common radicals, phonetic ordering, etc.

The set of numbers assigned to any particular character are called the character's *position codes*. The number of position codes required to index a particular character in a character set is called the *dimension* of the character set. ASCII, being a relatively small character set, is of dimension one, and each character in the set is indexed using a single position code, in the range 0 through 127 (if non-printing characters are included) or 33 through 126 (if only the printing characters are considered). JISX0208, i.e. Japanese Kanji, has thousands of characters, and is of dimension two – every character is indexed by two position codes, each in the range 33 through 126. (Note that the choice of the range here is somewhat arbitrary. Although a character set such as JISX0208 defines an *ordering* of all its characters, it does not define the actual mapping between numbers and characters. You could just as easily index the characters in JISX0208 using numbers in the range 0 through 93, 1 through 94, 2 through 95, etc. The reason for the actual range chosen is so that the position codes match up with the actual values used in the common encodings.)

An *encoding* is a way of numerically representing characters from one or more character sets into a stream of like-sized numerical values called *words*; typically these are 8-bit, 16-bit, or 32-bit quantities. If an encoding encompasses only one character set, then the position codes for the characters in that character set could be used directly. (This is the case with ASCII, and as a result, most people do not understand the difference between a character set and an encoding.) This is not possible, however, if more than one character set is to be used in the encoding. For example, printed Japanese text typically requires characters from multiple character sets – ASCII, JISX0208, and JISX0212, to be specific. Each of these is indexed using one or more position codes in the range 33 through 126, so the position codes could not be used directly or there would be no way to tell which character was meant. Different Japanese encodings handle this differently – JIS uses special escape characters to denote different character sets; EUC sets the high bit of the position codes for JISX0208 and JISX0212, and puts a special extra byte before each JISX0212 character; etc. (JIS, EUC, and most of the other encodings you will encounter are 7-bit or 8-bit encodings. There is one common 16-bit encoding, which is Unicode; this strives to represent all the world's characters in a single large character set. 32-bit encodings are generally used internally in programs to simplify the code that manipulates them; however, they are not much used externally because they are not very space-efficient.)

Encodings are classified as either *modal* or *non-modal*. In a *modal encoding*, there are multiple states that the encoding can be in, and the interpretation of the values in the stream depends on the current global state of the encoding. Special values in the encoding, called *escape sequences*, are used to change the global state. JIS, for example, is a modal encoding. The bytes '`ESC $ B`' indicate that, from then on, bytes are to be interpreted as position codes for JISX0208, rather than as ASCII. This effect is cancelled using the bytes '`ESC ( B`', which mean "switch from whatever the current state is to ASCII". To switch to JISX0212, the escape sequence '`ESC $ ( D`'. (Note that here, as is common, the escape sequences do in fact begin with '`ESC`'. This is not necessarily the case, however.)

A *non-modal encoding* has no global state that extends past the character currently being interpreted. EUC, for example, is a non-modal encoding. Characters in JISX0208 are encoded by setting the high bit of the position codes, and characters in JISX0212 are encoded by doing the same but also prefixing the character with the byte 0x8F.

The advantage of a modal encoding is that it is generally more space-efficient, and is easily extendable because there are essentially an arbitrary number of escape sequences that can be created. The disadvantage, however, is that it is much more difficult to work with if it is not being processed in a sequential manner. In the non-modal EUC encoding, for example, the byte 0x41 always refers to the letter '`A`'; whereas in JIS, it could either be the letter '`A`', or one of the two position codes in a JISX0208 character, or one of the two position codes in a JISX0212 character. Determining exactly which one is meant could be difficult and time-consuming if the previous bytes in the string have not already been processed.

Non-modal encodings are further divided into *fixed-width* and *variable-width* formats. A fixed-width encoding always uses the same number of words per character, whereas a variable-width encoding does not. EUC is a good example of a variable-width encoding: one to three bytes are used per character, depending on the character set. 16-bit and 32-bit encodings are nearly always fixed-width, and this is in fact one of the main reasons for using an encoding with a larger word size. The advantages of fixed-width encodings should be obvious. The advantages of variable-width encodings are that they are generally more space-efficient and allow for compatibility with existing 8-bit encodings such as ASCII.

Note that the bytes in an 8-bit encoding are often referred to as *octets* rather than simply as bytes. This terminology dates back to the days before 8-bit bytes were universal, when some computers had 9-bit bytes, others had 10-bit bytes, etc.

55.2 Charsets

A *charset* in MULE is an object that encapsulates a particular character set as well as an ordering of those characters. Charsets are permanent objects and are named using symbols, like faces.

charsetp *object* Function

This function returns non-`nil` if *object* is a charset.

55.2.1 Charset Properties

Charsets have the following properties:

`name` A symbol naming the charset. Every charset must have a different name; this allows a charset to be referred to using its name rather than the actual charset object.

`doc-string`
A documentation string describing the charset.

`registry` A regular expression matching the font registry field for this character set. For example, both the `ascii` and `latin-iso8859-1` charsets use the registry `"ISO8859-1"`. This field is used to choose an appropriate font when the user gives a general font specification such as '`-*-courier-medium-r-*-140-*`', i.e. a 14-point upright medium-weight Courier font.

`dimension`
Number of position codes used to index a character in the character set. XEmacs/MULE can only handle character sets of dimension 1 or 2. This property defaults to 1.

`chars` Number of characters in each dimension. In XEmacs/MULE, the only allowed values are 94 or 96. (There are a couple of pre-defined character sets, such as ASCII, that do not follow this, but you cannot define new ones like this.) Defaults to 94. Note that if the dimension is 2, the character set thus described is 94x94 or 96x96.

`columns` Number of columns used to display a character in this charset. Only used in TTY mode. (Under X, the actual width of a character can be derived from the font used to display the characters.) If unspecified, defaults to the dimension. (This is almost always the correct value, because character sets with dimension 2 are usually ideograph character sets, which need two columns to display the intricate ideographs.)

`direction`
A symbol, either `l2r` (left-to-right) or `r2l` (right-to-left). Defaults to `l2r`. This specifies the direction that the text should be displayed in, and will be left-to-right for most charsets but right-to-left for Hebrew and Arabic. (Right-to-left display is not currently implemented.)

`final` Final byte of the standard ISO 2022 escape sequence designating this charset. Must be supplied. Each combination of (*dimension*, *chars*) defines a separate namespace for final bytes, and each charset within a particular namespace must have a different final byte. Note that ISO 2022 restricts the final byte to the range 0x30 - 0x7E if dimension == 1, and 0x30 - 0x5F if dimension == 2. Note also that final bytes in the range 0x30 - 0x3F are reserved for user-defined (not official) character sets. For more information on ISO 2022, see Section 55.6 [Coding Systems], page 743.

`graphic` 0 (use left half of font on output) or 1 (use right half of font on output). Defaults to 0. This specifies how to convert the position codes that index a character in a character set into an index into the font used to display the character set. With `graphic` set to 0, position codes 33 through 126 map to font indices 33 through 126; with it set to 1, position codes 33 through 126 map to font indices 161 through 254 (i.e. the same number but with the high bit set). For example, for a font whose registry is ISO8859-1, the left half of the font (octets 0x20 - 0x7F) is the `ascii` charset, while the right half (octets 0xA0 - 0xFF) is the `latin-iso8859-1` charset.

`ccl-program`
A compiled CCL program used to convert a character in this charset into an index into the font. This is in addition to the `graphic` property. If a CCL program is defined, the position codes of a character will first be processed according to `graphic` and then passed through the CCL program, with the resulting values used to index the font.

This is used, for example, in the Big5 character set (used in Taiwan). This character set is not ISO-2022-compliant, and its size (94x157) does not fit within the maximum 96x96 size of ISO-2022-compliant character sets. As a result, XEmacs/MULE splits it (in a rather complex fashion, so as to group the most commonly used characters together) into two charset objects (`big5-1` and `big5-2`), each of size 94x94, and each charset object uses a CCL program to convert the modified position codes back into standard Big5 indices to retrieve a character from a Big5 font.

Most of the above properties can only be changed when the charset is created. See Section 55.2.3 [Charset Property Functions], page 737.

55.2.2 Basic Charset Functions

find-charset *charset-or-name* Function

This function retrieves the charset of the given name. If *charset-or-name* is a charset object, it is simply returned. Otherwise, *charset-or-name* should be a symbol. If there is no such charset, `nil` is returned. Otherwise the associated charset object is returned.

get-charset *name* Function

This function retrieves the charset of the given name. Same as `find-charset` except an error is signalled if there is no such charset instead of returning `nil`.

charset-list Function

This function returns a list of the names of all defined charsets.

make-charset *name doc-string props* Function

This function defines a new character set. This function is for use with Mule support. *name* is a symbol, the name by which the character set is normally referred. *doc-string* is a string describing the character set. *props* is a property list, describing the specific nature of the character set. The recognized properties are `registry`, `dimension`, `columns`, `chars`, `final`, `graphic`, `direction`, and `ccl-program`, as previously described.

make-reverse-direction-charset *charset new-name* Function

This function makes a charset equivalent to *charset* but which goes in the opposite direction. *new-name* is the name of the new charset. The new charset is returned.

charset-from-attributes *dimension chars final* &optional *direction* Function

This function returns a charset with the given *dimension*, *chars*, *final*, and *direction*. If *direction* is omitted, both directions will be checked (left-to-right will be returned if character sets exist for both directions).

charset-reverse-direction-charset *charset* Function

This function returns the charset (if any) with the same dimension, number of characters, and final byte as *charset*, but which is displayed in the opposite direction.

55.2.3 Charset Property Functions

All of these functions accept either a charset name or charset object.

charset-property *charset prop* Function

This function returns property *prop* of *charset*. See Section 55.2.1 [Charset Properties], page 735.

Convenience functions are also provided for retrieving individual properties of a charset.

charset-name *charset* Function

This function returns the name of *charset*. This will be a symbol.

charset-doc-string *charset* Function

This function returns the doc string of *charset*.

charset-registry *charset* Function

This function returns the registry of *charset*.

charset-dimension *charset* Function

This function returns the dimension of *charset*.

charset-chars *charset* Function

This function returns the number of characters per dimension of *charset*.

charset-columns *charset* Function

This function returns the number of display columns per character (in TTY mode) of *charset*.

charset-direction *charset* Function

This function returns the display direction of *charset* – either `l2r` or `r2l`.

charset-final *charset* Function

This function returns the final byte of the ISO 2022 escape sequence designating *charset*.

charset-graphic *charset* Function

This function returns either 0 or 1, depending on whether the position codes of characters in *charset* map to the left or right half of their font, respectively.

charset-ccl-program *charset* Function

This function returns the CCL program, if any, for converting position codes of characters in *charset* into font indices.

The only property of a charset that can currently be set after the charset has been created is the CCL program.

set-charset-ccl-program *charset ccl-program* Function

This function sets the `ccl-program` property of *charset* to *ccl-program*.

