sun® microsystems

# Sun™ Common Lisp User's Guide

Part Number 800-1517-10
Revision: A February 2, 1987

sun
microsystems

# Sun™ Common Lisp User's Guide

## Credits and Trademarks

# About This Book

Sun Common Lisp is a computer programming language that you can use to develop software. Common Lisp is a functional programming language—one that is especially suited to the writing of large, complex programs that can manipulate symbols as well as numbers.

This book presents the fundamentals of using Common Lisp. It is not designed to be a comprehensive description and specification of the Common Lisp language. Rather, it is a concise guide to Sun Common Lisp.

## Organization of This Book

The *Sun Common Lisp User's Guide* has thirteen chapters and three appendixes.

- Chapter 1. "Introduction" contains an overview of the Common Lisp language. It also describes the notational conventions and syntax used throughout the book.
- Chapter 2. "Starting Up" discusses invoking the Lisp environment, saving Lisp images, exiting Lisp, customizing the Lisp environment, entering and exiting the Debugger, and using the display facilities.
- Chapter 3. "Debugging Lisp Programs" describes the features of the Debugger.
- Chapter 4. "Tracing Functions" describes the features of the Trace Facility, a tool for debugging programs.
- Chapter 5. "Stepping Through an Evaluation" explains the use of the Stepper, a tool for examining programs.
- Chapter 6. "Inspecting Data Structures" describes the Inspector, a tool for inspecting and modifying Lisp objects.
- Chapter 7. "The Foreign Function Interface" describes the interface to foreign files and libraries.
- Chapter 8. "Running UNIX Programs from Lisp" discusses the function that is used to run Shell programs from the Lisp environment.
- Chapter 9. "Compiling Lisp Programs" describes the Compiler, which transforms code to a more efficient form for execution.
- Chapter 10. "Storage Management in Common Lisp" describes the Garbage Collector and explains how storage is allocated.
- Chapter 11. "The Flavor System" describes the facility for object-oriented programming.
- Chapter 12. "The Window Tool Kit" describes the tools for creating and accessing windows, bitmaps, and viewports.
- Chapter 13. "The Editor" is a guide to the Editor and its commands.
- Appendix A. "Alphabetical Listing of Common Lisp Functions" is a list of all the Common Lisp functions and extensions to Common Lisp described in the *Sun Common Lisp Reference Manual.*
- Appendix B. "Extensions to Common Lisp" is a list of all the extensions that are described in this guide.

- Appendix C. "Implementing Editor Commands" explains the data structures used by the Editor and describes the Lisp functions and macros that can be used to implement new Editor commands.

## Related Publications

The following publications contain related information that you may find useful:

- *Sun Common Lisp Reference Manual* is a complete technical reference to the language.
- *Common Lisp: The Language* by Guy L. Steele Jr. (Digital Press) is the basic implementation specification for the language.
- *Programming in Common Lisp* by Rodney A. Brooks (John Wiley & Sons) is an introductory text for those who are new to Lisp.

# Contents

# Figures

# Chapter 1. Introduction

# Chapter 1. Introduction

## About Common Lisp

Sun Common Lisp is a complete implementation of the Common Lisp language. It includes all of the Common Lisp functions, constants, variables, macros, and special forms. In addition, Sun Common Lisp provides many functions as extensions to Common Lisp and as enhancements to the user environment.

### The Language

Common Lisp is a functional, or applicative, language. It has two salient features—a list-based representation of data and an evaluator, or interpreter, that treats some lists as programs.

Lisp functions are equivalent to subroutines or procedures in other languages. In contrast to most other languages, Lisp functions can create and return arbitrary data objects as their values. These data objects can then be passed as arguments to other functions.

Programs and data have the same form in Lisp, and thus Lisp programs can easily process other Lisp programs. Programs are sequences of expressions composed of function calls.

While iteration, or looping, as a control structure is common in most programming languages, Lisp makes extensive use of recursion.

### The Environment

The Lisp system is an interactive one. When you type an expression at the terminal, Lisp evaluates it and displays the result automatically. Other programming languages compute by compiling and running programs. Lisp computes by evaluating the expressions that are typed to it.

Sun Common Lisp has a compiler that compiles Lisp code into machine code. User programs may run more efficiently as a result.

Debugging in Lisp can be done as a program is written. Every expression typed to Lisp is evaluated, and therefore at each stage of testing, the Lisp environment is available for examining the state of a program and its data structures. Large, complex programs can be incrementally built and tested.

Lisp manages storage by providing a dynamic heap of storage that is automatically allocated as needed and then reclaimed, or garbage collected, when no longer needed.

## Notational Conventions and Syntax

This manual adheres to a number of notational conventions.

### Syntactic Descriptions

The names of all Common Lisp functions, macros, special forms, constants, and variables are in boldface (**max**, for example). Names of the parameters are in italics (*number*, for example).

The syntactic descriptions of Common Lisp functions are presented using the Common Lisp lambda list syntax. Lambda lists consist of a series of arguments and lambda list keywords. The lambda list keywords indicate how arguments are processed; they do not appear in the actual function call form. In the syntactic descriptions of functions, they appear in a typewriter font.

- Required parameters appear first, immediately following the function name.
- Any optional parameters are specified next. They are preceded by the `&optional` lambda list keyword. Use of the `&optional` lambda list keyword indicates that arguments that follow it are optional.
- An `&rest` parameter may be specified next. It is preceded by the `&rest` lambda list keyword. Use of the `&rest` parameter indicates that an indefinite number of arguments may appear in the function call form and are bound to that parameter.
- The lambda list keyword `&key` indicates that the function accepts keyword arguments. The lambda list keyword `&key` is followed by the keywords that are permitted. Keywords are symbols preceded by a colon (`:start`, `:end`, `:count`, and so forth). When the function is called, a keyword argument is specified by giving the keyword itself, followed by the value that the keyword argument is to have. The keyword-value pairs may occur in any order in the argument list; they are not constrained by the order of the keyword parameters in the lambda list.

The first box illustrates the syntactic description of a Common Lisp function. When a function is called, its name and arguments, except for keyword arguments, must be typed in the order shown. Arguments may appear across several lines, since carriage-returns and linefeeds can occur wherever a space can occur and do not have any special meaning to the Lisp reader (the input-handling part of the Lisp system).

| | |
|---|---|
| **max** *number* **&rest** *more-numbers* | [*Function*] |

The expressions

```
(max 1)

(max 2)

(max 1 2 3)
```

represent syntactically correct calls to the function **max**.

The syntactic descriptions of Common Lisp macros and special forms are given in an extended Backus-Naur form (BNF) notation.

- A word in italics indicates a syntactic category (for example, *symbol*, *argument*, *variable*).
- Braces, brackets, stars, plus signs, and vertical bars are metasyntactic marks.
- Braces, { and }, group what they enclose. Braces may be followed by a star (*), which indicates that what they enclose may appear any number of times or not at all, or they may be followed by a plus sign (+), which indicates that what is enclosed may appear any nonzero number of times (that is, must appear at least once).

| | |
|---|---|
| $\{x\}^*$ | zero or more occurrences of $x$ |
| $\{x\}^+$ | one or more occurrences of $x$ |

- Brackets, [ and ], indicate that what they enclose is optional and can appear only once.

| | |
|---|---|
| $[x]$ | zero or one occurrences of $x$ |

- A vertical bar ( | ) separates mutually exclusive alternatives.
- The symbol ::= means "is defined by." It indicates that the term on the left side is defined by the expression on the right.

The boxed examples that follow illustrate the syntactic descriptions for macros and special forms. While functions are called according to a uniform syntax, the syntax of macros and special forms tends to vary widely.

This box shows the syntax of a macro:

**prog** ({*var* | (*var* [*init*] )}*) {*declaration*}* {*tag* | *statement*}* [*Macro*]

The following is a syntactically correct use of the **prog** macro:

```
(prog (x)
      (setq x 2)
      (return x))
```

This box shows the syntax of a special form:

**if** *test then* [*else*] [*Special Form*]

The expressions shown below are syntactically correct calls to the **if** special form.

```
(if t 1 2)

(if t 1)
```

The next box illustrates the documentation of a global variable. Note that global variables in Common Lisp by convention have names that begin and end with an asterisk.

***print-radix*** [*Variable*]

The following box illustrates the documentation of a constant:

**pi** [*Constant*]

## Examples and Code

The examples represent what is displayed on the screen during interaction with Lisp. The Common Lisp prompt is given by `>`. The expression that follows it displays what you have entered at the keyboard. This in turn is followed by the response of the Lisp system. Examples are printed in a typewriter font.

Lisp code in this manual is in lowercase. In general, the Lisp reader converts symbols into uppercase, and the Lisp system displays its responses in uppercase. You can write programs in either uppercase or lowercase, or a combination of the two, whichever you prefer.

In the text of this manual, everything that would be typed at the keyboard or that would appear on the terminal screen is typeset in a typewriter font with this exception: an argument or parameter is printed in italics, indicating that it serves as a placeholder for a real argument value that you are to supply.

Normal text is set in a roman font.

Numbers, including those appearing in examples, are in decimal format unless explicitly noted otherwise.

Parentheses stand for themselves. Parentheses enclose lists. Lists may contain zero or more items, including other lists. Calls to functions, special forms, and macros are lists and are therefore enclosed in parentheses.

The single quote character (`'`) is an abbreviation for the Lisp function **quote**. Thus, evaluating the Lisp expression `'`*form* is the same as evaluating the expression `(quote` *form*`)`. It means that the form following `quote` is not evaluated.

The semicolon character (`;`) indicates the beginning of a comment. A comment extends from the semicolon to the end of the line.

The `#|` and `|#` characters are nested comment characters that may appear in examples of code. They comment out sections of code.

The `#'` character is an abbreviation for the Lisp function **function**. Thus, evaluating the Lisp expression `#'`*function* is the same as evaluating the Lisp expression `(function` *function*`)`. It indicates that the form that follows it is to be interpreted as a function object.

The `#` syntax is used in the printed representation of many data types.

# Chapter 2. Starting Up

# Chapter 2. Starting Up

## Starting Up Lisp

Sun Common Lisp is an interactive environment. This section explains how to start up Lisp, how to use the facility known as the Debugger from within the Lisp environment, and how to return to what is known as the top level of Lisp.

### Invoking Lisp

Sun Common Lisp is invoked by typing the command **lisp** to the operating system prompt:

```
% lisp
```

After executing its initialization routines, Lisp responds with a prompt:

```
>
```

The prompt indicates that Lisp is ready to read an expression that is typed by the user at the terminal and to evaluate it. After evaluation, Lisp displays on the screen a response that is immediately followed by another prompt. This routine is known as the ***read-eval-print loop***, or the top-level loop. While operating in this fashion, Lisp is said to be at the top level.

### Entering and Exiting the Debugger

If Lisp reads an invalid form, encounters an interrupt, or for any reason cannot perform an evaluation, an error is signaled and the Debugger is entered. Upon entry, the Debugger prints a characterization of the type of error, which is followed by the Debugger prompt:

```
->
```

In the Debugger, the Lisp environment is available for evaluating and inspecting Lisp expressions. This means that any expression that can be typed to the top-level prompt can also be typed to the Debugger prompt.

The manner in which the Debugger evaluates what it reads and displays a response is similar to the evaluation that occurs at the top level. If the Debugger encounters an error, you enter a lower level of the Debugger. When this happens, the Debugger again characterizes the error and displays an additional prompt:

```
->->
```

The number of consecutive Debugger prompts corresponds to the Debugger level to which you are typing.

After characterizing an error, the Debugger displays a list of commands for responding to the situation. This list varies depending on the type of error but always includes the **:a** command:

```
:A    Abort to Lisp Top Level
```

By typing **:a** you exit the Debugger and reenter the top level of Lisp. (Because **:a** is a constant, it is not an error for the top level to read and evaluate it. The ensuing evaluation returns **:a**.)

See the chapter "Debugging Lisp Programs" for a more detailed discussion of the Debugger.

## Exiting Lisp

You can exit Lisp and return to the operating system environment by typing either the function **quit** or the function **abort** to the top-level prompt or to the Debugger prompt:

```
> (quit)
%
```

or

```
> (abort)
%
```

## Customizing the Lisp Environment

You can customize your Lisp environment by creating a file of expressions to be evaluated before the top-level prompt is printed. This file should be created on your home directory and named `"lisp-init.lisp"`, `"lisp-init.lbin"`, or `"lisp-init.2bin"`. The extensions `.lbin` and `.2bin` denote binary files compiled from source `.lisp` files; the `.lbin` extension indicates that the default value for the **:target** option applies, and the `.2bin` extension indicates that the source file was compiled with the **:target** option of 68020.

When you invoke Lisp, your home directory is first searched for a file named `"lisp-init.lbin"` or `"lisp-init.2bin"` and then for a file named `"lisp-init.lisp"`. If the file in question exists, it is loaded. If both the source and binary versions of the file exist, the appropriate binary version is loaded if it is more recent. If the source version is more recent, you will be asked which version to use. The files can contain arbitrary Lisp expressions, which are evaluated when the file is loaded. If none of the files exists, no error is signaled.

## Saving Lisp Images

You may save your Lisp images with the function **disksave**, which is an extension to Common Lisp. The function saves a copy of the executing Lisp image on disk.

## Using the Display Facilities

Sun Common Lisp normally starts up in ***terminal interaction mode***. In this mode, the characters you type are read directly by the top-level Lisp reader.

If you wish, you can start up Lisp in ***Editor interaction mode*** rather than in terminal interaction mode. In Editor interaction mode, the characters you type are displayed in an Editor buffer called the Lisp Buffer. The behavior of Lisp is the same in both modes except that in Editor interaction mode you can use Editor commands. In the Lisp Buffer, the Editor maintains a record of your interaction with Lisp, which you can review and correct.

Lisp will start up in Editor interaction mode when you type **lisp** to the Shell prompt if you have the function **ed** in your initialization file (`"lisp-init.lisp"`, `"lisp-init.lbin"`, or `"lisp-init.2bin"`). Lisp will also start up in Editor interaction mode if you save your Lisp image by calling the function **disksave** with **ed** specified as the value of the keyword argument **:restart-function**.

From terminal interaction mode, you can enter or reenter the Editor by using the function **ed**. You can use the key sequence **Ctrl-XCtrl-Z** to exit the Editor and enter Lisp in terminal interaction mode.

All of the Editor facilities are available within the Lisp Buffer, which is integrated with the Lisp environment. The Editor provides a wide variety of commands for manipulating text, for evaluating Lisp expressions, and for compiling functions. You can, for example, create new Editor buffers, edit source code text files, copy source code text to the Lisp Buffer, and execute code in the Lisp Buffer.

The Editor can be integrated with the Window Tool Kit on graphics displays that support the **suntools** system. You must start up Lisp from within the **suntools** environment. Then, by calling the function **ed**, you initialize the Window Tool Kit and enter Editor interaction mode. The Window Tool Kit displays the Lisp Buffer, a scratch buffer, and any other Editor buffers that are current.

In the Window Tool Kit environment, each window created by the Editor has a pop-up menu that describes the various commands that you may use. You can access this menu by clicking the right mouse button on the window you wish to manipulate. The menu lists the options for moving a window and for changing its size, and you can use the mouse to select an option by clicking the right mouse button. To mark text and to move the cursor, you click the left mouse button. See the chapter "The Window Tool Kit" for more detailed information.

You can also use the Editor window commands to select and delete windows. For example, the **New Window** command (**Ctrl-X2**) creates a new Editor window. The **Next Window** command (**Ctrl-XN**) allows you to select other windows. To delete a window, you can invoke the **Delete Window** (**Ctrl-XD**) command. See the chapter "The Editor" for details on all window and buffer commands.

## Sample Lisp Program

Displayed below is a sample Lisp program for balancing bank accounts. It is in a file called `"sample.lisp"` that is loaded into Lisp as shown in the session transcript in the following section. In this program, each time the function `create-account` is called, it creates an account that is a lexical closure.

Lexical closures can be viewed as data objects with local variable bindings. They are more fully explained in the *Sun Common Lisp Reference Manual.*

This particular closure includes the variable `balance`. Such a closure behaves like a function, that is, you can invoke it with the function **funcall**. The account closures can be called with such "verbs" as `deposit`, `withdraw`, and `query`. In the program, another function—`transfer`—is defined that permits the transfer of funds between accounts.

### The Program File

The file `"sample.lisp"` contains the program.

```
;;; Create a closure to represent a particular account.

(defun create-account (account-type)
  (let ((balance 0))
    (function
     (lambda (transaction-type &optional (amount 0))
      (case transaction-type

        ;; Add to the account

        ((deposit credit)
         (when (minusp amount)
           (format
            t
            "You can't deposit a negative amount of money.~@
             Try withdrawing instead.~%")
           (setf amount 0))
         (format
          t
```

```
       "Crediting $~$ to your ~A account, for a new balance ~
        of $~$.~%"
       amount
       account-type
       (incf balance amount))
      amount)

     ;; Subtract from the account

     ((withdraw debit deduct)
      (when (minusp amount)
        (format
         t
         "You can't withdraw a negative amount of money.~@
          Try depositing instead.~%")
        (setf amount 0))
      (when (> amount balance)
        (format
         t
        "Your ~A account lacks sufficient funds ~
         for a $~$ withdrawal.~%" account-type amount)
        (setf amount 0))
      (format
       t
       "Deducting $~$ from your ~A account, leaving $~$ in the ~
        account.~%"
       amount
       account-type
       (decf balance amount))
      amount)

    ;; Set the balance

     (set-balance
      (format
       t
       "Setting your ~A account balance to $~$~%"
        account-type
        amount)
      (setf balance amount))

     ;; Request the current balance

     ((query inquire)
      (format
       t
       "Your ~A account contains $~$~%" account-type balance)
      balance)
```

```
        ;; Unknown transaction-type

        (otherwise
         (cerror "Ignore unknown-type transaction."
                 "Unknown transaction-type: ~A"
                 transaction-type))))))

;;; Withdraw funds from one account and deposit them in another.

(defun transfer (amount source-account destination-account)
  (funcall destination-account
           'deposit
           (funcall source-account 'withdraw amount)))
```

## Terminal Session

The program file "sample.lisp" is loaded into Lisp as shown in the following terminal session. The session illustrates the interactive nature of programming in Lisp and demonstrates the use of some fundamental features of Common Lisp, such as closures. The session also demonstrates what to do when errors are signaled.

```
> (load "sample")
#P"sample.lisp"
> (compile 'create-account)
CREATE-ACCOUNT
> (compile 'transfer)
TRANSFER
> (setq checking (create-account "checking"))
#<Compiled-Function 4AB6CF>
> (setq savings (create-account "savings"))
#<Compiled-Function 4AB7DF>
> (funcall checking 'set-balance 22.25)
Setting your checking account balance to $22.25
22.25
> (funcall savings 'set-balance 543.12)
Setting your savings account balance to $543.12
543.12
> (funcall checking 'withdraw 17.34)
Deducting $17.34 from your checking account, leaving $4.91 in the account.
17.34
> (transfer 80.00 savings checking)
Deducting $80.00 from your savings account, leaving $463.12 in the account.
Crediting $80.00 to your checking account, for a new balance of $84.91.
80.0
```

```
> (funcall checking 'credti 350.00)
>>Error: Unknown transaction-type: CREDTI

CERROR:
   Required arg 0 (CONTINUE-FORMAT-STRING): "Ignore unknown-type transac-
tion."
   Required arg 1 (FORMAT-STRING): "Unknown transaction-type: ~A"
   Rest arg (FORMAT-ARGS): (CREDTI)

:A    Abort to Lisp Top Level
:C    Ignore unknown-type transaction.
-> :c
Ignore unknown-type transaction.
NIL
> (funcall checking 'credit 350.00)
Crediting $350.00 to your checking account, for a new balance of $434.91.
350.0
> (funcall checking 'deduct 500.00)
Your checking account lacks sufficient funds for a $500.00 withdrawal.
Deducting $0. from your checking account, leaving $434.91 in the account.
0
> (funcall checking 'query)
Your checking account contains $434.91
434.91
> (setf ira (create-account "retirement"))
#<Compiled-Function 4ACCC7>
> (funcall ira 'credit 2000.00)
Crediting $2000.00 to your retirement account, for a new balance of
$2000.00.
2000.0
> (quit)
%
```

## abort

**Purpose:** The function **abort** always terminates the Lisp environment. It immediately returns you to the operating system environment and reports *status* as the exit status of the Lisp process.

**Syntax:** **abort** `&optional` *status* [*Function*]

**Remarks:** Unlike the function **quit**, the function **abort** exits the Lisp environment immediately.

The optional argument *status* sets the exit status of the process that was running Lisp. It defaults to 0.

This function is an extension to Common Lisp.

**Examples:**

```
> (abort)
%
```

**See Also:** **quit**

## disksave

**Purpose:** The function **disksave** saves a copy of an executing Lisp image on disk.

**Syntax:**

```
disksave target-file &key :restart-function :full-gc :gc        [Function]
                          :reserved-free-segments
                          :dynamic-free-segments :verbose
```

**Remarks:** The argument *target-file* is a simple string containing the system-dependent namestring of the Lisp image file to be created; no pathname defaults are applied.

When the resulting disk file is executed, Lisp first calls any function specified as a value for the keyword argument **:restart-function**. The restart-function is called with no arguments. Lisp then processes the usual initialization file if one exists. Finally, Lisp enters the normal read-eval-print loop. If the restart-function does not return, no initialization or top-level evaluation is done. In this way, the saved image can be made to behave like a program.

The keyword arguments **:full-gc** and **:gc** control garbage collection during the save. If **:full-gc** is specified and non-**nil**, a garbage collection is performed before the image is saved. After the garbage collection, all dynamic storage is converted to static storage. Objects converted from dynamic storage to static storage are never recopied during a garbage collection. However, they are scanned for pointers to update. As a result, performance improvements in garbage collection may occur when large systems are loaded into Lisp and saved.

If **:gc** is specified and non-**nil** and **:full-gc** is unspecified or **nil**, a normal garbage collection is performed before the image is saved. Due to memory management constraints, **disksave** may force a garbage collection in addition to any requested through use of the **:full-gc** and **:gc** keywords.

**Note:** If neither **:gc** nor **:full-gc** is specified, the saved image may contain all of the garbage that is currently in the image, and therefore the files saved on disk may be considerably larger than necessary. When you call **disksave**, specifying **:gc** or **:full-gc** is recommended.

The keyword argument **:reserved-free-segments** specifies the number of segments of reserved memory that should be free in the saved image; if used, it should be set to a positive integer. Specifying a large number of segments permits you to load large amounts of code into the image or to create many foreign data structures without having to expand reserved memory and garbage collect. However, specifying a large number of segments also increases the size of your saved image as well as the amount of memory that is required to start it up.

The keyword argument **:dynamic-free-segments** specifies the number of segments of dynamic-0 space that should be free in the image; if used, it should be set to a positive integer. The total amount of dynamic space will be as follows:

(***amount of dynamic-0 space in use*** + ***free dynamic-0 space requested***) × 2

The amount is doubled because an equal amount of dynamic-1 space will be allocated. If you do not specify this keyword, all dynamic space that is allocated in the running image will be preserved when you call **disksave**.

If **:verbose** is specified and non-**nil**, the progress of the saving process is reported to the terminal.

If the Window Tool Kit has been initialized and the function **disksave** is invoked, the window environment is temporarily suspended. Once the **disksave** function has saved the Lisp image on disk, the windows on the running Lisp image are restored automatically to the state they were in before the **disksave** function call. To restore the state of the windows in the newly saved image, call the function **initialize-windows** or the function **ed** with no arguments (see the chapter "The Window Tool Kit").

This function is an extension to Common Lisp.

**Examples:**

```
> (disksave "test-lisp")                             ; Saves Lisp as
                                                     ; "test-lisp".

> (disksave "appl" :restart-function #'init-function) ; Saves Lisp as
                                                     ; "appl".
                                                     ; init-function is
                                                     ; called on start-up
                                                     ; of "appl".
```

## ed

**Purpose:** The function **ed** invokes the Editor.

If you are in the **suntools** environment, the function **ed** initializes the Window Tool Kit and invokes the Editor in a default display. The default display consists of a Lisp Buffer window that occludes a scratch buffer window.

If you do not specify an optional argument to **ed**, you enter the Editor in the same state in which you last left it.

**Syntax:** **ed &optional** *x* **&key :windows &allow-other-keys** [*Function*]

**Remarks:** You can specify a pathname, a string, or a symbol as the optional argument *x*. If you specify either a pathname or a string argument, **ed** allows you to edit the contents of the indicated file. If you specify a symbol argument that represents the name of an interpreted function, **ed** pretty-prints the corresponding function into a buffer that becomes the current buffer. You may then edit the text of the function definition. Any interpreted function that you edit must be reevaluated to make the changes effective in the current Lisp environment; the edited version is treated as a new function definition.

By default, the Editor starts up in the window environment, if one exists. If you have already used the Editor during the current Lisp session, buffers and window configurations that you established earlier are restored.

If you specify the keyword **:windows**, its argument must have one of the following values:

- **nil**

  If the keyword argument has this value, the Editor starts up as a terminal editor.

- **t**

  If the keyword argument has this value, the Editor starts up in the available window environment; if no window environment is available, an error is signaled.

- **:default**

  If the keyword argument has this value, the Editor restarts in the available window environment with a default configuration of windows. Changes you made to the window configuration in previous editing sessions are not retained. If no window environment is available, an error is signaled.

If you start up the Editor as a terminal editor, you cannot use it in the window environment in subsequent editing sessions. Similarly, if you start up the Editor in the Window Tool Kit, you must remain in the window environment in subsequent editing sessions.

You may also specify any of the keyword options that are valid for the function **initialize-windows**. These options are passed by **ed** to **initialize-windows** to initialize the Window Tool Kit in an environment that supports a window system. By invoking the function **windows-available-p**, you can determine if a window environment is available.

See the chapters "The Editor" and "The Window Tool Kit" for more information.

The keyword **:windows** and the keyword options that are passed to the function **initialize-windows** are extensions to the Common Lisp function **ed**.

**See Also:** **initialize-windows**

**windows-available-p**

## quit

**Purpose:** The function **quit** terminates the Lisp environment. Before returning to the operating system environment, however, **quit** exits to the top level of Lisp by using the special form **throw**. Therefore, if **quit** is called from inside the special form **unwind-protect**, all of the clean-up forms specified by the invocation of **unwind-protect** are executed before **quit** returns to the operating system environment. Thus, **quit** can be used to close all files.

**Syntax:** **quit** `&optional` *status* [*Function*]

**Remarks:** The optional argument *status* sets the exit status of the process that was running Lisp. It defaults to 0.

This function is an extension to Common Lisp.

**Examples:**

```
> (quit)
%
```

**See Also:** **abort**

# Chapter 3. Debugging Lisp Programs

# Chapter 3. Debugging Lisp Programs

## The Debugger

In Sun Common Lisp, the facility called the Debugger allows interactive examination and modification of elements of the Lisp environment, while providing the same interpretive facilities that are available at the top level of Lisp. Unlike the top level, the Debugger can be called recursively; that is, you can enter lower levels of it from inside the facility itself. At each recursive level of the Debugger, instructions for exiting and returning to the top level of Lisp are automatically displayed.

## Invoking the Debugger

The Debugger can be invoked by signals generated by Lisp and by signals generated by the operating system. A Lisp-generated signal is created by calls to the functions **error**, **cerror**, or **break**, or by a call to the macro **trace** (with the keyword **:break** set to non-**nil**). Calls such as these suspend Lisp evaluation and enter the Debugger in a controlled, explicit manner.

Signals, or interrupts, generated by the operating system can be encountered at any time and thus may interrupt the normal flow of Lisp evaluation in an unpredictable manner. The most common and useful interrupt is the keyboard interrupt.

The examples in this chapter assume that the interrupt character is **Control-C**.

The Debugger recognizes the manner in which it was invoked. The type of signal that caused invocation is identified upon entry and displayed on the screen followed by a colon (:) and a brief diagnosis of the calling situation. For example, if a keyboard interrupt is encountered, the Debugger prints the following:

```
>> Interrupt: Interrupt
```

The wrong number of arguments to a function looks like this:

```
 > (load)
 >>Error: LOAD cannot be called with 0 arguments
```

When you call the function **error** with the mandatory string argument, the argument is taken as the diagnosis:

```
> (error "This is my error.")
>>Error: This is my error.
```

The style of this last example is useful for building user-defined programs and functions. For example, you can construct a function that signals an error if its argument is not an integer:

```
>(defun integer-only(arg)
    (cond ((not (integerp arg)) (error "Argument is not an integer"))
          (t (+ 2 arg))))
INTEGER-ONLY
> (integer-only 3)
5
> (integer-only 'foo)
>>Error: Argument is not an integer
```

## The Debugger Stack

The Debugger makes the dynamic state of the Lisp environment available for examination by organizing this state into a series of stack frames called the ***stack***. A ***stack frame*** corresponds to one function call and contains the name of that function, its arguments, and any local variables. You can look at and examine the contents of a stack frame, and you can alter certain parameters of the contents by typing commands to the Debugger.

When it is entered, the Debugger decides which frame it is currently pointing to and displays the name of the associated function and the function's arguments below the signal. For example, calling an undefined function causes the function **symbol-function** to signal an error:

```
> (undefined 4)
>>Error: UNDEFINED has no global function definition

SYMBOL-FUNCTION:                    ; This is the stack frame.
   Required arg 0 (S): UNDEFINED
```

Calling a compiled function with the wrong number of arguments causes the function itself to signal the error:

```
> (eq 3 4 45)
>>Error: EQ cannot be called with 3 arguments
EQ:
   Required arg 0 (X): 3
   Required arg 1 (Y): 4
```

The following code defines and compiles the function **add-one**, which contains no argument type checking:

```
> (compile (defun add-one(arg) (+ 1 arg)))
;;; Compiling function ADD-ONE...assembling...emitting...done.
ADD-ONE
```

If you call **add-one** with a nonnumeric argument, the function **+**, not **add-one**, signals an error:

```
> (add-one 'z)
>>Error: Improper numeric arguments.

+:
   Required arg 0 (X): Z
   Required arg 1 (Y): 1
```

A partial examination of the stack reveals the following:

```
ERROR <- + <- ADD-ONE
```

In this stack frame, the arrows (<-) point toward the top of the stack. The first word, **ERROR**, is the function that called the Debugger. It is at the top of the stack. The arrows can also be thought of as representing the expression "calls the function." Thus, **ADD-ONE** "calls the function" **+**, which "calls the function" **ERROR**.