55.2.4 Predefined Charsets

The following charsets are predefined in the C code.

```
Name                    Type  Fi Gr Dir Registry
--------------------------------------------------------------
ascii                    94    B  0  l2r ISO8859-1
control-1                94       0  l2r ---
latin-iso8859-1          94    A  1  l2r ISO8859-1
latin-iso8859-2          96    B  1  l2r ISO8859-2
latin-iso8859-3          96    C  1  l2r ISO8859-3
latin-iso8859-4          96    D  1  l2r ISO8859-4
cyrillic-iso8859-5       96    L  1  l2r ISO8859-5
arabic-iso8859-6         96    G  1  r2l ISO8859-6
greek-iso8859-7          96    F  1  l2r ISO8859-7
hebrew-iso8859-8         96    H  1  r2l ISO8859-8
latin-iso8859-9          96    M  1  l2r ISO8859-9
thai-tis620              96    T  1  l2r TIS620
katakana-jisx0201        94    I  1  l2r JISX0201.1976
latin-jisx0201           94    J  0  l2r JISX0201.1976
japanese-jisx0208-1978   94x94 @  0  l2r JISX0208.1978
japanese-jisx0208        94x94 B  0  l2r JISX0208.19(83|90)
japanese-jisx0212        94x94 D  0  l2r JISX0212
chinese-gb2312           94x94 A  0  l2r GB2312
chinese-cns11643-1       94x94 G  0  l2r CNS11643.1
chinese-cns11643-2       94x94 H  0  l2r CNS11643.2
chinese-big5-1           94x94 0  0  l2r Big5
chinese-big5-2           94x94 1  0  l2r Big5
korean-ksc5601           94x94 C  0  l2r KSC5601
composite                96x96    0  l2r ---
```

The following charsets are predefined in the Lisp code.

```
Name                     Type  Fi Gr Dir Registry
--------------------------------------------------------------
arabic-digit             94    2  0  l2r MuleArabic-0
arabic-1-column          94    3  0  r2l MuleArabic-1
arabic-2-column          94    4  0  r2l MuleArabic-2
sisheng                  94    0  0  l2r sisheng_cwnn\|OMRON_UDC_ZH
chinese-cns11643-3       94x94 I  0  l2r CNS11643.1
chinese-cns11643-4       94x94 J  0  l2r CNS11643.1
chinese-cns11643-5       94x94 K  0  l2r CNS11643.1
chinese-cns11643-6       94x94 L  0  l2r CNS11643.1
chinese-cns11643-7       94x94 M  0  l2r CNS11643.1
ethiopic                 94x94 2  0  l2r Ethio
ascii-r2l                94    B  0  r2l ISO8859-1
ipa                      96    0  1  l2r MuleIPA
vietnamese-lower         96    1  1  l2r VISCII1.1
vietnamese-upper         96    2  1  l2r VISCII1.1
```

For all of the above charsets, the dimension and number of columns are the same.

Note that ASCII, Control-1, and Composite are handled specially. This is why some of the fields are blank; and some of the filled-in fields (e.g. the type) are not really accurate.

55.3 MULE Characters

make-char *charset arg1* &optional *arg2* — Function

This function makes a multi-byte character from *charset* and octets *arg1* and *arg2*.

char-charset *ch* — Function

This function returns the character set of char *ch*.

char-octet *ch* &optional *n* — Function

This function returns the octet (i.e. position code) numbered *n* (should be 0 or 1) of char *ch*. *n* defaults to 0 if omitted.

find-charset-region *start end* &optional *buffer* — Function

This function returns a list of the charsets in the region between *start* and *end*. *buffer* defaults to the current buffer if omitted.

find-charset-string *string* — Function

This function returns a list of the charsets in *string*.

55.4 Composite Characters

Composite characters are not yet completely implemented.

make-composite-char *string* — Function

This function converts a string into a single composite character. The character is the result of overstriking all the characters in the string.

composite-char-string *ch* — Function

This function returns a string of the characters comprising a composite character.

compose-region *start end* &optional *buffer* — Function

This function composes the characters in the region from *start* to *end* in *buffer* into one composite character. The composite character replaces the composed characters. *buffer* defaults to the current buffer if omitted.

decompose-region *start end* &optional *buffer* — Function

This function decomposes any composite characters in the region from *start* to *end* in *buffer*. This converts each composite character into one or more characters, the individual characters out of which the composite character was formed. Non-composite characters are left as-is. *buffer* defaults to the current buffer if omitted.

55.5 ISO 2022

This section briefly describes the ISO 2022 encoding standard. For more thorough understanding, please refer to the original document of ISO 2022.

Character sets (*charsets*) are classified into the following four categories, according to the number of characters of charset: 94-charset, 96-charset, 94x94-charset, and 96x96-charset.

94-charset ASCII(B), left(J) and right(I) half of JISX0201, ...

96-charset Latin-1(A), Latin-2(B), Latin-3(C), ...

94x94-charset
GB2312(A), JISX0208(B), KSC5601(C), ...

96x96-charset
none for the moment

The character in parentheses after the name of each charset is the *final character* F, which can be regarded as the identifier of the charset. ECMA allocates F to each charset. F is in the range of 0x30..0x7F, but 0x30..0x3F are only for private use.

Note: *ECMA* = European Computer Manufacturers Association

There are four *registers of charsets*, called G0 thru G3. You can designate (or assign) any charset to one of these registers.

The code space contained within one octet (of size 256) is divided into 4 areas: C0, GL, C1, and GR. GL and GR are the areas into which a register of charset can be invoked into.

```
C0: 0x00 - 0x1F
GL: 0x20 - 0x7F
C1: 0x80 - 0x9F
GR: 0xA0 - 0xFF
```

Usually, in the initial state, G0 is invoked into GL, and G1 is invoked into GR.

ISO 2022 distinguishes 7-bit environments and 8-bit environments. In 7-bit environments, only C0 and GL are used.

Charset designation is done by escape sequences of the form:

```
ESC [I] I F
```

where I is an intermediate character in the range 0x20 - 0x2F, and F is the final character identifying this charset.

The meaning of intermediate characters are:

```
$ [0x24]: indicate charset of dimension 2 (94x94 or 96x96).
( [0x28]: designate to G0 a 94-charset whose final byte is F.
) [0x29]: designate to G1 a 94-charset whose final byte is F.
* [0x2A]: designate to G2 a 94-charset whose final byte is F.
+ [0x2B]: designate to G3 a 94-charset whose final byte is F.
- [0x2D]: designate to G1 a 96-charset whose final byte is F.
. [0x2E]: designate to G2 a 96-charset whose final byte is F.
/ [0x2F]: designate to G3 a 96-charset whose final byte is F.
```

The following rule is not allowed in ISO 2022 but can be used in Mule.

```
, [0x2C]: designate to G0 a 96-charset whose final byte is F.
```

Here are examples of designations:

```
ESC ( B :             designate to G0 ASCII
ESC - A :             designate to G1 Latin-1
ESC $ ( A or ESC $ A : designate to G0 GB2312
ESC $ ( B or ESC $ B : designate to G0 JISX0208
ESC $ ) C :           designate to G1 KSC5601
```

To use a charset designated to G2 or G3, and to use a charset designated to G1 in a 7-bit environment, you must explicitly invoke G1, G2, or G3 into GL. There are two types of invocation, Locking Shift (forever) and Single Shift (one character only).

Locking Shift is done as follows:

```
LS0 or SI (0x0F): invoke G0 into GL
LS1 or SO (0x0E): invoke G1 into GL
LS2:  invoke G2 into GL
LS3:  invoke G3 into GL
LS1R: invoke G1 into GR
LS2R: invoke G2 into GR
LS3R: invoke G3 into GR
```

Single Shift is done as follows:

```
SS2 or ESC N: invoke G2 into GL
SS3 or ESC O: invoke G3 into GL
```

(#### Ben says: I think the above is slightly incorrect. It appears that SS2 invokes G2 into GR and SS3 invokes G3 into GR, whereas ESC N and ESC O behave as indicated. The above definitions will not parse EUC-encoded text correctly, and it looks like the code in mule-coding.c has similar problems.)

You may realize that there are a lot of ISO-2022-compliant ways of encoding multilingual text. Now, in the world, there exist many coding systems such as X11's Compound Text, Japanese JUNET code, and so-called EUC (Extended UNIX Code); all of these are variants of ISO 2022.

In Mule, we characterize ISO 2022 by the following attributes:

1. Initial designation to G0 thru G3.
2. Allow designation of short form for Japanese and Chinese.
3. Should we designate ASCII to G0 before control characters?
4. Should we designate ASCII to G0 at the end of line?
5. 7-bit environment or 8-bit environment.
6. Use Locking Shift or not.
7. Use ASCII or JIS0201-1976-Roman.
8. Use JISX0208-1983 or JISX0208-1976.

(The last two are only for Japanese.)

By specifying these attributes, you can create any variant of ISO 2022.

Here are several examples:

```
junet -- Coding system used in JUNET.
1. G0 <- ASCII, G1..3 <- never used
2. Yes.
3. Yes.
4. Yes.
5. 7-bit environment
6. No.
7. Use ASCII
8. Use JISX0208-1983

ctext -- Compound Text
1. G0 <- ASCII, G1 <- Latin-1, G2,3 <- never used
2. No.
3. No.
4. Yes.
5. 8-bit environment
6. No.
7. Use ASCII
8. Use JISX0208-1983

euc-china -- Chinese EUC.  Although many people call this
as "GB encoding", the name may cause misunderstanding.
1. G0 <- ASCII, G1 <- GB2312, G2,3 <- never used
2. No.
3. Yes.
4. Yes.
5. 8-bit environment
6. No.
7. Use ASCII
8. Use JISX0208-1983

korean-mail -- Coding system used in Korean network.
1. G0 <- ASCII, G1 <- KSC5601, G2,3 <- never used
2. No.
3. Yes.
4. Yes.
5. 7-bit environment
6. Yes.
7. No.
8. No.
```

Mule creates all these coding systems by default.

55.6 Coding Systems

A coding system is an object that defines how text containing multiple character sets is encoded into a stream of (typically 8-bit) bytes. The coding system is used to decode the stream into a series of characters (which may be from multiple charsets) when the text is read from a file or process, and is used to encode the text back into the same format when it is written out to a file or process.

For example, many ISO-2022-compliant coding systems (such as Compound Text, which is used for inter-client data under the X Window System) use escape sequences to switch between different charsets – Japanese Kanji, for example, is invoked with '`ESC $ ( B`'; ASCII is invoked with '`ESC ( B`'; and Cyrillic is invoked with '`ESC - L`'. See `make-coding-system` for more information.

Coding systems are normally identified using a symbol, and the symbol is accepted in place of the actual coding system object whenever a coding system is called for. (This is similar to how faces and charsets work.)

coding-system-p *object* Function

This function returns non-`nil` if *object* is a coding system.

55.6.1 Coding System Types

`nil`
`autodetect`
: Automatic conversion. XEmacs attempts to detect the coding system used in the file.

`no-conversion`
: No conversion. Use this for binary files and such. On output, graphic characters that are not in ASCII or Latin-1 will be replaced by a '`?`'. (For a no-conversion-encoded buffer, these characters will only be present if you explicitly insert them.)

`shift-jis`
: Shift-JIS (a Japanese encoding commonly used in PC operating systems).

`iso2022`
: Any ISO-2022-compliant encoding. Among other things, this includes JIS (the Japanese encoding commonly used for e-mail), national variants of EUC (the standard Unix encoding for Japanese and other languages), and Compound Text (an encoding used in X11). You can specify more specific information about the conversion with the *flags* argument.

`big5`
: Big5 (the encoding commonly used for Taiwanese).

`ccl`
: The conversion is performed using a user-written pseudo-code program. CCL (Code Conversion Language) is the name of this pseudo-code.