## Debugger Commands

When an error occurs and the Debugger is entered, it displays this prompt:

```
->
```

After this prompt, you can type any of the expressions that you would type to the top level of Lisp. The Debugger also responds to specialized commands. With these commands you can examine the stack and change the state of your environment.

You can see the complete list of possible commands by typing a question mark (?) to the prompt, as shown below. (Note that you can type Debugger commands in either uppercase or lowercase.)

```
-> ?
:N      Go down one frame (towards the caller).
:N k    Go down k frames.
:N fn   Go down to the next occurrence of the named function.
:P      Go up one frame.
:P k    Go up k frames.
:P fn   Go up to the previous occurrence of the named function.
:>      Go to the bottom of the stack (towards Lisp Top Level).
:> k    Go k frames up from the bottom of the stack.
:> fn   Go to the bottommost occurrence of the named function.
:<      Go to the top of the stack.
:< k    Go k frames down from the top of the stack.
:< fn   Go to the topmost occurrence of the named function.
:A      Abort to Lisp Top Level.
:B      Backtrace to the bottom of the stack.
:B k    Backtrace k frames.
:B fn   Backtrace to the named function.
:BC     Backtrace, showing catches, to the bottom of the stack.
:BC k   Backtrace, showing catches, k frames.
:BC fn  Backtrace, showing catches, to the named function.
:C      Continue from the debugger, at the place it was called.
:D      Display this frame and show options.
:V      Verbosely display this frame, showing up to 32 local variables.
:V k    Verbosely display this frame, showing up to k local variables.
:PP     PPRINT the source code, if any, associated with this frame.
:E      Escape to previous break.
:I k         Inspect local variable numbered k, and set * to result.
:I k   var2  Inspect as above, but also set var2 to result.
:I var       Inspect local variable named var, and set * to result.
:I var var2  Inspect as above, but also set var2 to result.
:L k         Print local variable numbered k, and set * to its value.
:L k   var2  Print as above, but also set var2 to the same value.
:L var       Print local variable named var, and set * to its value.
```

```
:L var var2  Print as above, but also set var2 to the same value.
:S k   exp   Evaluate exp and store result in local variable numbered k.
:S var exp   Evaluate exp and store result in local variable named var.
:R value         Return a value to caller of currently displayed frame.
:R (VALUES ...)  Return multiple values.
:HIDE   fn            Suppress the named function in backtraces.
:HIDE   :PACKAGE pkg  Suppress internal symbols in named package.
:UNHIDE fn            Permit the named function to appear in backtraces.
:UNHIDE :PACKAGE pkg  Permit internal symbols in named package to appear.
:UNHIDE :ALL          Permit everything to appear.
:BUG <filename>       Write a (verbose) bug report to the named file.
:? Describe the commands available (?, :H, :HELP, and HELP also work).
Any other expression will be evaluated and printed.
->
```

If you type the commands **:i**, **:l**, **:s**, **:r**, **:hide**, **:unhide**, and **:bug** with no arguments, you are prompted for those arguments.

If you wish to abort a command that is in progress, type **Control-C** to get a lower level of the Debugger. Then type **:e** at that lower level to return to the Debugger prompt at the level you were in when you decided to abort.

## Exiting the Debugger

Upon entry, the Debugger displays commands for exiting and returning to the top level of Lisp. This list of commands may vary, depending on the recursive level of the program and the type of entry into the Debugger. However, the list always includes the **:a** command:

```
:A      Abort to Lisp Top Level.
```

## Moving in the Stack

The Debugger provides commands for displaying the stack and for moving from one frame to another. The command **:b** displays or executes a backtrace on the stack from the current frame to the bottom of the stack. You can give a numeric argument *k* to some Debugger commands. The command **:b k**, for example, backtraces *k* frames from the current frame. You can also specify a function name as an argument to some Debugger commands. The command **:b fn** performs a backtrace from the current frame to the specified function.

The four frame-moving commands are the following:

- The **:n** command moves toward the bottom of the stack.
- The **:p** command moves toward the top of the stack.

- The :> command moves to the frame at the bottom of the stack.
- The :< command moves to the frame at the top of the stack.

These commands accept both numbers and functions as arguments. A numeric argument to either :n or :p is taken as the number of frames to move in the specified direction. The default is 1. A numeric argument to :> or :< is taken as the number of frames to move away from the specified stack end. Thus, typing the command `:< 3` moves you to the third frame from the top of the stack.

If you specify a function as an argument to these commands, they search for the next occurence of the named function in the specified direction (for :n or :p) or from the specified stack end (for :> or :<) and move to it.

The full session of the `add-one` example is displayed below:

```
> (add-one 'z)
>>Error: Improper numeric arguments.

+:                                 ; This is the contents of the frame
   Required arg 0 (X): Z           ; when the Debugger is entered.
   Required arg 1 (Y): 1

:A    Abort to Lisp Top Level      ; This is an exit command.
-> :b add-one                      ; Backtrace to function add-one.
+ <- ADD-ONE
-> :<                              ; Move to the top of the stack.
ERROR:                             ; Contents of the frame at top of
                                   ; the stack.
   Required arg 0 (FORMAT-STRING): "Improper numeric argument~P."
   Rest arg (FORMAT-ARGS): (2)
-> :b add-one                      ; Backtrace to function add-one.
ERROR <- + <- ADD-ONE
->
```

## Examining Stack Frames

The Debugger commands for examining individual stack frames are as follows:

- The :d command displays the current frame and exit options.
- The :v command prints a long display of the current frame, showing local variables.
- The :pp command pretty-prints the source code, if there is any, associated with the current frame.

The :d command acts as a convenient reminder. The :v command can take a numeric argument specifying the number of local variables to be displayed. The default is 32.

The code that follows demonstrates the use of these commands. It defines a function, bad-fun, that attempts to add an argument to a symbol if the argument is greater than 8:

```
> (defun bad-fun(x) (if (> x 8) (+ x 'q)))
BAD-FUN
> (bad-fun 9)
>>Error: Q should be of type NUMBER

+:
   Rest arg (RESTARG): (9 Q)

:A    Abort to Lisp Top Level
:C    Supply a new value

-> :b                                   ; Backtrace from function +.

+ <- IF <- BLOCK <- BAD-FUN <- EVAL <- unnamed function

-> :n block                             ; Go to the next frame called BLOCK.

BLOCK:
Original code: (BLOCK BAD-FUN (IF (> X 8) (+ X #)))

-> :pp                                  ; PPRINT the associated code.

(BLOCK BAD-FUN
  (IF (> X 8)
      (+ X 'Q)))

-> :n                                   ; Go to the next frame.
BAD-FUN:
Original code: (NAMED-LAMBDA BAD-FUN (X) (BLOCK BAD-FUN (IF # #)))
   Local 0 (X): 9

-> :pp

(NAMED-LAMBDA BAD-FUN                   ; Note the difference between this
              (X)                       ; code and the code associated with
              (BLOCK BAD-FUN            ; the block frame.
                (IF #
                    #)))
```

```
-> :<                                  ; Go to the top of the stack.
CERROR:
   Required arg 0 (CONTINUE-FORMAT-STRING): "Supply a new value"
   Required arg 1 (FORMAT-STRING): "~S should be of type ~A"
   Rest arg (FORMAT-ARGS): (Q NUMBER)

-> :b                                  ; Backtrace.

CERROR <- + <- IF <- BLOCK <- BAD-FUN <- EVAL <- unnamed function

-> :n +                                ; Move to the + frame.
+:
   Rest arg (RESTARG): (9 Q)

-> :v                                  ; Verbose display.
+:
 Rest arg (RESTARG): (9 Q)
 Local 1: 9
 Local 2: Q
 Local 3: (Q)

-> :d                                  ; Display the frame and options.
>>Error: Q should be of type NUMBER

+:
   Rest arg (RESTARG): (9 Q)

:A    Abort to Lisp Top Level
:C    Supply a new value

-> :c
Supply a new value                     ; Debugger message.
Enter a form to be evaluated: 1        ; Set the value of q to 1.
10                                     ; (+ 9 1)
>                                      ; This is the top level.
```

Notice the use of **:c** in the example. Although the exact workings of **:c** vary depending on each situation, its general purpose is to continue evaluation at the top level. In this case, it was determined that a new value was needed for the symbol `Q` before Lisp could continue evaluating the function **bad-fun**.

Notice the use of # when displaying source code. This symbol designates a form that is at a level that is nested beyond the value of the variable ***print-level***. Upon entry, the Debugger binds the variable ***print-level*** to the value of the variable ***debug-print-level***. The value of ***debug-print-level*** defaults to 3. Thus, in the example, the Debugger printed three levels of forms. All forms on the same level are indented equally, as shown below:

```
(NAMED-LAMBDA BAD-FUN
          (X)
          (BLOCK BAD-FUN
            (IF #
                #)))
```

Because ***print-level*** is bound to ***debug-print-level*** upon entry to the Debugger, if you rebind ***debug-print-level*** to a new value, it has no effect on the current recursive level of the Debugger.

The variables ***print-length*** and ***debug-print-length*** are similarly applied to the length in atoms of a particular form. A form whose length exceeds the specified value has its excess atoms represented by periods. If you alter ***print-length*** or ***print-level*** while in the Debugger, you are prompted to restore them to their old values when you exit the Debugger.

## Examining and Modifying Local Variables

The Debugger also supplies commands for examining and modifying a stack frame's local variables:

- The **:i** command inspects a variable.
- The **:l** prints a variable.
- The **:s** command sets the value of a variable.

These commands require a variable name or variable number as their first argument. The **:i** command enters the Inspector, a facility that allows you to inspect data structures (see the chapter "Inspecting Data Structures" for a detailed discussion of the Inspector). The **:l** command prints the value of the given variable. An optional second argument to **:i** or **:l** is taken as a nonlocal variable name, and the result of inspecting or printing is stored in that variable. This is useful when you want to examine data structures without worrying about accidentally altering the contents of the original. The result is also stored in the Lisp history variable *****. Thus, the following examples are synonymous:

```
-> :L 1 MyVar       ; Print the value of local variable number 1 and store
                    ; this value in the global variable MyVar.

-> :L 1             ; Print the value of local variable number 1.
-> (setq MyVar *)   ; Store the value of the previous evaluation in MyVar.
```

The **:s** command requires a Lisp expression as a second argument. The expression is evaluated, and the designated variable is set to the result. If no required arguments are given, all three of these commands prompt for them.

## Using the Debugger

The sample debugging session in this section demonstrates the use of some of the variable examination commands.

The code that follows defines the function `bad-loop`, which returns `t` if its first argument is greater than 3 and its second argument is less than 5; otherwise it loops continually.

```
> (defun bad-loop (x y)
     (do ((i 0 (+ 1 i)))
         (nil)
       (if (and (> x 3) (< y 5)) (return t))))
BAD-LOOP
```

Typing this expression induces a loop:

```
> (bad-loop 3 5)
```

Typing **Control-C** causes a keyboard interrupt that breaks the loop and enters the Debugger:

```
>>Interrupt: Interrupt

IF:
Original code: (IF NIL (RETURN NIL))

:A    Abort to Lisp Top Level
:C    Resume Interrupted Instruction

-> :b                                ; Backtrace the stack.
IF <- PROGN <- TAGBODY <- LET <- BLOCK <- BLOCK <- BAD-LOOP <- EVAL

-> :n let                            ; Move to the LET frame.
LET:
Original code: (LET ((I 0))
                    (TAGBODY #:G4 (IF NIL #) (IF # #) (PSETQ I #) (GO #:G4)))
   Local 0 (I): 1049                 ; Notice that the variable I from
                                     ; the DO loop has grown quite large.
                                     ; The interpreter has expanded the
                                     ; DO loop into a TAGBODY.
```

```
-> :n bad-loop                          ; Move to the BAD-LOOP frame.
BAD-LOOP:
Original code: (NAMED-LAMBDA BAD-LOOP (X Y) (BLOCK BAD-LOOP (DO # # #)))
   Local 0 (X): 3                       ; The variables do not satisfy the
   Local 1 (Y): 5                       ; return condition.  X must be
                                        ; > than 3,and Y must be < than 5.

-> :s 0 4                               ; Set the variable 0 (X) to 4.

-> :l 0 y                               ; Look at the variable 0 and set Y
4                                       ; to the result.

-> y                                    ; Y is set to 4.
4

-> :v                                   ; Look at the variables again.
BAD-LOOP:
Original code: (NAMED-LAMBDA BAD-LOOP (X Y) (BLOCK BAD-LOOP (DO # # #)))
   Local 0 (X): 4
   Local 1 (Y): 5                       ; But Y was set to 4. Remember
                                        ; that the second argument to
                                        ; :L is a global variable, not a
                                        ; local variable, despite the
                                        ; name duplication.

-> :s y 4                               ; Set Y to 4 in an accepted manner.

-> :v                                   ; Now look at the variables.
BAD-LOOP:
Original code: (NAMED-LAMBDA BAD-LOOP (X Y) (BLOCK BAD-LOOP (DO # # #)))
   Local 0 (X): 4
   Local 1 (Y): 4

-> :c                                   ; Continue the interrupted
                                        ; instruction.
Resume Interrupted Instruction
T
>
```

The values of the local variables are now changed, and the loop has been exited.

## *debug-print-length*

**Purpose:** The variable ***debug-print-length*** controls the length of printing in the Debugger.

When the Debugger, the Stepper, or the Inspector is entered, or when the tracing information for a traced function is being printed, the variable ***print-length*** is bound to the value of ***debug-print-length***. If you bind ***debug-print-length*** to **nil** or to a fixnum greater than or equal to 1, ***print-length*** is bound to that value; otherwise it is bound to 10. The default value of ***debug-print-length*** is 10.

**Syntax:** ***debug-print-length*** [*Variable*]

**Remarks:** This variable is an extension to Common Lisp.

**Examples:**

```
> (setq a '(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15))
(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15)
> (break)
>>Break:

EVAL:
   Required arg 0 (EXPRESSION): (BREAK)

:A    Abort to Lisp Top Level
:C    Return from break
-> a
(1 2 3 4 5 6 7 8 9 10 ...)
-> :c
Return from break
NIL
> (dolist (dpl '(nil 3 12 'foo))
    (let ((*debug-print-length* dpl)) (break)))
>>Break:                           ; *debug-print-length* is nil;
                                   ; thus *print-length* is nil.
LET:
Original code: (LET ((*DEBUG-PRINT-LENGTH* DPL)) (BREAK))
   Local 0 (0): NIL

:A    Abort to Lisp Top Level
:C    Return from break
-> a
(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15)
```

```
-> :c
Return from break
>>Break:                        ; *debug-print-length* is 3;
                                ; thus *print-length* is 3.
LET:
Original code: (LET ((*DEBUG-PRINT-LENGTH* DPL)) (BREAK))
   Local 0 (0): 3

:A    Abort to Lisp Top Level
:C    Return from break
-> a
(1 2 3 ...)
-> :c
Return from break
>>Break:                        ; *debug-print-length* is 12;
                                ; thus *print-length* is 12.

LET:
Original code: (LET ((*DEBUG-PRINT-LENGTH* DPL)) (BREAK))
   Local 0 (0): 12

:A    Abort to Lisp Top Level
:C    Return from break
-> a
(1 2 3 4 5 6 7 8 9 10 11 12 ...)
-> :c
Return from break
>>Break:                        ; *debug-print-length* is 'foo;
                                ; thus *print-length* is 10.

LET:
Original code: (LET ((*DEBUG-PRINT-LENGTH* DPL)) (BREAK))
   Local 0 (0): (QUOTE FOO)

:A    Abort to Lisp Top Level
:C    Return from break
-> a
(1 2 3 4 5 6 7 8 9 10 ...)
-> :c
Return from break
NIL
```

## *debug-print-level*

**Purpose:** The variable ***debug-print-level*** controls the level of printing in the Debugger.

When the Debugger, the Stepper, or the Inspector is entered, or when the tracing information for a traced function is being displayed, the variable ***print-level*** is bound to the value of ***debug-print-level***. If you bind ***debug-print-level*** to **nil** or to a fixnum greater than or equal to 1, ***print-level*** is bound to that value. Otherwise ***print-level*** is bound to 3. The default value of ***debug-print-level*** is 3.

**Syntax:** ***debug-print-level*** *[Variable]*

**Remarks:** This variable is an extension to Common Lisp.

**Examples:**

```
> (setq a '(1 (2 (3 (4 (5 (6 (7))))))))
(1 (2 (3 (4 (5 (6 (7)))))))
> (break)
>>Break:

EVAL:
   Required arg 0 (EXPRESSION): (BREAK)

:A    Abort to Lisp Top Level
:C    Return from break
-> a
(1 (2 (3 #)))
-> :c
Return from break
NIL
> (dolist (dpl '(nil 5 'foo))
    (let ((*debug-print-level* dpl)) (break)))
>>Break:                 ; Now *debug-print-level* is bound to nil;
                         ; thus *print-level* is nil.
LET:
Original code: (LET ((*DEBUG-PRINT-LEVEL* DPL)) (BREAK))
   Local 0 (0): NIL

:A    Abort to Lisp Top Level
:C    Return from break
-> a
(1 (2 (3 (4 (5 (6 (7)))))))
```

```
-> :c
Return from break
>>Break:                ; *debug-print-level* is bound to 5;
                        ; thus *print-level* is bound to 5.
LET:
Original code: (LET ((*DEBUG-PRINT-LEVEL* DPL)) (BREAK))
   Local 0 (0): 5

:A    Abort to Lisp Top Level
:C    Return from break
-> a
(1 (2 (3 (4 (5 #)))))
-> :c
Return from break
>>Break:                ; *debug-print-level is bound to 'foo;
                        ; thus *print-level* is bound to 3.
LET:
Original code: (LET ((*DEBUG-PRINT-LEVEL* DPL)) (BREAK))
   Local 0 (0): (QUOTE FOO)

:A    Abort to Lisp Top Level
:C    Return from break
-> a
(1 (2 (3 #)))
-> :c
Return from break
NIL
```

# Chapter 4. Tracing Functions

# Chapter 4. Tracing Functions

## Trace—A Tool for Debugging

The ***Trace Facility*** is a tool for debugging. It allows you to trace one or more functions and provides the ability to perform certain actions at the time a function is called or at the time it exits. It returns as its value a list of names of all functions it has traced. If **trace** is called with no arguments, a list of all functions that are currently being traced is returned. If a function is already being traced, **trace** calls the macro **untrace** before starting the new trace.

Calling **untrace** restores functions to their normal state. The macro may take multiple arguments. Calling it with no arguments untraces all the functions currently being traced.

Forms are evaluated in the lexical environment at the time of the trace call. Special forms cannot be traced because they are neither functions nor macros. Calling **trace** on a macro traces the ***macro expansion***, not the evaluation of the form.

When the tracing information for a traced function is being printed, the variable ***print-level*** is bound to the value of the variable ***debug-print-level***, and the variable ***print-length*** is bound to the value of the variable ***debug-print-length***.

The Common Lisp forms that are part of the Trace Facility are listed below:

| | |
|---|---|
| ***max-trace-indentation*** | ***trace-level*** |
| **trace** | ***trace-new-definitions*** |
| ***trace-arglist*** | ***trace-values*** |
| ***trace-bar-p*** | ***traced-function-list*** |
| ***trace-columns-per-level*** | **untrace** |

## Syntax for Trace

The syntax for **trace** is displayed below:

**trace** {*trace-spec*}* [*Macro*]

*trace-spec*::= *function-name* |
({*function-name* | ( {*function-name*}$^+$ )} {*keyword form*}*)

*keyword* ::= **:cond** |
**:entrycond** |
**:exitcond** |
**:break** |
**:exitbreak** |
**:entry** |
**:exit** |
**:step**

## Keyword Options

The macro **trace** may take the following keyword options:

- **:cond** *form*

  **:entrycond** *form*

  **:exitcond** *form*

  Tracing information is normally printed upon entry and exit. Unless modified by the **:entry** and **:exit** options, the printed information consists of the name of the function followed at entry by its arguments and on exit by its return values. You can control the printing with the keywords listed above. Information is printed upon entry if both the *form* for **:cond** and the *form* for **:entrycond** evaluate to non-**nil**. Information is printed upon exit if both the *form* for **:cond** and the *form* for **:exitcond** evaluate to non-**nil**. In each case, if the keyword is not specified, the default value of *form* is non-**nil**.

- **:break** *form*

  This keyword causes a breakpoint to be entered after printing the entry trace information, but before applying the traced function to its arguments. This occurs if and only if *form* evaluates to non-**nil**.

- **:exitbreak** *form*

  This option works exactly like **:break** except that the breakpoint is entered after the function has been executed and the exit trace information has been printed and before control returns.

- **:entry** ({*form*}$^+$)

  This option prints the values of the forms in the list upon function entry. They are preceded by two backslashes (\\).

- **:exit** ({*form*}$^+$)

  This option prints the values of the forms in the list upon exit from the function. They are preceded by two backslashes (\\).

- **:step** *form*

  This turns on the stepping facility if and only if *form* evaluates to non-**nil**. The stepping facility allows you to step through the execution of the function that is being traced and is explained in more detail in the chapter "Stepping Through an Evaluation."

## *max-trace-indentation*

**Purpose:** In the trace output, the variable ***max-trace-indentation*** represents the maximum number of spaces for indentation. The default is 60.

**Syntax:** ***max-trace-indentation*** [*Variable*]

**Remarks:** This variable is an extension to Common Lisp.

**Examples:**

```
> (defun recur(x) (when (> x 0) (recur (1- x))))
RECUR
> (trace recur)
(RECUR)
> (recur 2)
1 Enter RECUR 2
| 2 Enter RECUR 1
|   3 Enter RECUR 0
|   3 Exit RECUR NIL
| 2 Exit RECUR NIL
1 Exit RECUR NIL
NIL
> (let ((*max-trace-indentation* 2))(recur 2))
1 Enter RECUR 2
| 2 Enter RECUR 1
| 3 Enter RECUR 0
| 3 Exit RECUR NIL
| 2 Exit RECUR NIL
1 Exit RECUR NIL
NIL
```

## trace

**Purpose:** The **trace** macro allows you to trace one or more functions and to perform certain actions at the time a function is called or at the time it returns. It returns as its value a list of names of all the functions it has traced.

**Syntax:** **trace** {*trace-spec*}* [*Macro*]

*trace-spec*::= ***function-name*** |
({***function-name*** | ( {***function-name***}$^+$ )} {***keyword form***}*)

*keyword* ::= **:cond** |
**:entrycond** |
**:exitcond** |
**:break** |
**:exitbreak** |
**:entry** |
**:exit** |
**:step**

**Remarks:** The macro may take multiple arguments. If **trace** is called with no arguments, a list of all functions that are currently being traced is returned.

If a function is already being traced, **trace** calls **untrace** before starting the new trace. Calling **untrace** restores all functions to their normal state.

Calling **trace** on a macro traces the macro expansion, not the evaluation of the form. Special forms cannot be traced because they are neither functions nor macros.

In the trace output, two backslashes (\\) precede the tracing information.

**Examples:**

```
> (progn (untrace)(trace))
NIL
> (trace cdr)
(CDR)
> (trace)
(CDR)
> (cdr '(1 2))
1 Enter CDR (1 2)
1 Exit CDR (2)
(2)
> (trace (cdr :cond nil))
(CDR)
> (cdr '(1 2))
(2)
```

```
> (trace (cdr :entry ("entry banner") :exitcond nil))
(CDR)
> (cdr '(1 2))
1 Enter CDR (1 2)  \\ "entry banner"
(2)
```

**See Also:** **untrace**

## *trace-arglist*

**Purpose:** The variable ***trace-arglist*** provides information on the arguments of the function that is being traced. It is bound to the traced function's argument list and is available for inspection, but you should not change it.

**Syntax:** ***trace-arglist*** [*Variable*]

**Remarks:** This variable is an extension to Common Lisp.

**Examples:**

```
> (defun foo (x) (if (null x) x (foo (cdr x))))
FOO
> (trace (foo :entry (*trace-arglist*)))
(FOO)
> (foo '(1 2 3))
1 Enter FOO (1 2 3)  \\ ((1 2 3))
| 2 Enter FOO (2 3)  \\ ((2 3))
|   3 Enter FOO (3)  \\ ((3))
|   | 4 Enter FOO NIL  \\ (NIL)
|   | 4 Exit FOO NIL
|   3 Exit FOO NIL
| 2 Exit FOO NIL
1 Exit FOO NIL
NIL
```

## *trace-bar-p*

**Purpose:** The variable ***trace-bar-p*** controls whether columns of vertical bars are printed in the Trace Facility's output. Bars are printed if the value is not **nil**; otherwise spaces are printed. The default is t.

**Syntax:** ***trace-bar-p*** [*Variable*]

**Remarks:** This variable is an extension to Common Lisp.

**Examples:**

```
> (defun recur(x) (when (> x 0) (recur (1- x))))
RECUR
> (trace recur)
(RECUR)
> (recur 1)
1 Enter RECUR 1
| 2 Enter RECUR 0
| 2 Exit RECUR NIL
1 Exit RECUR NIL
NIL
> (let ((*trace-bar-p* nil)) (recur 1))
1 Enter RECUR 1
  2 Enter RECUR 0
  2 Exit RECUR NIL
1 Exit RECUR NIL
NIL
```

## *trace-columns-per-level*

**Purpose:** The variable ***trace-columns-per-level*** controls how many columns are added to the Trace Facility's output for each level of function call. The value must be a positive integer, and the default is 2.

**Syntax:** ***trace-columns-per-level*** [*Variable*]

**Remarks:** This variable is an extension to Common Lisp.

**Examples:**

```
> (defun recur(x) (when (> x 0) (recur (1- x))))
RECUR
> (trace recur)
(RECUR)
> (recur 1)
1 Enter RECUR 1
| 2 Enter RECUR 0
| 2 Exit RECUR NIL
1 Exit RECUR NIL
NIL
> (let ((*trace-columns-per-level* 10)) (recur 1))
1 Enter RECUR 1
|         2 Enter RECUR 0
|         2 Exit RECUR NIL
1 Exit RECUR NIL
NIL
```

## *trace-level*

**Purpose:** The variable ***trace-level*** represents how deeply nested a call to the macro **trace** is. The value is an integer.

**Syntax:** ***trace-level*** [*Variable*]

**Remarks:** This variable is an extension to Common Lisp.

**Examples:**

```
> (defun recur2(x) (unless (< x 0) (cons *trace-level* (recur2 (1- x)))))
RECUR2
> (trace recur2)
(RECUR2)
> (recur2 2)
1 Enter RECUR2 2
| 2 Enter RECUR2 1
|   3 Enter RECUR2 0
|   | 4 Enter RECUR2 -1
|   | 4 Exit RECUR2 NIL
|   3 Exit RECUR2 (3)
| 2 Exit RECUR2 (2 3)
1 Exit RECUR2 (1 2 3)
(1 2 3)
```

## *trace-new-definitions*

**Purpose:** The variable ***trace-new-definitions*** traces all new definitions when its value is non-**nil**. Its default value is **nil**, which means that new definitions are not traced.

**Syntax:** ***trace-new-definitions*** [*Variable*]

**Remarks:** This variable is an extension to Common Lisp.

**Examples:**

```
> (defun new1())
NEW1
> (trace)
NIL
> (let((*trace-new-definitions* t)) (defun new2 ()))
NEW2
> (trace)
(NEW2)
```

## *trace-values*

**Purpose:** The variable ***trace-values*** provides information on the returned values of the function that is currently being traced. You can inspect ***trace-values***, but you cannot control the behavior of the macro **trace** with it. It is bound to the traced function's list of returned values.

**Syntax:** ***trace-values*** [*Variable*]

**Remarks:** This variable is an extension to Common Lisp.

**Examples:**

```
> (trace (cdr :exitbreak t))
(CDR)
> (cdr '(1 2 3))
1 Enter CDR (1 2 3)
1 Exit CDR (2 3)
>>Break: Trace exit

(:TRACED CDR):
   Rest arg (G12703): ((1 2 3))

:A    Abort to Lisp Top Level
:C    Return from break
-> *trace-values*
((2 3))
-> (setq *trace-values* '((override-return)))
((OVERRIDE-RETURN))
-> :c
Return from break
(OVERRIDE-RETURN)
```

## *traced-function-list*

**Purpose:** The variable ***traced-function-list*** checks whether a function is being traced. Its value is a list of the names of all the functions that are being traced.

**Syntax:** ***traced-function-list*** [*Variable*]

**Remarks:** This variable is an extension to Common Lisp.

**Examples:**

```
> *traced-function-list*
NIL
> (trace cdr car)
(CDR CAR)
> *traced-function-list*
(CAR CDR)
```

## untrace

**Purpose:** The macro **untrace** restores functions to their normal state. It may take multiple arguments. Calling it with no arguments untraces all the functions currently being traced.

**Syntax:** **untrace** {*function-name*}* [*Macro*]

**Examples:**

```
> (progn (untrace) (trace car caar caaar))
(CAR CAAR CAAAR)
> (untrace car)
(CAR)
> (trace)
(CAAAR CAAR)
> (untrace)
(CAAR CAAAR)
> (trace)
NIL
> (untrace)
NIL
```

**See Also:** **trace**

# Chapter 5. Stepping Through an Evaluation

# Chapter 5. Stepping Through an Evaluation

# The Stepper—A Tool for Debugging

The **step** macro facility, or ***Stepper***, is a tool for examining programs. It allows you to step through the evaluation of a function or form to discover why it is behaving unexpectedly. The **step** macro can be called on any compiled or interpreted function. However, it is most useful when called on interpreted functions, since it only displays the steps involved in the evaluation of interpreted functions.

When **step** is called on a form, the variable ***print-level*** is bound to the value of ***debug-print-level***, and the variable ***print-length*** is bound to the value of ***debug-print-length***.

Before evaluation of any form begins, the form to be evaluated is partially displayed. To illustrate the recursion pattern, the display of each form is indented in proportion to its level of recursion. In the examples, the prompts -> and --> are part of what the Stepper displays. Characters to the right of these prompts are what the user types. After printing, the Stepper waits for a command that will tell it how to proceed.

The Stepper consists of the **step** macro and three variables:

| | |
|---|---|
| ***max-step-indentation*** | ***step-columns-per-level*** |
| **step** | ***step-level*** |

The variables, which are extensions to Common Lisp, allow you to adjust the format of the Stepper's output:

- ***max-step-indentation***

  The variable ***max-step-indentation*** controls indentation of the Stepper's output. With this variable you can set the maximum number of spaces allowed for indentation of the output.