`internal`
: Write out or read in the raw contents of the memory representing the buffer's text. This is primarily useful for debugging purposes, and is only enabled when XEmacs has been compiled with `DEBUG_XEMACS` set (the '`--debug`' configure option). **Warning**: Reading in a file using `internal` conversion can result in an internal inconsistency in the memory representing a buffer's text, which will produce unpredictable results and may cause XEmacs to crash. Under normal circumstances you should never use `internal` conversion.

55.6.2 EOL Conversion

`nil` Automatically detect the end-of-line type (LF, CRLF, or CR). Also generate subsidiary coding systems named *name*`-unix`, *name*`-dos`, and *name*`-mac`, that are identical to this coding system but have an EOL-TYPE value of `lf`, `crlf`, and `cr`, respectively.

`lf` The end of a line is marked externally using ASCII LF. Since this is also the way that XEmacs represents an end-of-line internally, specifying this option results in no end-of-line conversion. This is the standard format for Unix text files.

`crlf` The end of a line is marked externally using ASCII CRLF. This is the standard format for MS-DOS text files.

`cr` The end of a line is marked externally using ASCII CR. This is the standard format for Macintosh text files.

`t` Automatically detect the end-of-line type but do not generate subsidiary coding systems. (This value is converted to `nil` when stored internally, and `coding-system-property` will return `nil`.)

55.6.3 Coding System Properties

`mnemonic` String to be displayed in the modeline when this coding system is active.

`eol-type` End-of-line conversion to be used. It should be one of the types listed in Section 55.6.2 [EOL Conversion], page 744.

`post-read-conversion`
Function called after a file has been read in, to perform the decoding. Called with two arguments, *beg* and *end*, denoting a region of the current buffer to be decoded.

`pre-write-conversion`
Function called before a file is written out, to perform the encoding. Called with two arguments, *beg* and *end*, denoting a region of the current buffer to be encoded.

The following additional properties are recognized if *type* is `iso2022`:

`charset-g0`
`charset-g1`
`charset-g2`
`charset-g3`
The character set initially designated to the G0 - G3 registers. The value should be one of

- A charset object (designate that character set)
- `nil` (do not ever use this register)
- `t` (no character set is initially designated to the register, but may be later on; this automatically sets the corresponding `force-g*-on-output` property)

`force-g0-on-output`
`force-g1-on-output`
`force-g2-on-output`
`force-g3-on-output`
: If non-`nil`, send an explicit designation sequence on output before using the specified register.

`short`
: If non-`nil`, use the short forms '`ESC $ @`', '`ESC $ A`', and '`ESC $ B`' on output in place of the full designation sequences '`ESC $ ( @`', '`ESC $ ( A`', and '`ESC $ ( B`'.

`no-ascii-eol`
: If non-`nil`, don't designate ASCII to G0 at each end of line on output. Setting this to non-`nil` also suppresses other state-resetting that normally happens at the end of a line.

`no-ascii-cntl`
: If non-`nil`, don't designate ASCII to G0 before control chars on output.

`seven`
: If non-`nil`, use 7-bit environment on output. Otherwise, use 8-bit environment.

`lock-shift`
: If non-`nil`, use locking-shift (SO/SI) instead of single-shift or designation by escape sequence.

`no-iso6429`
: If non-`nil`, don't use ISO6429's direction specification.

`escape-quoted`
: If non-nil, literal control characters that are the same as the beginning of a recognized ISO 2022 or ISO 6429 escape sequence (in particular, ESC (0x1B), SO (0x0E), SI (0x0F), SS2 (0x8E), SS3 (0x8F), and CSI (0x9B)) are "quoted" with an escape character so that they can be properly distinguished from an escape sequence. (Note that doing this results in a non-portable encoding.) This encoding flag is used for byte-compiled files. Note that ESC is a good choice for a quoting character because there are no escape sequences whose second byte is a character from the Control-0 or Control-1 character sets; this is explicitly disallowed by the ISO 2022 standard.

`input-charset-conversion`
: A list of conversion specifications, specifying conversion of characters in one charset to another when decoding is performed. Each specification is a list of two elements: the source charset, and the destination charset.

`output-charset-conversion`
: A list of conversion specifications, specifying conversion of characters in one charset to another when encoding is performed. The form of each specification is the same as for `input-charset-conversion`.

The following additional properties are recognized (and required) if *type* is `ccl`:

`decode`
: CCL program used for decoding (converting to internal format).

`encode`
: CCL program used for encoding (converting to external format).

55.6.4 Basic Coding System Functions

find-coding-system *coding-system-or-name* Function

This function retrieves the coding system of the given name.

If *coding-system-or-name* is a coding-system object, it is simply returned. Otherwise, *coding-system-or-name* should be a symbol. If there is no such coding system, `nil` is returned. Otherwise the associated coding system object is returned.

get-coding-system *name* Function

This function retrieves the coding system of the given name. Same as `find-coding-system` except an error is signalled if there is no such coding system instead of returning `nil`.

coding-system-list Function

This function returns a list of the names of all defined coding systems.

coding-system-name *coding-system* Function

This function returns the name of the given coding system.

make-coding-system *name type* &optional *doc-string props* Function

This function registers symbol *name* as a coding system.

type describes the conversion method used and should be one of the types listed in Section 55.6.1 [Coding System Types], page 744.

doc-string is a string describing the coding system.

props is a property list, describing the specific nature of the character set. Recognized properties are as in Section 55.6.3 [Coding System Properties], page 745.

copy-coding-system *old-coding-system new-name* Function

This function copies *old-coding-system* to *new-name*. If *new-name* does not name an existing coding system, a new one will be created.

subsidiary-coding-system *coding-system eol-type* Function

This function returns the subsidiary coding system of *coding-system* with eol type *eol-type*.

55.6.5 Coding System Property Functions

coding-system-doc-string *coding-system* Function

This function returns the doc string for *coding-system*.

coding-system-type *coding-system* Function

This function returns the type of *coding-system*.

coding-system-property *coding-system prop* Function

This function returns the *prop* property of *coding-system*.

55.6.6 Encoding and Decoding Text

decode-coding-region *start end coding-system* &optional *buffer* Function

This function decodes the text between *start* and *end* which is encoded in *coding-system*. This is useful if you've read in encoded text from a file without decoding it (e.g. you read in a JIS-formatted file but used the `binary` or `no-conversion` coding system, so that it shows up as '`^[$B!<!+^[(B`'). The length of the encoded text is returned. *buffer* defaults to the current buffer if unspecified.

encode-coding-region *start end coding-system* &optional *buffer* Function

This function encodes the text between *start* and *end* using *coding-system*. This will, for example, convert Japanese characters into stuff such as '`^[$B!<!+^[(B`' if you use the JIS encoding. The length of the encoded text is returned. *buffer* defaults to the current buffer if unspecified.

55.6.7 Detection of Textual Encoding

coding-category-list Function

This function returns a list of all recognized coding categories.

set-coding-priority-list *list* Function

This function changes the priority order of the coding categories. *list* should be a list of coding categories, in descending order of priority. Unspecified coding categories will be lower in priority than all specified ones, in the same relative order they were in previously.

coding-priority-list Function

This function returns a list of coding categories in descending order of priority.

set-coding-category-system *coding-category coding-system* Function

This function changes the coding system associated with a coding category.

coding-category-system *coding-category* Function

This function returns the coding system associated with a coding category.

detect-coding-region *start end* &optional *buffer* Function

This function detects coding system of the text in the region between *start* and *end*. Returned value is a list of possible coding systems ordered by priority. If only ASCII characters are found, it returns `autodetect` or one of its subsidiary coding systems according to a detected end-of-line type. Optional arg *buffer* defaults to the current buffer.

55.6.8 Big5 and Shift-JIS Functions

These are special functions for working with the non-standard Shift-JIS and Big5 encodings.

decode-shift-jis-char *code* Function

This function decodes a JISX0208 character of Shift-JIS coding-system. *code* is the character code in Shift-JIS as a cons of type bytes. The corresponding character is returned.

encode-shift-jis-char *ch* Function

This function encodes a JISX0208 character *ch* to SHIFT-JIS coding-system. The corresponding character code in SHIFT-JIS is returned as a cons of two bytes.

decode-big5-char *code* Function

This function decodes a Big5 character *code* of BIG5 coding-system. *code* is the character code in BIG5. The corresponding character is returned.

encode-big5-char *ch* Function

This function encodes the Big5 character *char* to BIG5 coding-system. The corresponding character code in Big5 is returned.

55.7 CCL

CCL (Code Conversion Language) is a simple structured programming language designed for character coding conversions. A CCL program is compiled to CCL code (represented by a vector of integers) and executed by the CCL interpreter embedded in Emacs. The CCL interpreter implements a virtual machine with 8 registers called `r0`, ..., `r7`, a number of control structures, and some I/O operators. Take care when using registers `r0` (used in implicit *set* statements) and especially `r7` (used internally by several statements and operations, especially for multiple return values and I/O operations).

CCL is used for code conversion during process I/O and file I/O for non-ISO2022 coding systems. (It is the only way for a user to specify a code conversion function.) It is also used for calculating the code point of an X11 font from a character code. However, since CCL is designed as a powerful programming language, it can be used for more generic calculation where efficiency is demanded. A combination of three or more arithmetic operations can be calculated faster by CCL than by Emacs Lisp.

Warning: The code in '`src/mule-ccl.c`' and '`$packages/lisp/mule-base/mule-ccl.el`' is the definitive description of CCL's semantics. The previous version of this section contained several typos and obsolete names left from earlier versions of MULE, and many may remain. (I am not an experienced CCL programmer; the few who know CCL well find writing English painful.)

A CCL program transforms an input data stream into an output data stream. The input stream, held in a buffer of constant bytes, is left unchanged. The buffer may be filled by an external input operation, taken from an Emacs buffer, or taken from a Lisp string. The output buffer is a dynamic array of bytes, which can be written by an external output operation, inserted into an Emacs buffer, or returned as a Lisp string.

A CCL program is a (Lisp) list containing two or three members. The first member is the *buffer magnification*, which indicates the required minimum size of the output buffer as a multiple of the input buffer. It is followed by the *main block* which executes while there is input remaining, and an optional *EOF block* which is executed when the input is exhausted. Both the main block and the EOF block are CCL blocks.

A *CCL block* is either a CCL statement or list of CCL statements. A *CCL statement* is either a *set statement* (either an integer or an *assignment*, which is a list of a register to receive the assignment, an assignment operator, and an expression) or a *control statement* (a list starting with a keyword, whose allowable syntax depends on the keyword).

55.7.1 CCL Syntax

The full syntax of a CCL program in BNF notation:

```
CCL_PROGRAM :=
        (BUFFER_MAGNIFICATION
         CCL_MAIN_BLOCK
         [ CCL_EOF_BLOCK ])

BUFFER_MAGNIFICATION := integer
CCL_MAIN_BLOCK := CCL_BLOCK
CCL_EOF_BLOCK := CCL_BLOCK

CCL_BLOCK :=
        STATEMENT | (STATEMENT [STATEMENT ...])
STATEMENT :=
        SET | IF | BRANCH | LOOP | REPEAT | BREAK | READ | WRITE
        | CALL | END

SET :=
        (REG = EXPRESSION)
        | (REG ASSIGNMENT_OPERATOR EXPRESSION)
        | integer

EXPRESSION := ARG | (EXPRESSION OPERATOR ARG)

IF := (if EXPRESSION CCL_BLOCK [CCL_BLOCK])
BRANCH := (branch EXPRESSION CCL_BLOCK [CCL_BLOCK ...])
LOOP := (loop STATEMENT [STATEMENT ...])
BREAK := (break)
REPEAT :=
        (repeat)
        | (write-repeat [REG | integer | string])
        | (write-read-repeat REG [integer | ARRAY])
READ :=
        (read REG ...)
        | (read-if (REG OPERATOR ARG) CCL_BLOCK CCL_BLOCK)
        | (read-branch REG CCL_BLOCK [CCL_BLOCK ...])
WRITE :=
        (write REG ...)
        | (write EXPRESSION)
        | (write integer) | (write string) | (write REG ARRAY)
```

```
     | string
CALL := (call ccl-program-name)
END := (end)

REG := r0 | r1 | r2 | r3 | r4 | r5 | r6 | r7
ARG := REG | integer
OPERATOR :=
     + | - | * | / | % | & | '|' | ^ | << | >> | <8 | >8 | //
     | < | > | == | <= | >= | != | de-sjis | en-sjis
ASSIGNMENT_OPERATOR :=
     += | -= | *= | /= | %= | &= | '|=' | ^= | <<= | >>=
ARRAY := '[' integer ... ']'
```

55.7.2 CCL Statements

The Emacs Code Conversion Language provides the following statement types: *set*, *if*, *branch*, *loop*, *repeat*, *break*, *read*, *write*, *call*, and *end*.

Set statement:

The *set* statement has three variants with the syntaxes '`(`*reg* `=` *expression*`)`', '`(`*reg assignment_operator expression*`)`', and '*integer*'. The assignment operator variation of the *set* statement works the same way as the corresponding C expression statement does. The assignment operators are `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, and `>>=`, and they have the same meanings as in C. A "naked integer" *integer* is equivalent to a *set* statement of the form `(r0 =` *integer*`)`.

I/O statements:

The *read* statement takes one or more registers as arguments. It reads one byte (a C char) from the input into each register in turn.

The *write* takes several forms. In the form '`(write` *reg* `...)`' it takes one or more registers as arguments and writes each in turn to the output. The integer in a register (interpreted as an Emchar) is encoded to multibyte form (ie, Bufbytes) and written to the current output buffer. If it is less than 256, it is written as is. The forms '`(write` *expression*`)`' and '`(write` *integer*`)`' are treated analogously. The form '`(write` *string*`)`' writes the constant string to the output. A "naked string" '*string*' is equivalent to the statement '`(write` *string*`)`'. The form '`(write` *reg array*`)`' writes the *reg*th element of the *array* to the output.

Conditional statements:

The *if* statement takes an *expression*, a *CCL block*, and an optional *second CCL block* as arguments. If the *expression* evaluates to non-zero, the first *CCL block* is executed. Otherwise, if there is a *second CCL block*, it is executed.

The *read-if* variant of the *if* statement takes an *expression*, a *CCL block*, and an optional *second CCL block* as arguments. The *expression* must have the form `(`*reg operator* *operand*) (where *operand* is a register or an integer). The `read-if` statement first reads from the input into the first register operand in the *expression*, then conditionally executes a CCL block just as the `if` statement does.

The *branch* statement takes an *expression* and one or more CCL blocks as arguments. The CCL blocks are treated as a zero-indexed array, and the `branch` statement uses the *expression* as the index of the CCL block to execute. Null CCL blocks may be used as no-ops, continuing execution with the statement following the `branch` statement in the containing CCL block. Out-of-range values for the *EXPRESSION* are also treated as no-ops.

The *read-branch* variant of the *branch* statement takes an *register*, a *CCL block*, and an optional *second CCL block* as arguments. The `read-branch` statement first reads from the input into the *register*, then conditionally executes a CCL block just as the `branch` statement does.

Loop control statements:

The *loop* statement creates a block with an implied jump from the end of the block back to its head. The loop is exited on a `break` statement, and continued without executing the tail by a `repeat` statement.

The *break* statement, written '`(break)`', terminates the current loop and continues with the next statement in the current block.

The *repeat* statement has three variants, `repeat`, `write-repeat`, and `write-read-repeat`. Each continues the current loop from its head, possibly after performing I/O. `repeat` takes no arguments and does no I/O before jumping. `write-repeat` takes a single argument (a register, an integer, or a string), writes it to the output, then jumps. `write-read-repeat` takes one or two arguments. The first must be a register. The second may be an integer or an array; if absent, it is implicitly set to the first (register) argument. `write-read-repeat` writes its second argument to the output, then reads from the input into the register, and finally jumps. See the `write` and `read` statements for the semantics of the I/O operations for each type of argument.

Other control statements:

The *call* statement, written '`(call` *ccl-program-name*`)`', executes a CCL program as a subroutine. It does not return a value to the caller, but can modify the register status.

The *end* statement, written '`(end)`', terminates the CCL program successfully, and returns to caller (which may be a CCL program). It does not alter the status of the registers.

55.7.3 CCL Expressions

CCL, unlike Lisp, uses infix expressions. The simplest CCL expressions consist of a single *operand*, either a register (one of `r0`, ..., `r0`) or an integer. Complex expressions are lists of the form `(` *expression operator operand* `)`. Unlike C, assignments are not expressions.

In the following table, *X* is the target resister for a *set*. In subexpressions, this is implicitly `r7`. This means that `>8`, `//`, `de-sjis`, and `en-sjis` cannot be used freely in subexpressions, since they return parts of their values in `r7`. *Y* may be an expression, register, or integer, while *Z* must be a register or an integer.

Name	Operator	Code	C-like Description
CCL_PLUS	`+`	0x00	X = Y + Z
CCL_MINUS	`-`	0x01	X = Y - Z
CCL_MUL	`*`	0x02	X = Y * Z
CCL_DIV	`/`	0x03	X = Y / Z
CCL_MOD	`%`	0x04	X = Y % Z
CCL_AND	`&`	0x05	X = Y & Z
CCL_OR	`|`	0x06	X = Y | Z
CCL_XOR	`^`	0x07	X = Y ^ Z
CCL_LSH	`<<`	0x08	X = Y << Z
CCL_RSH	`>>`	0x09	X = Y >> Z
CCL_LSH8	`<8`	0x0A	X = (Y << 8) | Z
CCL_RSH8	`>8`	0x0B	X = Y >> 8, r[7] = Y & 0xFF
CCL_DIVMOD	`//`	0x0C	X = Y / Z, r[7] = Y % Z
CCL_LS	`<`	0x10	X = (X < Y)
CCL_GT	`>`	0x11	X = (X > Y)
CCL_EQ	`==`	0x12	X = (X == Y)
CCL_LE	`<=`	0x13	X = (X <= Y)
CCL_GE	`>=`	0x14	X = (X >= Y)
CCL_NE	`!=`	0x15	X = (X != Y)
CCL_ENCODE_SJIS	`en-sjis`	0x16	X = HIGHER_BYTE (SJIS (Y, Z)) r[7] = LOWER_BYTE (SJIS (Y, Z)
CCL_DECODE_SJIS	`de-sjis`	0x17	X = HIGHER_BYTE (DE-SJIS (Y, Z)) r[7] = LOWER_BYTE (DE-SJIS (Y, Z))

The CCL operators are as in C, with the addition of CCL_LSH8, CCL_RSH8, CCL_DIVMOD, CCL_ENCODE_SJIS, and CCL_DECODE_SJIS. The CCL_ENCODE_SJIS and CCL_DECODE_SJIS treat their first and second bytes as the high and low bytes of a two-byte character code. (SJIS stands for Shift JIS, an encoding of Japanese characters used by Microsoft. CCL_ENCODE_SJIS is a complicated transformation of the Japanese standard JIS encoding to Shift JIS. CCL_DECODE_SJIS is its inverse.) It is somewhat odd to represent the SJIS operations in infix form.

55.7.4 Calling CCL

CCL programs are called automatically during Emacs buffer I/O when the external representation has a coding system type of `shift-jis`, `big5`, or `ccl`. The program is specified by the coding system (see Section 55.6 [Coding Systems], page 743). You can also call CCL programs from other CCL programs, and from Lisp using these functions:

ccl-execute *ccl-program status* Function

Execute *ccl-program* with registers initialized by *status*. *ccl-program* is a vector of compiled CCL code created by `ccl-compile`. It is an error for the program to try to execute a CCL I/O command. *status* must be a vector of nine values, specifying the initial value for the R0, R1 .. R7 registers and for the instruction counter IC. A `nil` value for a register initializer causes the register to be set to 0. A `nil` value for the IC initializer causes execution to start at the beginning of the program. When the program is done, *status* is modified (by side-effect) to contain the ending values for the corresponding registers and IC.

ccl-execute-on-string *ccl-program status str* &optional *continue* Function

Execute *ccl-program* with initial *status* on *string*. *ccl-program* is a vector of compiled CCL code created by `ccl-compile`. *status* must be a vector of nine values, specifying the initial value for the R0, R1 .. R7 registers and for the instruction counter IC. A `nil` value for a register initializer causes the register to be set to 0. A `nil` value for the IC initializer causes execution to start at the beginning of the program. An optional fourth argument *continue*, if non-nil, causes the IC to remain on the unsatisfied read operation if the program terminates due to exhaustion of the input buffer. Otherwise the IC is set to the end of the program. When the program is done, *status* is modified (by side-effect) to contain the ending values for the corresponding registers and IC. Returns the resulting string.

To call a CCL program from another CCL program, it must first be registered:

register-ccl-program *name ccl-program* Function

Register *name* for CCL program *program* in `ccl-program-table`. *program* should be the compiled form of a CCL program, or nil. Return index number of the registered CCL program.

Information about the processor time used by the CCL interpreter can be obtained using these functions:

ccl-elapsed-time Function

Returns the elapsed processor time of the CCL interpreter as cons of user and system time, as floating point numbers measured in seconds. If only one overall value can be determined, the return value will be a cons of that value and 0.

ccl-reset-elapsed-time Function

Resets the CCL interpreter's internal elapsed time registers.

55.7.5 CCL Examples

This section is not yet written.

55.8 Category Tables

A category table is a type of char table used for keeping track of categories. Categories are used for classifying characters for use in regexps – you can refer to a category rather than having to use a complicated [] expression (and category lookups are significantly faster).

There are 95 different categories available, one for each printable character (including space) in the ASCII charset. Each category is designated by one such character, called a *category designator*. They are specified in a regexp using the syntax '`\cX`', where X is a category designator. (This is not yet implemented.)

A category table specifies, for each character, the categories that the character is in. Note that a character can be in more than one category. More specifically, a category table maps from a character to either the value `nil` (meaning the character is in no categories) or a 95-element bit vector, specifying for each of the 95 categories whether the character is in that category.

Special Lisp functions are provided that abstract this, so you do not have to directly manipulate bit vectors.

category-table-p *obj* Function

This function returns `t` if *arg* is a category table.

category-table &optional *buffer* Function

This function returns the current category table. This is the one specified by the current buffer, or by *buffer* if it is non-`nil`.

standard-category-table Function

This function returns the standard category table. This is the one used for new buffers.

copy-category-table &optional *table* Function

This function constructs a new category table and return it. It is a copy of the *table*, which defaults to the standard category table.

set-category-table *table* &optional *buffer* Function

This function selects a new category table for *buffer*. One argument, a category table. *buffer* defaults to the current buffer if omitted.

category-designator-p *obj* Function

This function returns `t` if *arg* is a category designator (a char in the range ‘‘ ’’ to ‘‘~’’).

category-table-value-p *obj* Function

This function returns `t` if *arg* is a category table value. Valid values are `nil` or a bit vector of size 95.