- ***step-columns-per-level***

  The variable ***step-columns-per-level*** controls the number of columns that are added to the output for each level of function call.

- ***step-level***

  The value of the variable ***step-level*** reflects how deeply nested the call to **step** is.

## Using the Stepper

If you type the expression `(step` *form*`)`, the *form* argument is evaluated in step mode. If you type the expression `(step` *fun1 fun2 ... funn*`)`, the Stepper is entered only when one of the functions in the list is called by interpreted code. This is equivalent to typing the expression

`(trace ((`*fun1 ... funn*`) :cond nil :step t))`

## Stepping Commands

After printing, the Stepper waits for a command that will tell it how to proceed. If you type a question mark (?) after stepping through an evaluation, the Stepper displays a list of the basic stepping commands:

```
:n          Evaluate current expression in step mode.
:s          Evaluate current expression without stepping.
:u          Evaluate current expression without stepping
               and go up one level of stepping.
:x          Finish evaluation, but turn Stepper off.
:p          Print current expression.
:pp         Pretty-print current expression.
:b          Enter the Debugger.
:q          Exit to Top Level.
:h or ?     Print this text.
```

## *max-step-indentation*

| | | |
|---|---|---|
| **Purpose:** | The variable ***max-step-indentation*** sets the maximum number of spaces allowed for indentation of the Stepper's output. The default is 60. | |
| **Syntax:** | ***max-step-indentation*** | [*Variable*] |
| **Remarks:** | This variable is an extension to Common Lisp. | |

## step

**Purpose:** The **step** macro is a debugging tool that examines the behavior of programs by stepping through the evaluation of functions.

**Syntax:** **step** *form* | {*function-name*}$^+$ [*Macro*]

**Remarks:** The macro **step** may be called on any compiled or interpreted function.

**Examples:**

```
> (defun f(x) (when x (1+ x)))
F
> (step (f 1))
  (F 1) -> :n                         ; The symbol -> is a prompt
                                      ; printed by the Stepper, and
                                      ; characters to the right of
                                      ; this prompt are entered by
                                      ; the user.
    (FUNCTION F) -> :n
    #<Interpreted-Function (NAMED-LAMBDA F (X) (BLOCK F (WHEN X #)))
50A217>
    1 = 1
    (BLOCK F (WHEN X (1+ X))) -> :s
    2
  2
2
> (step f)
(F)
> (f 1)
  (BLOCK F (WHEN X (1+ X))) -> :n
    (WHEN X (1+ X)) --> :n            ; The symbol--> is a prompt
                                      ; printed by the Stepper, and
                                      ; characters to the right of
                                      ; this prompt are entered by
                                      ; the user.  Unlike ->, -->
                                      ; indicates that the displayed
                                      ; form is a macro expansion.
      X = 1
      (1+ X) -> :u
      2
    2
  2
2
```

## *step-columns-per-level*

**Purpose:** The variable ***step-columns-per-level*** controls how many columns are added to the Stepper's output for each level of function call. The value is an integer, and the default is 2.

**Syntax:** ***step-columns-per-level*** [*Variable*]

**Remarks:** This variable is an extension to Common Lisp.

## *step-level*

**Purpose:** The variable ***step-level*** represents how deeply nested the call to the macro **step** is. The value is an integer.

**Syntax:** ***step-level*** [*Variable*]

**Remarks:** This variable is an extension to Common Lisp.

**Examples:**

```
> *step-level*
0
```

# Chapter 6. Inspecting Data Structures

# Chapter 6. Inspecting Data Structures

## About the Inspector

The *Inspector* facility allows you to inspect data structures. The command level of the Inspector is a read-eval-inspect loop, similar to the top-level read-eval-print loop. When the function **inspect** is called on a particular object, it displays the components of that object, and the object can then be modified. The components displayed depend on the kind of object.

## Summary of Inspector Commands

The Inspector is entered by calling the function **inspect**. This is the syntax:

**inspect** *object* [*Function*]

When **inspect** has been called on an object, typing a question mark (?) at the prompt (>>) returns a list of possible commands, which is displayed on the screen as shown below. You may type the commands in either uppercase or lowercase.

```
>> ?
:Q      Quit the inspector, returning the current object.
:R      Redisplay the current object.
:U      Pop the inspector stack.
:S N EXP
        Evaluate EXP and store the result in slot N of the current object.
:L N    Set *PRINT-LENGTH* for the inspector to N (possibly NIL).
:D N    Set *PRINT-LEVEL* for the inspector to N (possibly NIL).
?, :?, :H, :HELP
        Print this message.


A positive integer selects a component of the current object to be
inspected, pushing the current object on the inspector stack.  Any
other input is evaluated and inspected, also pushing the current object
on the stack.  The variable * can be used to refer to the current object
in an expression to be evaluated.
>>
```

## Sample Terminal Session

```
> (defstruct (foo) a b (c 0))                 ; Define a Lisp structure,
FOO                                           ; foo, with constructor
                                              ; make-foo and slots a, b,
                                              ; and c, with slot c
                                              ; initialized to 0.
> (defstruct (baz) x y z)                     ; Define a Lisp structure,
BAZ                                           ; baz, with constructor
                                              ; make-baz and slots x, y, z.
> (setq aa (make-foo))
#S(FOO A NIL B NIL C 0)
> (inspect aa)                                ; The Inspector displays the
#<Structure FOO 3D9293>                       ; structure aa with its slots
                                              ; numbered for reference.
[0: A] NIL
[1: B] NIL
[2: C] 0


>> 44
44 is out of range as an index for this item.
>> 2
0

>> :u                                         ; Pop to previous level.
#<Structure FOO 3D9293>


[0: A] NIL
[1: B] NIL
[2: C] 0
>> (setq bb (make-baz :x 11 :y (list 22 33))); This input is evaluated
#<Structure BAZ 3DC3A3>                       ; and then inspected.


[0: X] 11
[1: Y] (22 33)
[2: Z] NIL
>> 1
#<List 3DCC99>
[0] 22
[1] 33
>> :q
(22 33)
```

**Note:** The value returned from the Inspector is that of the last object examined.

## inspect

**Purpose:** The function **inspect** is used for examining data structures. When called on an object, **inspect** displays the object's components. The object can then be modified.

**Syntax:** **inspect** *object* [*Function*]

**Example:**

```
> (setq a 1)
1
> (inspect a)
1

>> :q
1
> (inspect 'a)
#<Symbol 3C81FD>

[0: NAME] "A"
[1: VALUE] 1
[2: FUNCTION] Undefined
[3: PLIST] NIL
[4: PACKAGE] #<Package "USER" 39DD0B>
>> :q
A
```

# Chapter 7. The Foreign Function Interface

# Chapter 7. The Foreign Function Interface

# Introduction to the Foreign Function Interface

Sun Common Lisp provides a ***foreign function interface*** that allows you to link compiled C and FORTRAN files with a Lisp program and to link Lisp programs into running C code. With this interface, you can create Lisp streams and extract file descriptors associated with Lisp streams so that you can send input to or receive output from foreign files. The foreign function interface also lets you make operating system calls from within Lisp.

All of the functions, macros, and variables in the foreign function interface are extensions to Common Lisp. See the function pages at the end of this chapter for complete syntactic descriptions of these extensions.

## Defining and Calling Foreign Functions

The following macros and functions allow you to define and locate functions and symbols that can be used in the foreign function interface:

| | |
|---|---|
| **define-c-callable** | **define-fortran-function** |
| **define-c-function** | **foreign-address-of** |
| **define-foreign-symbol** | **register-lisp-function** |

The macros **define-c-function** and **define-fortran-function** create a Lisp function that calls a foreign function. Lisp arguments are converted to the appropriate foreign data type before the foreign function call. The value returned by the foreign function is then converted to a Lisp data type.

The macro **define-c-callable** defines a Lisp function that can be called from C code. The C arguments are converted to Lisp data types, and the value returned by the function is coerced to a C data type and returned to the caller.

The macro **define-foreign-symbol** prepares its argument to be bound to the value of a foreign symbol. This allows Lisp functions that have been created with **define-c-function** and **define-fortran-function** to access symbols from compiled foreign language functions.

The function **foreign-address-of** accepts a Lisp symbol that has been defined with **define-c-function** or **define-fortran-function** and returns the integer starting address of the associated foreign function. The starting address allows you to pass a foreign procedure as an argument to another foreign function. The function **register-lisp-function** accepts the name of a Lisp function that has been defined with **define-c-callable** and returns its integer starting address. This address allows you to pass a Lisp function as an argument to a C procedure.

## Loading Foreign Language Files

The following variable and functions allow you to load foreign functions and symbols that have been defined in the foreign function interface:

*foreign-temporary-directory*
**load-foreign-files**
**load-foreign-libraries**

The function **load-foreign-files** loads foreign language compiled files into the running Lisp environment. The function **load-foreign-libraries** loads selected functions from foreign language library files. Files have **.o** extensions, and libraries have **.a** extensions.

When invoked, both **load-foreign-files** and **load-foreign-libraries** automatically load all functions and symbols that have been defined with **define-c-function**, **define-fortran-function**, and **define-foreign-symbol** since the last load call.

The variable *foreign-temporary-directory* specifies the directory in which temporary files are created during the loading of foreign files.

## Making Lisp Streams from File Descriptors

The following functions allow you to perform input and output operations from within the foreign function interface:

**extract-stream-handles**
**make-lisp-stream**

The function **make-lisp-stream** creates a Common Lisp stream from a UNIX file descriptor. The function **extract-stream-handles** returns the UNIX input and output file descriptors for a given Common Lisp stream.

## Calling System Functions

The function **syscall** uses a UNIX system call number to call system functions.

**syscall**

## Data Types Passed to Foreign Functions

The data types listed in the tables that follow may be passed to foreign functions. All Common Lisp simple strings are stored with a terminating null byte so that they may be passed to C functions without difficulty. This terminating null byte is not included in the length of the string returned by the **length** function.

For functions that are defined with **define-fortran-function**, arguments of the data type **integer** or **float** are copied and passed by reference. If the FORTRAN subroutine modifies the argument, the modification is not reflected in the Lisp calling program. If you need a modifiable integer or float argument to use with a FORTRAN program, use a single-element array of type **(signed-byte 32)** or type **single-float**. Any string argument passed to FORTRAN functions must be a **simple string**; that is, the string argument must be a simple array that is not displaced and does not contain fill pointers.

Before you call a given foreign function for the first time, you must invoke either **define-c-function** or **define-fortran-function** to make the foreign function available in Lisp. You must also load the foreign function with **load-foreign-files** or **load-foreign-libraries**.

| **Lisp** | **C** |
|---|---|
| **integer** (32 bits or less) | **long** |
| **character** | **long** |
| **float** | **double** |
| **(array (unsigned-byte 8))** | **unsigned char** [ ] |
| **(array (signed-byte 8))** | **char** [ ] |
| **(array (unsigned-byte 16))** | **unsigned short** [ ] |
| **(array (signed-byte 16))** | **short** [ ] |
| **(array (unsigned-byte 32))** | **unsigned long** [ ] |
| **(array (signed-byte 32))** | **long** [ ] |
| **(array string-char)** | **char** [ ] |
| **(array single-float)** | **float** [ ] |
| **boolean (t or nil)** | **boolean (1 or 0)** |

**Figure 7–1. Table of Data Types for C Programs**

| **Lisp** | **FORTRAN** |
|---|---|
| integer (32 bits or less) | integer*4 |
| character | integer*4 |
| float | real*4 |
| (array (unsigned-byte 8)) | logical*1 array |
| (array (signed-byte 8)) | logical*1 array |
| (array (unsigned-byte 16)) | integer*2 array |
| (array (signed-byte 16)) | integer*2 array |
| (array (unsigned-byte 32)) | logical*4 array |
| (array (signed-byte 32)) | integer*4 array |
| (array string-char) | character*n |
| (array single-float) | real*4 array |

**Figure 7–2. Table of Data Types for FORTRAN Programs**

## define-c-callable

**Purpose:** The macro **define-c-callable** defines a Lisp function that can be called from a running C program.

**Syntax:** **define-c-callable** *name arglist* {*form*}* [*Macro*]

**Remarks:** The ***name*** argument must be a symbol; it is not evaluated. The name of the Lisp function is returned as the value of the macro.

The ***arglist*** argument is a list of lists of the form (***symbol type***), where ***symbol*** is the name of an argument and ***type*** is a keyword that specifies the data type of the argument. The ***type*** element must be one of the following:

- **:integer**

  This keyword indicates that a C argument of type **int** will be passed to the Lisp function.

- **:string**

  This keyword indicates that a C string argument with a terminating null byte will be passed to the Lisp function. The ***symbol*** argument is bound to a Lisp string that contains a copy of the string passed by the C procedure. Modifications made to the copied string are not made to the C string that is passed.

- **:coerce-double-to-single**

  This keyword indicates that a C argument of type **double** will be passed to the Lisp function; all floating-point arguments that C passes to procedures must be in double-float format. The double-float argument is coerced to a Lisp single-float argument.

- **:pointer**

  This keyword indicates that a pointer to an arbitrary C object will be passed to the Lisp function.

The value returned by the Lisp function is converted to the appropriate C data type before it is returned to the calling procedure. Any of the types listed in the "Table of Data Types for C Programs" is valid except for the type **boolean**. If the Lisp function attempts to return a data object that does not have one of the types listed in the table, the value returned is 0.

The body of the function consists of the forms specified by the *form* arguments; these forms are executed in order when the function is called.

## define-c-callable

This macro is an extension to Common Lisp.

Examples:

```
; This example assumes that the file callmeback.o has been compiled from
; the following C source code with cc -c callmeback.c.
;
;    int callmeback(lproc)
;    int (*lproc)();
;    {
;      (*lproc)(1234,1.234,"Bazola");
;    }
;

> (define-c-function callmeback (fn))     ; Define a C function with
CALLMEBACK                                ; one argument.
> (load-foreign-files '("callmeback.o"))  ; Load the C code above.
NIL
> (define-c-callable anytime              ; Define the function to be
    ((x :integer)                         ; called.
     (y :coerce-double-to-single)
     (z :string))
    (format t "~&Called with ~S, ~S, ~S.~%" x y z)
    123)                                  ; 123 is the value to be
ANYTIME                                   ; returned.
> (callmeback (register-lisp-function 'anytime))
Called with 1234, 1.234, "Bazola".
123
```

## define-c-function, define-fortran-function

**Purpose:** The macros **define-c-function** and **define-fortran-function** create a Lisp function that calls a foreign function. Lisp arguments are converted to the appropriate foreign data type before the foreign function call. The value returned by the foreign function is then converted to a Lisp data type.

**Syntax:** **define-c-function** *function-name arglist* [*Macro*]
`&key :result-type`

**define-fortran-function** *function-name arglist* [*Macro*]
`&key :result-type`

**Remarks:** The *function-name* argument to **define-c-function** and **define-fortran-function** associates a Lisp function name with an appropriate foreign function name. Once the association has been specified, the resulting Lisp function can be used to call the foreign function from within Lisp. The *function-name* argument can be one of the following:

- a list of the form (*symbol string*)

  In the list (*symbol string*), the *symbol* element stands for the Lisp function you are defining, and the *string* element is the name of the foreign function. You would use this form of list as the *function-name* argument when the Lisp function name cannot be directly associated with the appropriate foreign function name, either because the foreign function name contains letters in both uppercase and lowercase or because the foreign function name is entirely different from the Lisp function name.