Appendix A Tips and Standards

This chapter describes no additional features of XEmacs 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 XEmacs 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 you write a function that you think ought to be added to Emacs under a certain name, such as `twiddle-files`, don't call it by that name in your program. Call it `mylib-twiddle-files` in your program, and send mail to '`bug-gnu-emacs@prep.ai.mit.edu`' suggesting we add it to Emacs. If and when we do, we can change the name easily enough.

  If one prefix is insufficient, your package may use two or three alternative common prefixes, so long as they make sense.

  Separate the prefix from the rest of the symbol name with a hyphen, '`-`'. This will be consistent with XEmacs itself and with most Emacs Lisp programs.
- It is often useful to put a call to `provide` in each separate library program, at least if there is more than one entry point to the program.
- If a file requires certain other library programs to be loaded beforehand, then the comments at the beginning of the file should say so. Also, use `require` to make sure they are loaded.
- If one file *foo* uses a macro defined in another file *bar*, *foo* should contain this expression before the first use of the macro:

  ```
  (eval-when-compile (require 'bar))
  ```

  (And *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 178.

  Using `eval-when-compile` avoids loading *bar* when the compiled version of *foo* is *used*.
- 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 26.4 [Hooks], page 378.
- If the purpose of a function is to tell you whether a certain condition is true or false, give the function a name that ends in '`p`'. If the name is one word, add just '`p`'; if the name is multiple words, add '`-p`'. Examples are `framep` and `frame-live-p`.

- If a user option variable records a true-or-false condition, give it a name that ends in '-flag'.
- Please do not define *C-c letter* as a key in your major modes. These sequences are reserved for users; they are the **only** sequences reserved for users, so 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 make that work go to waste, and inconvenience users.
- Sequences consisting of *C-c* followed by {, }, <, >, *:* or *;* are also reserved for major modes.
- Sequences consisting of *C-c* followed by any other punctuation character are allocated for minor modes. Using them in a major mode is not absolutely prohibited, but if you do that, the major mode binding may be shadowed from time to time by minor modes.
- You should not bind *C-h* following any prefix character (including *C-c*). If you don't bind *C-h*, it is automatically available as a help character for listing the subcommands of the prefix character.
- You should not bind a key sequence ending in ⟨ESC⟩ except following another ⟨ESC⟩. (That is, it is ok to bind a sequence ending in ⟨ESC⟩ ⟨ESC⟩.)

  The reason for this rule is that a non-prefix binding for ⟨ESC⟩ in any context prevents recognition of escape sequences as function keys in that context.
- Applications should not bind mouse events based on button 1 with the shift key held down. These events include *S-mouse-1*, *M-S-mouse-1*, *C-S-mouse-1*, and so on. They are reserved for users.
- Modes should redefine *mouse-2* as a command to follow some sort of reference in the text of a buffer, if users usually would not want to alter the text in that buffer by hand. Modes such as Dired, Info, Compilation, and Occur redefine it in this way.
- When a package provides a modification of ordinary Emacs behavior, it is good to include a command to enable and disable the feature, Provide a command named *whatever*-mode which turns the feature on or off, and make it autoload (see Section 14.2 [Autoload], page 198). Design the package so that simply loading it has no visible effect—that should not enable the feature. Users will request the feature by invoking the command.
- It is a bad idea to define aliases for the Emacs primitives. Use the standard names instead.
- Redefining 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 XEmacs, prominent comments at the beginning of the file should say which functions are replaced, and how the behavior of the replacements differs from that of the originals.

- Please keep the names of your XEmacs 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 34.2.4 [Text Lines], page 490.
- Don't call functions that set the mark, unless setting the mark is one of the intended features of your program. The mark is a user-level feature, so it is incorrect to change the mark except to supply a value for the user's benefit. See Section 35.6 [The Mark], page 504.

  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.
- Use lists rather than vectors, except when there is a particular reason to use a vector. Lisp has more facilities for manipulating lists than for vectors, and working with lists is usually more convenient.

  Vectors are advantageous for tables that are substantial in size and are accessed in random order (not searched front to back), provided there is no need to insert or delete elements (only lists allow that).
- The recommended way to print a message in the echo area is with the `message` function, not `princ`. See Section 45.3 [The Echo Area], page 648.
- 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 137.

  Do not use `message`, `throw`, `sleep-for`, or `beep` to report errors.
- An error message should start with a capital letter but should not end with a period.
- Try to avoid using recursive edits. Instead, do what the Rmail `e` command does: use a new local keymap that contains one command defined to switch back to the old local keymap. Or do what the `edit-options` command does: switch to another buffer and let the user switch back at will. See Section 19.10 [Recursive Editing], page 310.
- In some other systems there is a convention of choosing variable names that begin and end with '`*`'. We don't use that convention in Emacs Lisp, so please don't use it in your programs. (Emacs uses such names only for 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 XEmacs 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 XEmacs Lisp even when a compiled function is calling another compiled function.
- Using the primitive list-searching functions `memq`, `member`, `assq`, or `assoc` is even faster than explicit iteration. It 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 in byte-compiled code, avoiding the need for an ordinary function call. It is a good idea to use these functions rather than alternatives. To see whether a function is handled specially by the compiler, examine its `byte-compile` property. If the property is non-`nil`, then the function is handled specially.

  For example, the following input will show you that `aref` is compiled specially (see Section 6.3 [Array Functions], page 104) while `elt` is not (see Section 6.1 [Sequence Functions], page 101):

  ```
  (get 'aref 'byte-compile)
       ⇒ byte-compile-two-args

  (get 'elt 'byte-compile)
       ⇒ nil
  ```
- If calling a small function accounts for a substantial part of your program's running time, make the function inline. This eliminates the function call overhead. Since making a function inline reduces the flexibility of changing the program, don't do it unless it gives a noticeable speedup in something slow enough that users care about the speed. See Section 11.9 [Inline Functions], page 175.

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 variable or subroutine of a Lisp program might as well have a documentation string. In earlier Emacs versions, you could save space by using a comment instead of a documentation string, but that is no longer the case.
- The first line of the documentation string should consist of one or two complete sentences that stand on their own as a summary. *`M-x apropos`* displays just the first line,

and if it doesn't stand on its own, the result looks bad. In particular, start the first line with a capital letter and end with a period.

The documentation string can have additional lines that expand on the details of how to use the function or variable. The additional lines should be made up of complete sentences also, but they may be filled if that looks good.

- For consistency, phrase the verb in the first sentence of a documentation string as an infinitive with "to" omitted. For instance, use "Return the cons of A and B." in preference to "Returns the cons of A and B." Usually it looks good to do likewise for the rest of the first paragraph. Subsequent paragraphs usually look better if they have proper subjects.
- Write documentation strings in the active voice, not the passive, and in the present tense, not the future. For instance, use "Return a list containing A and B." instead of "A list containing A and B will be returned."
- Avoid using the word "cause" (or its equivalents) unnecessarily. Instead of, "Cause Emacs to display text in boldface," write just "Display text in boldface."
- Do not start or end a documentation string with whitespace.
- Format the documentation string so that it fits in an Emacs window on an 80-column screen. It is a good idea for most lines to be no wider than 60 characters. The first line can be wider if necessary to fit the information that ought to be there.

  However, rather than simply filling the entire documentation string, you can make it much more readable by choosing line breaks with care. Use blank lines between topics if the documentation string is long.
- **Do not** indent subsequent lines of a documentation string so that the text is lined up in the source code with the text of the first line. This looks nice in the source code, but looks bizarre when users view the documentation. Remember that the indentation before the starting double-quote is not part of the string!
- A variable's documentation string should start with '`*`' if the variable is one that users would often want to set interactively. If the value is a long list, or a function, or if the variable would be set only in init files, then don't start the documentation string with '`*`'. See Section 10.5 [Defining Variables], page 149.
- The documentation string for a variable that is a yes-or-no flag should start with words such as "Non-nil means...", to make it clear that all non-`nil` values are equivalent and indicate explicitly what `nil` and non-`nil` mean.
- When a function's documentation string mentions the value of an argument of the function, use the argument name in capital letters as if it were a name for that value. Thus, the documentation string of the function `/` refers to its second argument as '`DIVISOR`', because the actual argument name is `divisor`.

  Also use all caps for meta-syntactic variables, such as when you show the decomposition of a list or vector into subunits, some of which may vary.
- When a documentation string refers to a Lisp symbol, write it as it would be printed (which usually means in lower case), with single-quotes around it. For example: '`` `lambda' ``'. There are two exceptions: write `t` and `nil` without single-quotes.
- 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 Emacs displays the documentation string, it substitutes whatever key is currently bound to forward-char. (This is normally 'C-f', but it may be some other character if the user has moved key bindings.) See Section 27.3 [Keys in Documentation], page 384.

- In documentation strings for a major mode, you will want to refer to the key bindings of that mode's local map, rather than global ones. Therefore, use the construct '\\<...>' once in the documentation string to specify which key map to use. Do this before the first use of '\\[...]'. The text inside the '\\<...>' should be the name of the variable containing the local keymap for the major mode.

  It is not practical to use '\\[...]' very many times, because display of the documentation string will become slow. So use this to describe the most important commands in your major mode, and then use '\\{...}' to display the rest of the mode's keymap.

A.4 Tips on Writing Comments

We recommend these conventions for where to put comments and how to indent them:

';'
: Comments that start with a single semicolon, ';', should all be aligned to the same column on the right of the source code. Such comments usually explain how the code on the same line does its job. In Lisp mode and related modes, the *M-;* (indent-for-comment) command automatically inserts such a ';' in the right place, or aligns such a comment if it is already present.

  This and following examples are taken from the Emacs sources.

```
(setq base-version-list                 ; there was a base
      (assoc (substring fn 0 start-vn)  ; version to which
             file-version-assoc-list))  ; this looks like
                                        ; a subversion
```

';;'
: Comments that start with two semicolons, ';;', should be aligned to the same level of indentation as the code. Such comments usually describe the purpose of the following lines or the state of the program at that point. For example:

```
(prog1 (setq auto-fill-function
             ...
             ...
  ;; update modeline
  (redraw-modeline)))
```

  Every function that has no documentation string (because it is use only internally within the package it belongs to), should have instead a two-semicolon comment right before the function, explaining what the function does and how to call it properly. Explain precisely what each argument means and how the function interprets its possible values.

';;;'
: Comments that start with three semicolons, ';;;', should start at the left margin. Such comments are used outside function definitions to make general statements explaining the design principles of the program. For example:

```
;;; This Lisp code is run in XEmacs
;;; when it is to operate as a server
;;; for other processes.
```

Another use for triple-semicolon comments is for commenting out lines within a function. We use triple-semicolons for this precisely so that they remain at the left margin.

```
(defun foo (a)
;;; This is no longer necessary.
;;;  (force-mode-line-update)
  (message "Finished with %s" a))
```

'`;;;;`' Comments that start with four semicolons, '`;;;;`', should be aligned to the left margin and are used for headings of major sections of a program. For example:

```
;;;; The kill ring
```

The indentation commands of the Lisp modes in XEmacs, such as *M-;* (`indent-for-comment`) and ⟨TAB⟩ (`lisp-indent-line`) automatically indent comments according to these conventions, depending on the number of semicolons. See section "Manipulating Comments" in *The XEmacs Reference Manual*.

A.5 Conventional Headers for XEmacs Libraries

XEmacs 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 XEmacs.
copying permissions...
```

The very first line should have this format:

```
;;; filename --- description
```

The description should be complete in one line.

After the copyright notice come several *header comment* lines, each beginning with '`;;` *header-name*`:`'. Here is a table of the conventional possibilities for *header-name*:

'`Author`' This line states the name and net address of at least the principal author of the library.

If there are multiple authors, you can list them on continuation lines led by `;;` and a tab character, like this:

```
;; Author: Ashwin Ram <Ram-Ashwin@cs.yale.edu>
;;      Dave Sill <de5@ornl.gov>
;;      Dave Brennan <brennan@hal.com>
;;      Eric Raymond <esr@snark.thyrsus.com>
```

'Maintainer'
: This line should contain a single name/address as in the Author line, or an address only, or the string '`FSF`'. If there is no maintainer line, the person(s) in the Author field are presumed to be the maintainers. The example above is mildly bogus because the maintainer line is redundant.

  The idea behind the '`Author`' and '`Maintainer`' lines is to make possible a Lisp function to "send mail to the maintainer" without having to mine the name out by hand.

  Be sure to surround the network address with '`<...>`' if you include the person's full name as well as the network address.

'Created'
: This optional line gives the original creation date of the file. For historical interest only.

'Version'
: If you wish to record version numbers for the individual Lisp program, put them in this line.

'Adapted-By'
: In this header line, place the name of the person who adapted the library for installation (to make it fit the style conventions, for example).

'Keywords'
: This line lists keywords for the `finder-by-keyword` help command. This field is important; it's how people will find your package when they're looking for things by topic area. To separate the keywords, you can use spaces, commas, or both.

Just about every Lisp library ought to have the '`Author`' and '`Keywords`' header comment lines. Use the others if they are appropriate. You can also put in header lines with other header names—they have no standard meanings, so they can't do any harm.

We use additional stylized comments to subdivide the contents of the library file. Here is a table of them:

'`;;; Commentary:`'
: This begins introductory comments that explain how the library works. It should come right after the copying permissions.

'`;;; 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 XEmacs, 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 Building XEmacs; Allocation of Objects

This chapter describes how the runnable XEmacs executable is dumped with the preloaded Lisp libraries in it and how storage is allocated.

There is an entire separate document, the *XEmacs Internals Manual*, devoted to the internals of XEmacs from the perspective of the C programmer. It contains much more detailed information about the build process, the allocation and garbage-collection process, and other aspects related to the internals of XEmacs.

B.1 Building XEmacs

This section explains the steps involved in building the XEmacs executable. You don't have to know this material to build and install XEmacs, since the makefiles do all these things automatically. This information is pertinent to XEmacs maintenance.

The *XEmacs Internals Manual* contains more information about this.

Compilation of the C source files in the '`src`' directory produces an executable file called '`temacs`', also called a *bare impure XEmacs*. It contains the XEmacs Lisp interpreter and I/O routines, but not the editing commands.

Before XEmacs is actually usable, a number of Lisp files need to be loaded. These define all the editing commands, plus most of the startup code and many very basic Lisp primitives. This is accomplished by loading the file '`loadup.el`', which in turn loads all of the other standardly-loaded Lisp files.

It takes a substantial time to load the standard Lisp files. Luckily, you don't have to do this each time you run XEmacs; '`temacs`' can dump out an executable program called '`xemacs`' that has these files preloaded. '`xemacs`' starts more quickly because it does not need to load the files. This is the XEmacs executable that is normally installed.

To create '`xemacs`', use the command '`temacs -batch -l loadup dump`'. The purpose of '`-batch`' here is to tell '`temacs`' to run in non-interactive, command-line mode. ('`temacs`' can *only* run in this fashion. Part of the code required to initialize frames and faces is in Lisp, and must be loaded before XEmacs is able to create any frames.) The argument '`dump`' tells '`loadup.el`' to dump a new executable named '`xemacs`'.

The dumping process is highly system-specific, and some operating systems don't support dumping. On those systems, you must start XEmacs with the '`temacs -batch -l loadup run-temacs`' command each time you use it. This takes a substantial time, but since you need to start Emacs once a day at most—or once a week if you never log out—the extra time is not too severe a problem. (In older versions of Emacs, you started Emacs from '`temacs`' using '`temacs -l loadup`'.)

You are free to start XEmacs directly from '`temacs`' if you want, even if there is already a dumped '`xemacs`'. Normally you wouldn't want to do that; but the Makefiles do this when you rebuild XEmacs using '`make all-elc`', which builds XEmacs and simultaneously compiles any out-of-date Lisp files. (You need '`xemacs`' in order to compile Lisp files. However, you also need the compiled Lisp files in order to dump out '`xemacs`'. If both of these are missing or corrupted, you are out of luck unless you're able to bootstrap '`xemacs`' from '`temacs`'. Note that '`make all-elc`' actually loads the alternative loadup file 'loadup-el.el', which works like 'loadup.el' but disables the pure-copying process and forces XEmacs to ignore any compiled Lisp files even if they exist.)

You can specify additional files to preload by writing a library named 'site-load.el' that loads them. You may need to increase the value of PURESIZE, in 'src/puresize.h', to make room for the additional files. You should *not* modify this file directly, however; instead, use the '--puresize' configuration option. (If you run out of pure space while dumping 'xemacs', you will be told how much pure space you actually will need.) However, the advantage of preloading additional files decreases as machines get faster. On modern machines, it is often not advisable, especially if the Lisp code is on a file system local to the machine running XEmacs.

You can specify other Lisp expressions to execute just before dumping by putting them in a library named 'site-init.el'. However, if they might alter the behavior that users expect from an ordinary unmodified XEmacs, it is better to put them in 'default.el', so that users can override them if they wish. See Section 50.1.1 [Start-up Summary], page 691.

Before 'loadup.el' dumps the new executable, it finds the documentation strings for primitive and preloaded functions (and variables) in the file where they are stored, by calling Snarf-documentation (see Section 27.2 [Accessing Documentation], page 382). These strings were moved out of the 'xemacs' executable to make it smaller. See Section 27.1 [Documentation Basics], page 381.

dump-emacs *to-file from-file* — Function

This function dumps the current state of XEmacs 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 XEmacs that was already dumped, you must set command-line-processed to nil first for good results. See Section 50.1.4 [Command Line Arguments], page 694.

run-emacs-from-temacs &rest *args* — Function

This is the function that implements the 'run-temacs' command-line argument. It is called from 'loadup.el' as appropriate. You should most emphatically *not* call this yourself; it will reinitialize your XEmacs process and you'll be sorry.

emacs-version — Command

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

```
(emacs-version)
  ⇒ "XEmacs 20.1 [Lucid] (i586-unknown-linux2.0.29)
                 of Mon Apr  7 1997 on altair.xemacs.org"
```

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 XEmacs was built at the local site.

```
emacs-build-time "Mon Apr  7 20:28:52 1997"
     ⇒
```

emacs-version Variable

The value of this variable is the version of Emacs being run. It is a string, e.g. `"20.1 XEmacs Lucid"`.

The following two variables did not exist before FSF GNU Emacs version 19.23 and XEmacs version 19.10, which reduces their usefulness at present, but we hope they will be convenient in the future.

emacs-major-version Variable

The major version number of Emacs, as an integer. For XEmacs version 20.1, the value is 20.

emacs-minor-version Variable

The minor version number of Emacs, as an integer. For XEmacs version 20.1, the value is 1.

B.2 Pure Storage

XEmacs Lisp uses two kinds of storage for user-created Lisp objects: *normal storage* and *pure storage*. Normal storage is where all the new data created during an XEmacs 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 XEmacs.

Pure storage is allocated only while '`temacs`' is loading the standard preloaded Lisp libraries. In the file '`xemacs`', it is marked as read-only (on operating systems that permit this), so that the memory space can be shared by all the XEmacs jobs running on the machine at once. Pure storage is not expandable; a fixed amount is allocated when XEmacs is compiled, and if that is not sufficient for the preloaded libraries, '`temacs`' aborts with an error message. If that happens, you must increase the compilation parameter `PURESIZE` using the '`--puresize`' option to '`configure`'. This normally won't happen unless you try to preload additional libraries or add features to the standard ones.

purecopy *object* Function

This function makes a copy of *object* in pure storage and returns it. It copies strings by simply making a new string with the same characters in pure storage. It recursively copies the contents of vectors and cons cells. It does not make copies of other objects such as symbols, but just returns them unchanged. It signals an error if asked to copy markers.

This function is a no-op except while XEmacs is being built and dumped; it is usually called only in the file '`xemacs/lisp/prim/loaddefs.el`', but a few packages call it just in case you decide to preload them.

pure-bytes-used Variable

The value of this variable is the number of bytes of pure storage allocated so far. Typically, in a dumped XEmacs, 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 XEmacs initially (allowing those functions to be sharable and non-collectible). Dumping XEmacs as an executable always writes `nil` in this variable, regardless of the value it actually has before and after dumping.

You should not change this flag in a running XEmacs.

B.3 Garbage Collection

When a program creates a list or the user defines a new function (such as by loading a library), that data is placed in normal storage. If normal storage runs low, then XEmacs asks the operating system to allocate more memory in blocks of 2k bytes. Each block is used for one type of Lisp object, so symbols, cons cells, markers, etc., are segregated in distinct blocks in memory. (Vectors, long strings, buffers and certain other editing types, which are fairly large, are allocated in individual blocks, one per object, while small strings are packed into blocks of 8k bytes. [More correctly, a string is allocated in two sections: a fixed size chunk containing the length, list of extents, etc.; and a chunk containing the actual characters in the string. It is this latter chunk that is either allocated individually or packed into 8k blocks. The fixed size chunk is packed into 2k blocks, as for conses, markers, etc.])

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. XEmacs provides a *garbage collector* to reclaim this abandoned storage. (This name is traditional, but "garbage recycler" might be a more intuitive metaphor for this facility.)

The garbage collector operates by finding and marking all Lisp objects that are still accessible to Lisp programs. To begin with, it assumes all the symbols, their values and associated function definitions, and any data presently on the stack, are accessible. Any objects that can be reached indirectly through other accessible objects are also accessible.

When marking is finished, all objects still unmarked are garbage. No matter what the Lisp program or the user does, it is impossible to refer to them, since there is no longer a way to reach them. Their space might as well be reused, since no one will miss them. The second ("sweep") phase of the garbage collector arranges to reuse them.

The sweep phase puts unused cons cells onto a *free list* for future allocation; likewise for symbols, markers, extents, events, floats, compiled-function objects, and the fixed-size portion of strings. It compacts the accessible small string-chars chunks so they occupy fewer 8k blocks; then it frees the other 8k blocks. Vectors, buffers, windows, and other large objects are individually allocated and freed using `malloc` and `free`.

> **Common Lisp note:** unlike other Lisps, XEmacs 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
 (*plist*))