- a symbol

  You would use a symbol as the *function-name* argument when the Lisp function name can be directly associated with the foreign function name. The Lisp function name is lowercased, any dashes in the name are converted to underbar characters, and either an underbar prefix or suffix is added if appropriate.

Before you call a given foreign function for the first time, you must call either **define-c-function** or **define-fortran-function** to make the appropriate foreign function available in Lisp. You must also call **load-foreign-files** or **load-foreign-libraries** to load all the functions that have been defined with **define-c-function** or **define-fortran-function** since the last load.

The *arglist* argument to **define-c-function** and **define-fortran-function** is a list of the arguments accepted by the foreign function. The argument list differs from a Common Lisp lambda list in the following ways:

- keywords

  The **&optional** lambda list keyword and the **&rest** lambda list keyword are allowed, but the **&key** lambda list keyword and associated keyword arguments are not allowed.

- *supplied-p-parameter* variable

  A *supplied-p-parameter* variable to the **&optional** lambda list keyword is not allowed.

- type-checking syntax

  When the name of an argument appears in the argument list, it can be replaced by a list of the form (*name type*), where the element *type* is a keyword specifying what kind of argument is to be passed to the function. Possible argument types include all of the types that the keyword **:result-type** accepts, as well as the keyword **:string**. The keyword **:string** forces a simple string to be passed as a C string. If a function defined by **define-c-function** or **define-fortran-function** is called with an argument that does not match the given type, an error is signaled.

The **:result-type** keyword option to **define-c-function** and **define-fortran-function** specifies the type of result to be returned from the foreign function. The possible values of this option are as follows:

- **:procedure**

  This keyword indicates that the foreign function returns no value (a C function of type **void**, for example).

- **:boolean**

  This keyword indicates that the foreign function returns true (non-0) or false (0). Lisp converts these values to **t** and **nil** respectively.

- **:fixnum**

  This keyword indicates that the foreign function returns a signed 30-bit integer. This is a special type that is provided because returning a **:fixnum** value does not allocate new storage.

- **:integer**

  This keyword indicates that the foreign function returns a signed 32-bit integer.

- **:uinteger**

  This keyword indicates that the foreign function returns an unsigned 32-bit integer.

- **:pointer**

  This keyword indicates that the foreign function returns a foreign pointer. Foreign pointers returned with the **:pointer** keyword can only be used as arguments to other foreign function calls.

- **:single**

  For **define-fortran-function**, this keyword indicates that the FORTRAN function returns a floating-point number in single-float format; the Lisp function also returns the number in single-float format. For **define-c-function**, this keyword is not supported because C routines cannot return floating-point arguments in single-float format.

- **:coerce-double-to-single**

  This keyword indicates that the foreign function returns a foreign floating-point number in double-float format, which Lisp converts to single-float format.

These macros are extensions to Common Lisp.

**Examples:**

```
; This example defines a function that calls the C library "printf"
; function with a control string and some arguments to print.

> (define-c-function printf ((ctl-string :string) &rest args))
PRINTF
> (load-foreign-libraries '("_printf"))      ; Actually load the code.
"/tmp/Lu00268050.o"
> (printf "It's %d %s" 3 "in the morning.")  ; Print a string.
21
> (printf "                                  ; Print a newline character
")                                           ; so that the string will
                                             ; appear.  Until you enter
                                             ; this command, the string
                                             ; is kept in a buffer used
                                             ; by printf.
It's 3 in the morning.
```

## define-foreign-symbol

**Purpose:** The macro **define-foreign-symbol** arranges for a Lisp symbol to be bound to the value of a symbol from a compiled foreign language file.

**Syntax:** **define-foreign-symbol** *symbol-name* [*Macro*]

**Remarks:** The *symbol-name* argument to **define-foreign-symbol** associates a Lisp symbol with an appropriate foreign symbol. Once the association has been specified, the resulting Lisp symbol can be used to access the foreign symbol from within Lisp. The *symbol-name* argument can be one of the following:

- a list of the form (*symbol string*)

  In the list (*symbol string*), the *symbol* element stands for the Lisp symbol you are defining, and the *string* element is the name of the foreign symbol. You would use this form of list as the *symbol-name* argument when the Lisp symbol name cannot be directly associated with the appropriate foreign symbol name, either because the foreign symbol name contains letters in both uppercase and lowercase or because the foreign symbol name is entirely different from the Lisp symbol name.

- a symbol

  You would use a symbol as the *symbol-name* argument when the Lisp symbol name can be directly associated with the foreign symbol name.

The Lisp symbol is not bound until you have defined it with **define-foreign-symbol** and have loaded it with either **load-foreign-files** or **load-foreign-libraries**.

This macro is an extension to Common Lisp.

## extract-stream-handles

**Purpose:** The function **extract-stream-handles** returns the UNIX file descriptors that the given Common Lisp stream uses for input and output.

**Syntax:** **extract-stream-handles** *common-lisp-stream* [*Function*]

**Remarks:** The function always returns two values. For unidirectional streams, one of the returned values is **nil**, which indicates that there is no associated UNIX file descriptor.

**Note:** You should not perform input or output operations that involve both the file descriptors and the Common Lisp stream associated with them. Doing so may cause unpredictable results.

This function is an extension to Common Lisp.

## foreign-address-of

**Purpose:** The function **foreign-address-of** accepts a Lisp symbol that has been defined with **define-c-function** or **define-fortran-function** and returns as an integer the starting address of the given foreign function.

**Syntax:** **foreign-address-of** *function-name* [*Function*]

**Remarks:** You would use such a starting address to pass a foreign function as an argument to another foreign function.

This function is an extension to Common Lisp.

## *foreign-temporary-directory*

**Purpose:** The variable ***foreign-temporary-directory*** specifies the directory in which temporary files are created during the loading of foreign files. The default directory is `"/tmp"`.

**Syntax:** ***foreign-temporary-directory*** [*Variable*]

**Remarks:** The temporary files are named `Lu`*n*`.o`, where *n* is a unique number constructed from the UNIX process identifier. These files are required for successful execution of the function **disksave** and for subsequent loadings of foreign files.

**Note:** You should not delete the `Lu`*n*`.o` files from the temporary directory while you are in the Lisp environment. The temporary files are deleted automatically when you exit Lisp with a call to the function **quit**. However, if the Lisp environment is terminated abnormally, the `Lu`*n*`.o` files sometimes are not deleted automatically. In that case you may delete them with no adverse side effects.

This variable is an extension to Common Lisp.

## load-foreign-files

Purpose: The function **load-foreign-files** loads foreign language compiled (.o) files into Lisp.

Syntax: **load-foreign-files** *files* `&optional` *libraries* [*Function*]

Remarks: The *files* argument is a list of strings that name the files to be loaded.

The default value for the *libraries* argument is the list (`"-lc"`); that is, the loader automatically searches the C run-time library. If you wish to search other libraries as well as the C run-time library, you must include `"-lc"` at the end of the list.

If you load FORTRAN code, you should include the FORTRAN libraries `"-lF77"`, `"-lI77"`, and `"-lU77"` along with the C library `"-lc"`.

When you invoke this function, it loads all the functions and symbols that you have defined with **define-c-function**, **define-fortran-function**, or **define-foreign-symbol** since the last load.

This function is an extension to Common Lisp.

Examples:

```
; This example assumes that the file for-eg-ex.o has been compiled from
; the following C source code with cc -c for-eg-ex.c.
;
;    int cfunc(s)
;    char s[];
;    {
;      strcpy (s, "Hello");
;      return(4);
;    }

> (define-c-function cfunc(s) :result-type :integer)
CFUNC
> (load-foreign-files '("for-eg-ex.o"))
T
> (setq s ".............")
"............."
> (cfunc s)
4
> s
"Hello........"
```

## load-foreign-libraries

**Purpose:** The function **load-foreign-libraries** loads functions from foreign language compiled library files (.a) into Lisp.

**Syntax:** **load-foreign-libraries** *symbols* `&optional` *libraries* [*Function*]

**Remarks:** The *symbols* argument is a list of Lisp strings that name the foreign functions as they appear in a foreign object file: `"_fn"` for C functions named `fn`, and `"fn_"` for FORTRAN functions named `fn`. Such functions are then called from Lisp with the name `fn`. Symbols that have been defined by **define-c-function** or **define-fortran-function** since the last load are loaded automatically and should not be specified here. The *symbols* argument is provided for backward compatibility with **foreign-load-libraries**; it should normally be **nil**.

The default value for the *libraries* argument is the list (`"-lc"`); that is, the loader automatically searches the C run-time library. If you wish to search other libraries as well as the C run-time library, you must include `"-lc"` at the end of the list.

If you load FORTRAN code, you should include the FORTRAN libraries `"-lF77"`, `"-lI77"`, and `"-lU77"` along with the C library `"-lc"`.

When you invoke this function, it loads all the functions and symbols that you have defined with **define-c-function**, **define-fortran-function**, or **define-foreign-symbol** since the last load.

This function is an extension to Common Lisp.

**Examples:**

```
; hypot is a C math library routine that computes the hypotenuse of
; a triangle whose other sides are given by two floating-point numbers.
; mktemp is a routine that makes a unique filename from its string
; argument.

> (define-c-function mktemp (name) :result-type :pointer)
MKTEMP
> (define-c-function hypot (x y) :result-type :single)
HYPOT
> (load-foreign-libraries '("_mktemp" "_hypot") '("-lm"))
T
> (hypot 23.6 12.2)
26.566896
```

```
> (setq s "abcXXXXXX")
"abcXXXXXX"
> (mktemp s)
1344269752
> s
"abca00305"
```

## make-lisp-stream

**Purpose:** The function **make-lisp-stream** constructs a Common Lisp stream from UNIX file descriptors.

**Syntax:** **make-lisp-stream** `&key :input-handle :output-handle :element-type :stream-type :name` [*Function*]

**Remarks:** If the Common Lisp stream you construct is used for input, the **:input-handle** keyword argument must be a UNIX file descriptor for a device that is opened for input. If the Common Lisp stream is used for output, the **:output-handle** keyword argument must be a UNIX file descriptor for a device that is opened for output. You may construct bidirectional Common Lisp streams either from two separate file descriptors or from a single file descriptor for a device that is opened for both input and output.

The **:element-type** keyword argument gives the element type to be associated with the stream; it is specified as if for the Common Lisp function **open**. The default value is `'string-char`.

The **:stream-type** keyword option indicates whether seeking is easy, difficult, or impossible on the device designated by the UNIX file descriptor that is associated with the stream. The possible values of this option are as follows:

- **:random**

  This keyword indicates that the device designated by the file descriptor has random access and that seeking is easy. This value is the default.

- **:sequential**

  This keyword indicates that the device designated by the file descriptor has sequential access and that seeking is difficult.

- **:ephemeral**

  This keyword indicates that seeking is impossible on the device designated by this file descriptor, as it is with pipes or network connections.

The **:name** keyword argument is a filename that is associated with the constructed stream. It defaults to `"Unknown_file"`.

This function is an extension to Common Lisp.

## register-lisp-function

**Purpose:** The function **register-lisp-function** accepts a foreign symbol that has been defined with **define-c-callable** and returns as an integer the starting address of the associated Lisp function.

**Syntax:** **register-lisp-function** *function-name* [*Function*]

**Remarks:** You would use such a starting address to pass a Lisp function as an argument to a foreign function.

This function is an extension to Common Lisp.

## syscall

**Purpose:** The function **syscall** calls system functions.

**Syntax:** **syscall** *system-call-number* `&rest` *arguments* [*Function*]

**Remarks:** The *system-call-number* argument gives the number of a UNIX system call, as defined in `/usr/include/syscall.h` (see the *UNIX Programmer's Reference Manual*).

The remaining arguments are those that will be passed to the foreign function that has been called. Each argument is a Lisp object reference that is converted to a form suitable for a C function. A maximum of 64 arguments may be passed to a foreign function.

The function **syscall** returns two values—the results returned by the system in the registers D0 and D1.

This function is an extension to Common Lisp.

**Examples:**

```
; 142 is the system call number for the function "gethostid," as defined
; in /usr/include/syscall.h.

> (syscall 142)
16782277
0
```

# Chapter 8. Running UNIX Programs from Lisp

# Chapter 8. Running UNIX Programs from Lisp

## Introduction to Running UNIX Programs

The function **run-unix-program**, which is an extension to Common Lisp, provides the ability to run other UNIX programs (that is, programs that can be called from the Shell) from the Lisp environment. Its syntax is as follows:

**run-unix-program** *name* &key `:input :output` [*Function*]
`:error-output`
`:wait :arguments`
`:if-input-does-not-exist`
`:if-output-exists`
`:if-error-output-exists`

The *name* argument is a pathname or an object that can be coerced to a pathname. It represents the name of the program to be run. If the pathname is a relative pathname, each directory in the environment variable **PATH** is searched for the filename that corresponds to *name*. If the pathname is an absolute pathname, that is, if the name begins with a slash (/), then that file is used. The namestring of *name* is the **argv**[0] parameter for the program.

Four values are returned by **run-unix-program**.

1. The first value is a stream. If either the **:input** or the **:output** keyword argument is **:stream**, that stream communicates with the running process and is the first value returned. If neither keyword is **:stream**, the first value is **nil**.
2. If the **:error-output** keyword argument is **:stream**, the second value returned is the resulting input stream from which Lisp can read the program's error output. If **:error-output** is not **:stream**, the second value returned is **nil**.
3. If the **:wait** keyword argument is **t**, the third value is the exit status of the program that was run. Otherwise the exit status is **nil**.
4. If the program is running, the fourth value is its UNIX process id. If the program has run to completion, the fourth value is **nil**.

## Keyword Options

You can specify a number of keyword arguments to **run-unix-program**, as the syntactic description shows. The keyword arguments **:input**, **:output**, and **:error-output** determine where the program gets and where it sends its standard input, standard output, and error output respectively.

- **:input**
  - **nil**

    If the **:input** keyword argument is **nil**, the standard input for the program is taken from the standard input for Lisp, which is normally input from the terminal. This is analogous to typing a simple command to the Shell with no redirection. The default value for **:input** is **nil**.
  - a filename

    If the **:input** keyword argument is a filename, the standard input for the program is read from the file. This is analogous to using the symbol < with a Shell command. If the file does not exist, what happens depends on the value of the argument **:if-input-does-not-exist**.
    - **:error**

      If **:if-input-does-not-exist** is **:error**, an error is signaled.
    - **:create**

      If **:if-input-does-not-exist** is **:create**, an empty file to read from is created and given the specified filename.
    - **nil**

      If **:if-input-does-not-exist** is **nil**, **run-unix-program** returns **nil** without taking any action.
  - **:stream**

    If the **:input** keyword argument is **:stream**, the standard input for the program is read from Lisp through a newly created output stream that is returned as the first value of **run-unix-program**. Anything written to that stream by Lisp is read by the program. When the stream is closed, the program reads an end-of-file indicator. If for some reason the program terminates, writing to that stream notifies Lisp with a broken pipe signal.
  - a stream

    If the **:input** keyword argument is a stream, the standard input for the program is read from that Common Lisp stream. The stream must be an input stream open to a file or another process. The stream must not be given a string stream, synonym stream, or other higher-level Lisp stream.