```
⇒ ((73362 . 8325) (13718 . 164)
(5089 . 5098) 949121 118677
(conses-used 73362 conses-free 8329 cons-storage 658168
symbols-used 13718 symbols-free 164 symbol-storage 335216
bit-vectors-used 0 bit-vectors-total-length 0
bit-vector-storage 0 vectors-used 7882
vectors-total-length 118677 vector-storage 537764
compiled-functions-used 1336 compiled-functions-free 37
compiled-function-storage 44440 short-strings-used 28829
long-strings-used 2 strings-free 7722
short-strings-total-length 916657 short-string-storage 1179648
long-strings-total-length 32464 string-header-storage 441504
floats-used 3 floats-free 43 float-storage 2044 markers-used 5089
markers-free 5098 marker-storage 245280 events-used 103
events-free 835 event-storage 110656 extents-used 10519
extents-free 2718 extent-storage 372736
extent-auxiliarys-used 111 extent-auxiliarys-freed 3
extent-auxiliary-storage 4440 window-configurations-used 39
window-configurations-on-free-list 5
window-configurations-freed 10 window-configuration-storage 9492
popup-datas-used 3 popup-data-storage 72 toolbar-buttons-used 62
toolbar-button-storage 4960 toolbar-datas-used 12
toolbar-data-storage 240 symbol-value-buffer-locals-used 182
symbol-value-buffer-local-storage 5824
symbol-value-lisp-magics-used 22
symbol-value-lisp-magic-storage 1496
symbol-value-varaliases-used 43
symbol-value-varalias-storage 1032 opaque-lists-used 2
opaque-list-storage 48 color-instances-used 12
color-instance-storage 288 font-instances-used 5
font-instance-storage 180 opaques-used 11 opaque-storage 312
range-tables-used 1 range-table-storage 16 faces-used 34
face-storage 2584 glyphs-used 124 glyph-storage 4464
specifiers-used 775 specifier-storage 43869 weak-lists-used 786
weak-list-storage 18864 char-tables-used 40
char-table-storage 41920 buffers-used 25 buffer-storage 7000
extent-infos-used 457 extent-infos-freed 73
extent-info-storage 9140 keymaps-used 275 keymap-storage 12100
consoles-used 4 console-storage 384 command-builders-used 2
command-builder-storage 120 devices-used 2 device-storage 344
frames-used 3 frame-storage 624 image-instances-used 47
image-instance-storage 3008 windows-used 27 windows-freed 2
window-storage 9180 lcrecord-lists-used 15
lcrecord-list-storage 360 hashtables-used 631
hashtable-storage 25240 streams-used 1 streams-on-free-list 3
streams-freed 12 stream-storage 91))
```

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.

plist A list of alternating keyword/value pairs providing more detailed information. (As you can see above, quite a lot of information is provided.)

gc-cons-threshold User Option

The value of this variable is the number of bytes of storage that must be allocated for Lisp objects after one garbage collection in order to trigger another garbage collection. A cons cell counts as eight bytes, a string as one byte per character plus a few bytes of overhead, and so on; space allocated to the contents of buffers does not count. Note that the subsequent garbage collection does not happen immediately when the threshold is exhausted, but only the next time the Lisp evaluator is called.

The initial threshold value is 500,000. If you specify a larger value, garbage collection will happen less often. This reduces the amount of time spent garbage collecting, but increases total memory use. You may want to do this when running a program that creates lots of Lisp data.

You can make collections more frequent by specifying a smaller value, down to 10,000. A value less than 10,000 will remain in effect only until the subsequent garbage collection, at which time `garbage-collect` will set the threshold back to 10,000. (This does not apply if XEmacs was configured with '`--debug`'. Therefore, be careful when setting `gc-cons-threshold` in that case!)

memory-limit Function

This function returns the address of the last byte XEmacs 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.

pre-gc-hook Variable

This is a normal hook to be run just before each garbage collection. Interrupts, garbage collection, and errors are inhibited while this hook runs, so be extremely careful in what you add here. In particular, avoid consing, and do not interact with the user.

post-gc-hook Variable

This is a normal hook to be run just after each garbage collection. Interrupts, garbage collection, and errors are inhibited while this hook runs, so be extremely careful in what you add here. In particular, avoid consing, and do not interact with the user.

gc-message Variable

This is a string to print to indicate that a garbage collection is in progress. This is printed in the echo area. If the selected frame is on a window system and `gc-pointer-glyph` specifies a value (i.e. a pointer image instance) in the domain of the selected frame, the mouse cursor will change instead of this message being printed.

gc-pointer-glyph Glyph

This holds the pointer glyph used to indicate that a garbage collection is in progress. If the selected window is on a window system and this glyph specifies a value (i.e. a pointer image instance) in the domain of the selected window, the cursor will be changed as specified during garbage collection. Otherwise, a message will be printed in the echo area, as controlled by `gc-message`. See Chapter 43 [Glyphs], page 625.

If XEmacs was configured with '`--debug`', you can set the following two variables to get direct information about all the allocation that is happening in a segment of Lisp code.

debug-allocation Variable

If non-zero, print out information to stderr about all objects allocated.

debug-allocation-backtrace Variable

Length (in stack frames) of short backtrace printed out by `debug-allocation`.

Appendix C Standard Errors

Here is the complete list of the error symbols in standard Emacs, grouped by concept. The list includes each symbol's message (on the `error-message` property of the symbol) and a cross reference to a description of how the error can occur.

Each error symbol has an `error-conditions` property that is a list of symbols. Normally this list includes the error symbol itself and the symbol `error`. Occasionally it includes additional symbols, which are intermediate classifications, narrower than `error` but broader than a single error symbol. For example, all the errors in accessing files have the condition `file-error`.

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 136, for an explanation of how errors are generated and handled.

symbol *string*; *reference*.

`error` `"error"`
See Section 9.5.3 [Errors], page 136.

`quit` `"Quit"`
See Section 19.8 [Quitting], page 306.

`args-out-of-range`
`"Args out of range"`
See Chapter 6 [Sequences Arrays Vectors], page 101.

`arith-error`
`"Arithmetic error"`
See `/` and `%` in Chapter 3 [Numbers], page 45.

`beginning-of-buffer`
`"Beginning of buffer"`
See Section 34.2 [Motion], page 488.

`buffer-read-only`
`"Buffer is read-only"`
See Section 30.7 [Read Only Buffers], page 438.

`cyclic-function-indirection`
`"Symbol's chain of function indirections contains a loop"`
See Section 8.2.4 [Function Indirection], page 123.

`domain-error`
`"Arithmetic domain error"`

`end-of-buffer`
`"End of buffer"`
See Section 34.2 [Motion], page 488.

`end-of-file`
: `"End of file during parsing"`
This is not a `file-error`.
See Section 17.3 [Input Functions], page 253.

`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 28 [Files], page 391.

`file-locked`
: This is a `file-error`.
See Section 28.5 [File Locks], page 398.

`file-already-exists`
: This is a `file-error`.
See Section 28.4 [Writing to Files], page 397.

`file-supersession`
: This is a `file-error`.
See Section 30.6 [Modification Time], page 437.

`invalid-function`
: `"Invalid function"`
See Section 8.2.3 [Classifying Lists], page 123.

`invalid-read-syntax`
: `"Invalid read syntax"`
See Section 17.3 [Input Functions], page 253.

`invalid-regexp`
: `"Invalid regexp"`
See Section 37.2 [Regular Expressions], page 550.

`mark-inactive`
: `"The mark is not active now"`

`no-catch`
: `"No catch for tag"`
See Section 9.5.1 [Catch and Throw], page 134.

`overflow-error`
: `"Arithmetic overflow error"`

`protected-field`
: `"Attempt to modify a protected field"`

`range-error`
: `"Arithmetic range error"`

`search-failed`
: `"Search failed"`
See Chapter 37 [Searching and Matching], page 549.

`setting-constant`
: `"Attempt to set a constant symbol"`
See Section 10.2 [Variables that Never Change], page 145.

`singularity-error`
: `"Arithmetic singularity error"`

`tooltalk-error`
: `"ToolTalk error"`
See Chapter 52 [ToolTalk Support], page 719.

`undefined-keystroke-sequence`
: `"Undefined keystroke sequence"`

`void-function`
: `"Symbol's function definition is void"`
See Section 11.8 [Function Cells], page 173.

`void-variable`
: `"Symbol's value as variable is void"`
See Section 10.6 [Accessing Variables], page 151.

`wrong-number-of-arguments`
: `"Wrong number of arguments"`
See Section 8.2.3 [Classifying Lists], page 123.

`wrong-type-argument`
: `"Wrong type argument"`
See Section 2.7 [Type Predicates], page 35.

These error types, which are all classified as special cases of `arith-error`, can occur on certain systems for invalid use of mathematical functions.

`domain-error`
: `"Arithmetic domain error"`
See Section 3.9 [Math Functions], page 56.

`overflow-error`
: `"Arithmetic overflow error"`
See Section 3.9 [Math Functions], page 56.

`range-error`
: `"Arithmetic range error"`
See Section 3.9 [Math Functions], page 56.

`singularity-error`
: `"Arithmetic singularity error"`
See Section 3.9 [Math Functions], page 56.

`underflow-error`
: `"Arithmetic underflow error"`
See Section 3.9 [Math Functions], page 56.

Appendix D Buffer-Local Variables

The table below lists the general-purpose Emacs variables that are automatically local (when set) in each buffer. Many Lisp packages define such variables for their internal use; we don't list them here.

`abbrev-mode`
: see Chapter 39 [Abbrevs], page 579

`auto-fill-function`
: see Section 36.13 [Auto Filling], page 529

`buffer-auto-save-file-name`
: see Section 29.2 [Auto-Saving], page 425

`buffer-backed-up`
: see Section 29.1 [Backup Files], page 421

`buffer-display-table`
: see Section 45.11 [Display Tables], page 659

`buffer-file-format`
: see Section 28.13 [Format Conversion], page 417

`buffer-file-name`
: see Section 30.4 [Buffer File Name], page 434

`buffer-file-number`
: see Section 30.4 [Buffer File Name], page 434

`buffer-file-truename`
: see Section 30.4 [Buffer File Name], page 434

`buffer-file-type`
: see Section 28.14 [Files and MS-DOS], page 419

`buffer-invisibility-spec`
: see Section 45.5 [Invisible Text], page 653

`buffer-offer-save`
: see Section 28.2 [Saving Buffers], page 394

`buffer-read-only`
: see Section 30.7 [Read Only Buffers], page 438

`buffer-saved-size`
: see Section 34.1 [Point], page 487

`buffer-undo-list`
: see Section 36.9 [Undo], page 523

`cache-long-line-scans`
: see Section 34.2.4 [Text Lines], page 490

`case-fold-search`
: see Section 37.7 [Searching and Case], page 565

`ctl-arrow`
    see Section 45.10 [Usual Display], page 658

`comment-column`
    see section "Comments" in *The XEmacs User's Manual*

`default-directory`
    see Section 50.3 [System Environment], page 697

`defun-prompt-regexp`
    see Section 34.2.6 [List Motion], page 493

`fill-column`
    see Section 36.13 [Auto Filling], page 529

`goal-column`
    see section "Moving Point" in *The XEmacs User's Manual*

`left-margin`
    see Section 36.16 [Indentation], page 534

`local-abbrev-table`
    see Chapter 39 [Abbrevs], page 579

`local-write-file-hooks`
    see Section 28.2 [Saving Buffers], page 394

`major-mode`
    see Section 26.1.4 [Mode Help], page 369

`mark-active`
    see Section 35.6 [The Mark], page 504

`mark-ring`
    see Section 35.6 [The Mark], page 504

`minor-modes`
    see Section 26.2 [Minor Modes], page 371

`modeline-format`
    see Section 26.3.1 [Modeline Data], page 373

`modeline-buffer-identification`
    see Section 26.3.2 [Modeline Variables], page 374

`modeline-format`
    see Section 26.3.1 [Modeline Data], page 373

`modeline-modified`
    see Section 26.3.2 [Modeline Variables], page 374

`modeline-process`
    see Section 26.3.2 [Modeline Variables], page 374

`mode-name`
    see Section 26.3.2 [Modeline Variables], page 374

`overwrite-mode`
: see Section 36.4 [Insertion], page 513

`paragraph-separate`
: see Section 37.8 [Standard Regexps], page 566

`paragraph-start`
: see Section 37.8 [Standard Regexps], page 566

`point-before-scroll`
: Used for communication between mouse commands and scroll-bar commands.

`require-final-newline`
: see Section 36.4 [Insertion], page 513

`selective-display`
: see Section 45.6 [Selective Display], page 654

`selective-display-ellipses`
: see Section 45.6 [Selective Display], page 654

`tab-width`
: see Section 45.10 [Usual Display], page 658

`truncate-lines`
: see Section 45.2 [Truncation], page 648

`vc-mode`
: see Section 26.3.2 [Modeline Variables], page 374

Appendix E Standard Keymaps

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

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

`bookmark-map`
: A keymap containing bindings to bookmark functions.

`Buffer-menu-mode-map`
: A keymap used by Buffer Menu mode.

`c++-mode-map`
: A keymap used by C`++` mode.

`c-mode-map`
: A keymap used by C mode. A sparse keymap used by C mode.

`command-history-map`
: A keymap used by Command History mode.

`ctl-x-4-map`
: A keymap for subcommands of the prefix *C-x 4*.

`ctl-x-5-map`
: A keymap for subcommands of the prefix *C-x 5*.

`ctl-x-map`
: A keymap for *C-x* commands.

`debugger-mode-map`
: A keymap used by Debugger mode.

`dired-mode-map`
: A keymap for `dired-mode` buffers.

`edit-abbrevs-map`
: A keymap used in `edit-abbrevs`.

`edit-tab-stops-map`
: A keymap used in `edit-tab-stops`.

`electric-buffer-menu-mode-map`
: A keymap used by Electric Buffer Menu mode.

`electric-history-map`
: A keymap used by Electric Command History mode.

`emacs-lisp-mode-map`
: A keymap used by Emacs Lisp mode.

`help-map`
: A keymap for characters following the Help key.

`Helper-help-map`
: A 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 keymap used by the *e* command of Info.

`Info-mode-map`
: A keymap containing Info commands.

`isearch-mode-map`
: A keymap that defines the characters you can type within incremental search.

`itimer-edit-map`
: A keymap used when in Itimer Edit mode.

`lisp-interaction-mode-map`
: A keymap used by Lisp mode.

`lisp-mode-map`
: A keymap used by Lisp mode.

A keymap for minibuffer input with completion.

`minibuffer-local-isearch-map`
: A keymap for editing isearch strings in the minibuffer.

`minibuffer-local-map`
: Default keymap to use when reading from the minibuffer.

`minibuffer-local-must-match-map`
: A keymap for minibuffer input with completion, for exact match.

`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.

`modeline-map`
: The keymap consulted for mouse-clicks on the modeline of a window.

`objc-mode-map`
: A keymap used in Objective C mode as a local map.

`occur-mode-map`
: A local keymap used by Occur mode.

`overriding-local-map`
: A keymap that overrides all other local keymaps.

`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.

`read-expression-map`
The minibuffer keymap used for reading Lisp expressions.

`read-shell-command-map`
The minibuffer keymap used by shell-command and related commands.

`shared-lisp-mode-map`
A keymap for commands shared by all sorts of Lisp modes.

`text-mode-map`
A keymap used by Text mode.

`toolbar-map`
The keymap consulted for mouse-clicks over a toolbar.

`view-mode-map`
A keymap used by View mode.

Appendix F Standard Hooks

The following is a list of hook variables that let you provide functions to be called from within Emacs on suitable occasions.

Most of these variables have names ending with '-hook'. They are *normal hooks*, run by means of run-hooks. The value of such a hook is a list of functions. The recommended way to put a new function on such a hook is to call add-hook. See Section 26.4 [Hooks], page 378, 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 arguments 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' or '-functions' 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-menubar-hook

activate-popup-menu-hook

ad-definition-hooks

adaptive-fill-function

add-log-current-defun-function

after-change-functions

after-delete-annotation-hook

after-init-hook

after-insert-file-functions

after-revert-hook

after-save-hook

after-set-visited-file-name-hooks

after-write-file-hooks

auto-fill-function

auto-save-hook

before-change-functions

before-delete-annotation-hook

before-init-hook

before-revert-hook

blink-paren-function

buffers-menu-switch-to-buffer-function

c++-mode-hook

c-delete-function

c-mode-common-hook

c-mode-hook

c-special-indent-hook
calendar-load-hook
change-major-mode-hook
command-history-hook
comment-indent-function
compilation-buffer-name-function
compilation-exit-message-function
compilation-finish-function
compilation-parse-errors-function
compilation-mode-hook
create-console-hook
create-device-hook
create-frame-hook
dabbrev-friend-buffer-function
dabbrev-select-buffers-function
delete-console-hook
delete-device-hook
delete-frame-hook
deselect-frame-hook
diary-display-hook
diary-hook
dired-after-readin-hook
dired-before-readin-hook
dired-load-hook
dired-mode-hook
disabled-command-hook
display-buffer-function
ediff-after-setup-control-frame-hook
ediff-after-setup-windows-hook
ediff-before-setup-control-frame-hook
ediff-before-setup-windows-hook
ediff-brief-help-message-function
ediff-cleanup-hook
ediff-control-frame-position-function
ediff-display-help-hook
ediff-focus-on-regexp-matches-function
ediff-forward-word-function
ediff-hide-regexp-matches-function

ediff-keymap-setup-hook
ediff-load-hook
ediff-long-help-message-function
ediff-make-wide-display-function
ediff-merge-split-window-function
ediff-meta-action-function
ediff-meta-redraw-function
ediff-mode-hook
ediff-prepare-buffer-hook
ediff-quit-hook
ediff-registry-setup-hook
ediff-select-hook
ediff-session-action-function
ediff-session-group-setup-hook
ediff-setup-diff-regions-function
ediff-show-registry-hook
ediff-show-session-group-hook
ediff-skip-diff-region-function
ediff-split-window-function
ediff-startup-hook
ediff-suspend-hook
ediff-toggle-read-only-function
ediff-unselect-hook
ediff-window-setup-function
edit-picture-hook
electric-buffer-menu-mode-hook
electric-command-history-hook
electric-help-mode-hook
emacs-lisp-mode-hook
fill-paragraph-function
find-file-hooks
find-file-not-found-hooks
first-change-hook
font-lock-after-fontify-buffer-hook
font-lock-beginning-of-syntax-function
font-lock-mode-hook
fume-found-function-hook
fume-list-mode-hook

fume-rescan-buffer-hook
fume-sort-function
gnus-startup-hook
hack-local-variables-hook
highlight-headers-follow-url-function
hyper-apropos-mode-hook
indent-line-function
indent-mim-hook
indent-region-function
initial-calendar-window-hook
isearch-mode-end-hook
isearch-mode-hook
java-mode-hook
kill-buffer-hook
kill-buffer-query-functions
kill-emacs-hook
kill-emacs-query-functions
kill-hooks
LaTeX-mode-hook
latex-mode-hook
ledit-mode-hook
lisp-indent-function
lisp-interaction-mode-hook
lisp-mode-hook
list-diary-entries-hook
load-read-function
log-message-filter-function
m2-mode-hook
mail-citation-hook
mail-mode-hook
mail-setup-hook
make-annotation-hook
makefile-mode-hook
map-frame-hook
mark-diary-entries-hook
medit-mode-hook
menu-no-selection-hook
mh-compose-letter-hook

mh-folder-mode-hook
mh-letter-mode-hook
mim-mode-hook
minibuffer-exit-hook
minibuffer-setup-hook
mode-motion-hook
mouse-enter-frame-hook
mouse-leave-frame-hook
mouse-track-cleanup-hook
mouse-track-click-hook
mouse-track-down-hook
mouse-track-drag-hook
mouse-track-drag-up-hook
mouse-track-up-hook
mouse-yank-function
news-mode-hook
news-reply-mode-hook
news-setup-hook
nongregorian-diary-listing-hook
nongregorian-diary-marking-hook
nroff-mode-hook
objc-mode-hook
outline-mode-hook
perl-mode-hook
plain-TeX-mode-hook
post-command-hook
post-gc-hook
pre-abbrev-expand-hook
pre-command-hook
pre-display-buffer-function
pre-gc-hook
pre-idle-hook
print-diary-entries-hook
prolog-mode-hook
protect-innocence-hook
remove-message-hook
revert-buffer-function
revert-buffer-insert-contents-function