    The stream must have been created by a call to the function **open** or by a call to the function **run-unix-program**. (Note that the **with-open-file** macro does an implicit **open**; that is, if **with-open-file** is called, a stream is created.)

    You can make pipes by taking the output from one program and feeding it to another. First, make an input stream by calling **run-unix-program** with **:output :stream**. Then call **run-unix-program**, using that stream as the **:input** argument.

When a Lisp stream is passed to a subprogram, that stream can no longer be written to or read from by Lisp. It is effectively closed.

- :output
  - **nil**

    If the **:output** keyword argument is **nil**, the standard output of the program goes to the standard output of Lisp, which is normally the terminal. This is analogous to typing a simple command to the Shell and specifying no redirection. The default value for **:output** is **nil**.

  - a filename

    If the **:output** keyword argument is a filename, the standard output of the program will be written to the specified file. This is analogous to using the symbol > with a Shell command. If the file already exists, then the action taken depends on the **:if-output-exists** argument.

    — **:error**

    If **:if-output-exists** is **:error** or is omitted, an error is signaled.

    — **:append**

    If **:if-output-exists** is **:append**, standard output from the program is appended to the end of the file.

    — **:supersede**

    If **:if-output-exists** is **:supersede**, the file is replaced by the standard output from the program.

    — **nil**

    If **:if-output-exists** is **nil**, then **run-unix-program** returns **nil** without taking any other action.

  - **:stream**

    If the **:output** keyword argument is **:stream**, the standard output of the program goes to a newly created input stream that is returned as the first value of **run-unix-program**. Lisp can read from this stream to get the program's standard output.

  - a stream

    If the **:output** keyword argument is a stream, the standard output of the program is written to the stream. The stream must be an output stream. The same restrictions that apply to a stream specified as **:input** apply to one specified as **:output**. Once a Lisp stream is passed to a subprogram, the stream can no longer be written to or read from by Lisp. It is effectively closed.

- **:error-output**

  The **:error-output** keyword argument specifies where error output from the program is to go. Its possible values and their meanings are the same as those of the **:output** keyword argument, but an additional possible value for **:error-output** of **:output** allows error output to be merged with the program's standard output. Also, the action taken when a specified error output file already exists is determined by the **:if-error-output-exists** argument, rather than by the **:if-output-exists** argument.

  - **nil**

    If the **:error-output** keyword argument is **nil**, the standard output of the program goes to the standard output of Lisp, which is normally the terminal. This is analogous to typing a simple command to the Shell and specifying no redirection. The default value for **:error-output** is **nil.**

  - a filename

    If the **:error-output** keyword argument is a filename, the standard output of the program will be written to the specified file. If the file already exists, then the action taken depends on the **:if-error-output-exists** argument.

    - **:error**

      If **:if-error-output-exists** is **:error** or is omitted, an error is signaled.

    - **:append**

      If **:if-error-output-exists** is **:append**, error output from the program is appended to the end of the file.

    - **:supersede**

      If **:if-error-output-exists** is **:supersede**, the file is replaced by the error output from the program.

    - **nil**

      If **:if-error-output-exists** is **nil**, then **run-unix-program** returns **nil** without taking any other action.

  - **:stream**

    If the **:error-output** keyword argument is **:stream**, an input stream is created and returned as the second value of **run-unix-program**. Lisp can then read the error output of the program from this stream.

  - a stream

    If the **:error-output** keyword argument is a stream, the error output of the program is written to the stream. The stream must be an output stream. The same restrictions that apply to a stream specified as **:input** apply to one specified as **:error-output**. Once a Lisp stream is passed to a subprogram, the stream can no longer be written to or read from by Lisp. It is effectively closed.

  - **:output**

    If the **:error-output** keyword argument is **:output**, the error output for the program is merged with the standard output. Both are sent to the location indicated by the **:output** argument.

- **:wait**

  The keyword argument **:wait** determines whether or not Lisp waits for the program to complete. Its default value is **t**.

  - If **:wait** is **t** or is omitted, Lisp does not begin running until the program has completed.
  - If **:wait** is **nil**, Lisp continues to run in parallel with the program.
  - If **:wait** is **t** and if **:input**, **:output**, or **:error-output** is **:stream**, an error is signaled. Waiting and having a stream open to a process can lead to deadlock; thus these combinations are not permitted.

- **:arguments**

  The keyword **:arguments** must be a list of strings that are the normal Shell arguments to the program. Together with the *name* argument, it determines the **argv** and **argc** parameters of the program.

The keyword arguments **:if-input-does-not-exist**, **:if-output-exists**, and **:if-error-output-exists** are significant only if **:input**, **:output**, or **:error-output** is a filename. The default value of each of the three arguments is **:error**.

- **:if-input-does-not-exist**

  The keyword **:if-input-does-not-exist** can be one of the following:

  - **:error**, which is the default, signals an error.
  - **:create** creates an empty file.
  - **nil** returns **nil** without doing anything if the file mentioned does not exist.

- **:if-output-exists**

  **:if-error-output-exists**

  The keyword arguments **:if-output-exists** and **:if-error-output-exists** can be one of the following:

  - **:error**, which is the default, signals an error.
  - **:append** appends output to the file.
  - **:supersede** replaces the file with the new output.
  - **nil** returns **nil** without doing anything if the file mentioned already exists.

## Annotated Examples

The three examples that follow illustrate various uses of **run-unix-program**.

### Example 1—Running a Program

This example shows how to run a program and have Lisp communicate with it through a stream. Lisp does not wait for the program to complete. This example assumes you have a program called "banner" on /usr/games.

```
> (defun type-until-hung (stream)          ; This function prints the
    (do ((char (read-char-no-hang stream)  ; output from a program
               (read-char-no-hang stream))) ; that has been started
        ((null char))                      ; with run-unix-program.
      (write-char char)))                  ; On a heavily loaded system,
                                           ; pauses in program output
                                           ; may cause premature
                                           ; termination of this function.
                                           ; In this case the function
                                           ; may simply be called again
                                           ; to get the rest of the
                                           ; output.
TYPE-UNTIL-HUNG

> (setq shell (run-unix-program "csh" :input :stream :output :stream
                                      :wait nil))
#<Stream BUFFERED-STREAM 101BF73B>         ; The shell is set to the
                                           ; stream that is stdin and
                                           ; stdout for csh.  The terminal
                                           ; remains stderr.

> (format shell "/usr/games/banner test~%") ; Send a banner command to csh.
NIL
```

```
> (type-until-hung shell)                   ; Print the output from banner.


  #####  ######   ####    #####
    #    #        #         #
    #    #####    ####      #
    #    #           #      #
    #    #        #  #      #
    #    ######   ####      #

NIL
> (format shell "unknown~%")                ; Send an erroneous command
NIL                                         ; to the shell.
> unknown: Command not found.               ; This is stderr output; it is
                                            ; not trapped.
(type-until-hung shell)                     ; No output was sent to stdout.
NIL
> (format shell "exit~%")                   ; Make csh exit.
NIL
> (close shell)                             ; Close the stream used for
                                            ; stdin and stdout.
NIL
```

**Note:** Many UNIX programs exit on reading **EOF** from **stdin**. If you run a program that does not, you should always give it an exit command before you close its input stream.

## Example 2—Creating Output for Lisp to Process

In the next example, output is created that Lisp can process. If you had an UNIX program named `"frob"` that created output you wanted to process in Lisp, you could write the following:

```
(defun frobify (input arguments)
  (with-open-stream (frob (run-unix-program "frob" :output :stream
                                                   :input input
                                                   :arguments arguments
                                                   :wait nil))
    (process-frob-stream frob)))
```

The `input` parameter could be a filename or a stream, and the arguments would consist of a list of the necessary command line arguments to the program `"frob"`.

## Example 3—Program with Unusual Syntax

In this final example, if you had a natural language front-end for a program that took an unusual syntax, you could write the following:

```
(defun run-strange-interactive-program-smartly (args)
  (with-open-stream (strange (run-unix-program
                               "strange-interactive-program"
                               :input :stream
                               :output :stream
                               :error-output :output
                               :wait nil
                               :arguments
                               (strangify-arguments args)))
                    (loop
                     (write-string
                      (strangify-command
                       (read-english-command *standard-input*))
                      strange)
                     (write-line
                      (smarten-up-response (read-strange-response strange))
                      *standard-output*))))
```

In this program,

- **read-english-command** knows how to parse a natural language command.
- **strangify-command** turns the parsed English command into a command that is suitable for the strange program.
- **read-strange-response** knows how to parse the output from the strange program.
- **smarten-up-response** turns output from the strange program into a readable format.

## run-unix-program

**Purpose:** The function **run-unix-program** provides the ability to run other UNIX programs from the Lisp environment.

**Syntax:** **run-unix-program** *name* &key `:input :output :error-output` `:wait :arguments` `:if-input-does-not-exist` `:if-output-exists` `:if-error-output-exists` [*Function*]

**Remarks:** The *name* argument is a pathname or an object that can be coerced to a pathname. It represents the name of the program to be run. If the pathname is a relative pathname, each directory in the environment variable **PATH** is searched for the filename that corresponds to *name*. If the pathname is an absolute pathname, that is, if the name begins with a slash (/), then that file is used. The namestring of *name* is the **argv**[0] parameter for the program.

Four values are returned by **run-unix-program**.

1. The first value is a stream. If either the **:input** or the **:output** keyword argument is **:stream**, that stream communicates with the running process and is the first value returned. If neither keyword is **:stream**, the first value is **nil**.
2. If the **:error-output** keyword argument is **:stream**, the second value returned is the resulting input stream from which Lisp can read the program's error output. If **:error-output** is not **:stream**, the second value returned is **nil**.
3. If the **:wait** keyword argument is **t**, the third value is the exit status of the program that was run. Otherwise the exit status is **nil**.
4. If the program is running, the fourth value is its UNIX process id. If the program has run to completion, the fourth value is **nil**.

This function is an extension to Common Lisp.

## run-unix-program

Examples:

```
;; This example shows how to run a program and have Lisp wait for it to
;; complete.  The example assumes you have a program called "banner"
;; on /usr/games.

> (run-unix-program "csh")
% /usr/games/banner test

  #####  ######   ####    #####
    #    #       #          #
    #    #####    ####      #
    #    #            #     #
    #    #       #    #     #
    #    ######   ####      #

% exit
% NIL
NIL
0
NIL
```

# Chapter 9. Compiling Lisp Programs

# Chapter 9. Compiling Lisp Programs

# Introduction to the Compiler

The Compiler allows you to transform interpreted code to a more efficient form. Generally, compiled code behaves exactly like its interpreted counterpart but does not do as much checking. As a result, compiled code may cause obscure behavior in situations in which the interpreted version would signal an error. Compiling can be done in two ways: by compiling a file or by compiling an individual function in the current environment.

To compile a file, use the function **compile-file**, which produces binary files from Lisp source files. The compiled functions contained in the binary files become available for use when the binary files are loaded into Lisp. By convention, Lisp source files usually have the extension `.lisp`. The corresponding extension for binary files is one of the following:

- `.lbin`

  If the value of the **:target** option to **compile-file** is the default, the extension for binary files is `.lbin`. Typing the expression `(compile-file "foo.lisp")` produces the file `"foo.lbin"`.

- `.2bin`

  If the value of the **:target** option to **compile-file** is `68020`, the extension for binary files is `.2bin`. Typing the expression `(compile-file "foo.lisp")` produces the file `"foo.2bin"`.

To compile an interpreted function in the current Lisp environment, use the function **compile**, which replaces the interpreted function's definition with the compiled version.

The functions and forms listed below are relevant to compiling files.

| | |
|---|---|
| **clear-undef** | **declare** |
| **compile** | **eval-when** |
| **compile-file** | **locally** |
| **compiled-function-p** | **proclaim** |
| **compiler-let** | **the** |
| **compiler-options** | |

## Compiling and Keyword Options

You may specify keyword options to the function **compile-file** and to the function **compiler-options**, which is an extension to Common Lisp.

```
compile-file input-pathname &key :output-file                    [Function]
                                 :messages
                                 :warnings
                                 :fast-entry
                                 :tail-merge
                                 :notinline
                                 :target
```

```
compiler-options &key :messages                                  [Function]
                      :warnings
                      :fast-entry
                      :tail-merge
                      :notinline
                      :target
```

The keyword options have default values, but you may supply different ones by specifying values to the keyword arguments of **compile-file**. In the following code, for example, `nil` is specified as the value of the keyword `:tail-merge`. This tells the Compiler to compile the file `"foo"` and to do no tail recursion optimizations on it.

```
(compile-file "foo" :tail-merge nil)
```

The keyword options **:messages**, **:warnings**, **:fast-entry**, **:tail-merge**, **:notinline**, and **:target** are extensions to Common Lisp. You may reset their default values with the function **compiler-options**.

**Note:** You cannot reset the default value of the keyword **:output-file**.

## How the Compiler Uses Declarations

An important difference between the Interpreter and the Compiler lies in how they treat declarations. Declarations that are ignored by the Interpreter are often used by the Compiler as advice in order to produce faster and more efficient code. This applies in particular to **type** declarations. Global declarations, or proclamations, affect only the dynamic bindings of variables; thus proclaiming a variable to be of certain type has no effect on the lexical bindings of that variable. There are several categories of declarations.

### Special Declarations

A **special** declaration specifies that the given variables are all special variables. References to the variables will thus refer to the dynamic binding of the variables. If the **declare** special form is used to make a special declaration, the declaration observes the rules of lexical scope. If, however, a **special** proclamation is made, all bindings of variables with the given name are special.

The effect of a **special** declaration is exactly the same in both the Compiler and the Interpreter. However, compiled access to special variables is done through an in-line coding of the function **symbol-value**. If the special variable is unbound, obscure errors may result. You may declare the function **symbol-value** notinline in situations where checking is desired.

### Type Declarations

Declarations known as **type** declarations, which are ignored in interpreted code, are extensively used by the Compiler to produce faster and more efficient code. Their use is especially important in code that involves array and sequence manipulation. Declaring simple vectors and simple arrays may result in significant improvements in the running speed of the code. Similarly, specialized arithmetic operations, fixnum operations in particular, tend to run faster than their generic counterparts.

Care must be exercised when using declarations. For example, the expression `(+ x y)` cannot be optimized to its fixnum version even if the `x` and `y` arguments have been declared as fixnums. Because of potential overflow into the nonfixnum domain, the Compiler needs to know that the result type of this form is a fixnum.

You can pass result type information to the Compiler by using **ftype** or the special form **the**. In the following expression, for example, the Compiler views the *form* argument as having the type given by the *type-specifier* argument:

(the *type-specifier form*)

In the following code, the Compiler will replace the call to the generic **aref** function with **svref**, which expands to a simple in-line memory reference:

```
(defun foo (x)
       (declare (type simple-vector x))
       (aref x 1))
```

Similarly, the call to generic + is replaced by in-line fixnum arithmetic in this code:

```
(defun baz (x y)
       (declare (type fixnum x y))
       (the fixnum (+ x y)))
```

Often only fixnum arithmetic is needed. Type declarations can be used to write fixnum-specific versions for generic routines. For example, you can define binary fixnum addition as follows:

```
(defmacro fixnum-plus (x y)
  '(the fixnum (+ (the fixnum ,x) (the fixnum ,y))))
```

This code specifies that `fixnum-plus` will be used wherever + occurs, and this may result in faster arithmetic operations.