```
rmail-edit-mode-hook
rmail-mode-hook
rmail-retry-setup-hook
rmail-summary-mode-hook
scheme-indent-hook
scheme-mode-hook
scribe-mode-hook
select-frame-hook
send-mail-function
shell-mode-hook
shell-set-directory-error-hook
special-display-function
suspend-hook
suspend-resume-hook
temp-buffer-show-function
term-setup-hook
terminal-mode-hook
terminal-mode-break-hook
TeX-mode-hook
tex-mode-hook
text-mode-hook
today-visible-calendar-hook
today-invisible-calendar-hook
tooltalk-message-handler-hook
tooltalk-pattern-handler-hook
tooltalk-unprocessed-message-hook
unmap-frame-hook
vc-checkin-hook
vc-checkout-writable-buffer-hook
vc-log-after-operation-hook
vc-make-buffer-writable-hook
view-hook
vm-arrived-message-hook
vm-arrived-messages-hook
vm-chop-full-name-function
vm-display-buffer-hook
vm-edit-message-hook
vm-forward-message-hook
```

vm-iconify-frame-hook
vm-inhibit-write-file-hook
vm-key-functions
vm-mail-hook
vm-mail-mode-hook
vm-menu-setup-hook
vm-mode-hook
vm-quit-hook
vm-rename-current-buffer-function
vm-reply-hook
vm-resend-bounced-message-hook
vm-resend-message-hook
vm-retrieved-spooled-mail-hook
vm-select-message-hook
vm-select-new-message-hook
vm-select-unread-message-hook
vm-send-digest-hook
vm-summary-mode-hook
vm-summary-pointer-update-hook
vm-summary-redo-hook
vm-summary-update-hook
vm-undisplay-buffer-hook
vm-visit-folder-hook
window-setup-hook
write-contents-hooks
write-file-data-hooks
write-file-hooks
write-region-annotate-functions
x-lost-selection-hooks
x-sent-selection-hooks
zmacs-activate-region-hook
zmacs-deactivate-region-hook
zmacs-update-region-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-**. Key entries are not permuted, however.

#

$

%

&

'

(

B

C

D

E

F

G

H

I

J

K

L

M

O

P

Q

R

T

U

V

W

X

Y

Z

Short Contents

Table of Contents