Type information can be propagated using **ftype** declarations. For example, the proclamation `(proclaim '(ftype (function (fixnum) fixnum) erk))` causes the fixnum version of addition to be used in the code that follows:

```
(defun do-some-arithmetic (x)
       (declare (type fixnum x))
       (the fixnum (+ x (erk x))))
```

## Inline and Notinline Declarations

The Interpreter ignores **inline** and **notinline** declarations.

Many of the accessors, setters, and constructors (such as **car**, **cdr**, **cons**, and **svref**) for Common Lisp data types are coded in-line by the Compiler. Obscure errors may result when these operations are applied to illegitimate data. Any Lisp function can be declared notinline. If a function is so declared, a call to the corresponding function replaces the in-line code. The **inline** declaration has no effect on user-supplied functions.

## Ignore Declarations

The **ignore** declaration prevents the Compiler from issuing a warning when the variable in question is not referred to in the body of the code.

## Optimization Declarations

Common Lisp provides a mechanism for telling the Compiler your priorities in trade-offs with which it might be faced. The implementation of the Common Lisp specification translates the given advice to Compiler keyword options.

In Common Lisp you can proclaim or declare advice to the Compiler with an **optimize** declaration. The following code is an example:

```
(optimize (speed 2) (safety 1))
```

There are four optimization classes. Each class is assigned an integer between 0 and 3 inclusive. The default values are shown below:

| | |
|---|---|
| **speed** | **3** |
| **safety** | **1** |
| **space** | **0** |
| **compilation-speed** | **0** |

The keyword options **:fast-entry**, **:tail-merge**, and **:notinline**, which can be specified to **compile-file** and **compiler-options**, may be activated or suppressed by declared advice to the Compiler by using the four optimization classes. Conditions under which they are activated are the following:

| | |
|---|---|
| **:fast-entry** | **safety = 0** |
| **:tail-merge** | **speed = 1, 2, 3** |
| **:notinline** | **speed = 0** |

## clear-undef

**Purpose:** At the end of compilation, the Compiler prints out a list of all the currently undefined functions. The function **clear-undef** resets this list to **nil**.

**Syntax:** **clear-undef** [*Function*]

**Remarks:** This function is an extension to Common Lisp.

## compile

**Purpose:** The function **compile** is used to compile an interpreted function in the current Lisp environment. It replaces the interpreted function's definition with the compiled version. It produces a compiled code object from a lambda expression. The lambda expression is the argument *definition* if it is present; otherwise the function definition of the symbol *name* is the relevant lambda expression.

If the *name* argument is **nil**, **compile** returns the compiled code object; otherwise the function definition of the symbol *name* is set to the compiled code object, and **compile** returns that symbol.

**Syntax:** **compile** *name* **&optional** *definition* [*Function*]

**Examples:**

```
> (defun foo (x) (+ x x))
FOO
> (compile 'foo)
;;; Compiling function FOO...assembling...emitting...done.
;;; Warning: Redefining FOO
FOO
> (foo 2)
4
> (funcall (compile nil '(lambda (x) (+ x x))) 3)
;;; Compiling function...assembling...emitting...done.
6
```

## compile-file

**Purpose:** The function **compile-file** produces binary files from Lisp source files. The compiled functions contained in the binary files become available for use when the binary files are loaded into Lisp.

**Syntax:** **compile-file** *input-pathname* &key `:output-file` `:messages` `:warnings` `:fast-entry` `:tail-merge` `:notinline` `:target` [*Function*]

**Remarks:** The function converts a file specified by the *input-pathname* argument into compiled code.

If given, the **:output-file** option specifies where the compiled code is sent; its argument should be a pathname or a string describing a valid filename. The binary file that is produced from the source file is given that name, and any existing file with that name is overwritten.

If this option is not specified or if its argument is bound to **nil**, the file extension for the binary file is determined as follows:

- source file has the extension `.lisp`

  In this case, the extension `.lisp` is replaced by one of these extensions.

  - `.lbin`

    If the value of the **:target** option is the default, the extension `.lisp` becomes `.lbin`.

  - `.2bin`

    If the value of the **:target** option is 68020, the extension `.lisp` becomes `.2bin`.

- source file does not have the extension `.lisp`

  When the extension is different from `.lisp` or when there is no extension, one of these extensions is attached to the end of the source filename.

  - `.lbin`

    If the value of the **:target** option is the default, the extension `.lbin` is attached to the end of the filename.

  - `.2bin`

    If the value of the **:target** option is `68020`, the extension `.2bin` is attached to the end of the filename.

The default value of this option is **nil**.

The **:messages** option controls the fate of the progress messages issued by the Compiler. A value of **nil** means issue no progress messages; otherwise the value should specify a stream to which messages can be sent. The default value is **t**, which sends the messages to the standard terminal device.

The **:warnings** option controls the warnings issued by the Compiler. A value of **nil** means issue no warnings; otherwise the value must specify a stream to which warnings can be sent. The default value is **t**, which sends the warnings to ***error-output***.

If the **:fast-entry** option has a non-**nil** value, the Compiler does not insert code to check the number of arguments on entry to a function with a fixed number of arguments. Thus, calls to functions compiled in this manner are slightly faster. The default value is **nil**.

If the **:tail-merge** option has a non-**nil** value, the Compiler converts tail-recursive calls to iterative constructions and thus eliminates the overhead of some function calls. The default value is **t**.

If the **:notinline** option has a non-**nil** value, the Compiler behaves as if all functions have been declared notinline; see the section on inline and notinline declarations for more details. The default value is **nil**.

If the value of the **:target** option is `68020`, the Compiler generates binary files specifically for the MC68020 processor. Such files will run slightly faster in some cases, but they will not run on MC68010 processors. The binary files produced have a default extension of `.2bin`. The default value of the **:target** option is `68K`. In this case, the Compiler produces code that can be run on both the MC68010 and the MC68020 processors, and the default file extension is `.lbin`.

The keywords **:messages**, **:warnings**, **:fast-entry**, **:tail-merge**, **:notinline**, and **:target** are extensions to Common Lisp.

**See Also:** **compiler-options**

## compiler-options

**Purpose:** The function **compiler-options** resets the default values of the keyword options **:messages**, **:warnings**, **:fast-entry**, **:tail-merge**, **:notinline**, and **:target** of the function **compile-file**.

**Syntax:** **compiler-options** `&key :messages` *[Function]*

```
                  :warnings
                  :fast-entry
                  :tail-merge
                  :notinline
                  :target
```

**Remarks:** The keyword **:messages** controls the fate of the progress messages issued by the Compiler. A value of **nil** means issue no progress messages; otherwise the keyword should specify a stream to which messages can be sent. The default value is **t**, which sends the messages to the standard terminal device.

The keyword **:warnings** controls the warnings issued by the Compiler. A value of **nil** means issue no warnings; otherwise the keyword value must specify a stream to which warnings can be sent. The default value is **t**, which sends the warnings to ***error-output***.

If the **:fast-entry** keyword has a non-**nil** value, the Compiler does not check the number of arguments on entry to a function with a fixed number of arguments. Thus, calls to functions compiled in this manner are slightly faster. The default value of **:fast-entry** is **nil**.

If the **:tail-merge** option has a non-**nil** value, the Compiler converts tail-recursive calls to iterative constructions and thus eliminates the overhead of some function calls. The default value of **:tail-merge** is **t**.

If the **:notinline** option has a non-**nil** value, the Compiler behaves as if all functions have been declared notinline; see the section on inline and notinline declarations for more details. The default value of **:notinline** is **nil**.

If the value of the **:target** keyword is `68020`, the Compiler generates binary files specifically for the MC68020 processor. Such files will run slightly faster in some cases, but they will not run on MC68010 processors. The binary files produced have a default extension of `.2bin`. The default value of the **:target** option is `68K`. In this case, the Compiler produces code that can be run on both the MC68010 and the MC68020 processors, and the default file extension is `.lbin`.

This function is an extension to Common Lisp.

**See Also:** **compile-file**

## declare

**Purpose:** The **declare** special form may be used to make declarations within certain forms. Declarations may occur in lambda expressions and in the following forms:

| | |
|---|---|
| **defmacro** | **labels** |
| **defsetf** | **let** |
| **deftype** | **let*** |
| **defun** | **locally** |
| **do** | **macrolet** |
| **do*** | **multiple-value-bind** |
| **do-all-symbols** | **prog** |
| **do-external-symbols** | **prog*** |
| **do-symbols** | **with-open-stream** |
| **dolist** | **with-open-file** |
| **dotimes** | **with-output-to-string** |
| **flet** | **with-input-from-string** |

**Syntax:** declare {*decl-spec*}* [*Special Form*]

*decl-spec*::= (special {*var*}*) |
(type *type-specifier* {*var*}*) |
(ftype *type-specifier* {*function-name*}*) |
(function *function-name* ({*type-specifier*}*) {*type-specifier*}*) |
(inline {*function-name*}*) |
(notinline {*function-name*}*) |
(ignore {*var*}*) |
(optimize {*quality value*}*) |
(declaration {*declaration-name*}*)

*quality*::= speed | space | safety | compilation-speed

*value*::= 0 | 1 | 2 | 3

**Remarks:** Declarations may only occur where specified by the syntax of these forms.

Macros may expand into declarations as long as this syntax is observed.

The declaration specifier argument is not evaluated.

For more information, see the *Sun Common Lisp Reference Manual.*

Examples:

```
> (defun foo (y)                                     ; This y is regarded
       (declare (special y))                         ; as special.
       (let ((y t))                                  ; This y is regarded
          (list y                                    ; as lexical.
                (locally (declare (special y)) y)))) ; This y refers to the
                                                     ; special binding of y.
FOO
> (foo nil)
(T NIL)
```

## eval-when

**Purpose:** The special form **eval-when** is used to specify when a particular body of code is to be executed.

This time is defined by the *situation* arguments. The value of each argument must be **compile**, **load**, or **eval**.

If **eval** is specified, the evaluator evaluates the *form* arguments at execution time. If **compile** is specified, the Compiler evaluates the *form* arguments at compilation time. If **load** is specified and the file containing the **eval-when** is compiled, then the forms are compiled; they are executed when the output file produced by the Compiler is loaded.

The value of the last *form* evaluated is returned as the result of **eval-when**. If no forms are executed, **eval-when** returns **nil**.

**Syntax:** **eval-when** ({*situation*}*) {*form*}* [*Special Form*]

**Remarks:** The *form* arguments are executed in order.

**Examples:**

```
> (setq foo 3)
3
> (eval-when (compile) (setq foo 2))
NIL
> foo
3
> (eval-when (eval) (setq foo 2))
2
> foo
2
```

## locally

**Purpose:** The **locally** macro makes local declarations that affect only its *form* arguments.

**Syntax:** **locally** {*declaration*}* {*form*}* [*Macro*]

**Examples:**

```
> (defun foo (y)                                   ; This y is regarded
       (declare (special y))                       ; as special.
       (let ((y t))                                ; This y is regarded
          (list y                                  ; as lexical.
                (locally (declare (special y)) y)))) ; This y refers to the
                                                   ; special binding of y.
FOO
> (foo nil)
(T NIL)
```

## proclaim

**Purpose:** The **proclaim** function makes a global declaration, or proclamation.

A proclamation whose declaration specifier declares a variable to be **special** makes all occurrences of that variable name special references.

**Syntax:** **proclaim** *decl-spec* [*Function*]

**Remarks:** Although the effect of a proclamation is global, it may be overridden by a local declaration.

The argument of **proclaim** is evaluated. It may therefore be a computed declaration specifier.

**Examples:**

```
> (proclaim '(special prosp))
T
> (setq prosp 1 reg 1)
1
> (let ((prosp 2)(reg 2))
    (set 'prosp 3)(set 'reg 3)
    (list prosp reg))
(3 2)
> (list prosp reg)
(1 3)
```

**See Also:** **defvar**

**defparameter**

(In the *Sun Common Lisp Reference Manual*)

## the

**Purpose:** The special form **the** specifies that the value produced by a form will be of a certain type.

The *value-type* argument is a type specifier; it is not evaluated. The *form* argument is evaluated.

The **the** special form returns the value or values that result from *form*.

**Syntax:** **the** *value-type form* [*Special Form*]

**Remarks:** You can use the macro **setf** with **the** type declarations. In this case the declaration is transferred to the form that specified the new value. The resulting **setf** form is then analyzed.

**Examples:**

```
> (the list '(a b))
(A B)
> (the (values integer list) (values 5 '(a b)))
5
(A B)
> (let ((i 100))
    (declare (fixnum i))
    (the fixnum (1+ i)))
101
```

# Chapter 10. Storage Management in Common Lisp

# Chapter 10. Storage Management in Common Lisp

## About the Garbage Collector

A program called the ***Garbage Collector*** manages storage in Sun Common Lisp. The Garbage Collector (GC) is a stop-and-copy compacting collector. It organizes memory into areas. Each area consists of some number of 64-kilobyte Lisp segments and is typed according to the kind of data it may contain.

**Note:** In this chapter, the notion of a segment refers to the way in which Lisp organizes memory, not to the way in which the operating system or underlying hardware is structured.

### Areas

The possible area types are read-only, static, and dynamic.

***Read-only areas*** contain data structures that can never be garbage-collected and whose contents can never be altered. The system code and some system data structures are put into read-only areas.

***Static areas*** contain data structures that can never be garbage-collected, but whose contents may be altered. Certain permanent data structures are put in static areas.

***Dynamic areas*** are the areas from which storage is recovered through garbage collection. Dynamic storage is organized into two semi-spaces: Dynamic-0-Area and Dynamic-1-Area. Normally, only one of the semi-spaces is in use at any given time. When a garbage collection occurs, the retained objects from the semi-space in use are copied into the other semi-space, which then becomes the current semi-space.

Dynamic and static memory are expanded as required. For example, dynamic space can be expanded if the total size of the retained objects does not permit creation of a new object. This expansion typically happens upon the attempted creation of a new object. Another point of expansion occurs when the space for nondynamic storage is exhausted. If this space is exhausted, it will be expanded.

## Garbage Collection—What Happens

When the Garbage Collector is invoked, the message

```
;;; GC:
```

is displayed on the screen. When garbage collection is finished, further information is printed after the colon. Here is an example of a final message:

```
;;; GC: 360 words [1440 bytes] of dynamic storage in use.
;;; 458140 words [1832560 bytes] of free storage available before a GC.
;;; 916640 words [3666560 bytes] of free storage available if GC is
                                                              disabled.
```

The first line states the amount of dynamic storage in use. The second line tells how much storage is available before the next garbage collection occurs. The third line tells how much storage would be available if the Garbage Collector were disabled.

If all the dynamic storage in both semi-spaces is to be used for consing, call the function **gc-off** to disable the Garbage Collector. Note, however, that disabling the Garbage Collector may make future garbage collection impossible. When the Garbage Collector is disabled, all memory normally allocated for dynamic semi-spaces is used for creating objects. This typically implies that garbage collection is impossible because there may not be enough room in which to copy the retained objects.

Whenever 50 percent of the remaining space has been consumed, the Garbage Collector asks whether you want to reconsider and enable it. The optional argument *no-reconsideration* to **gc-off** controls this periodic querying. When *no-reconsideration* is non-**nil**, you can use all the available storage without being asked to reconsider. The default value is **nil**.

## The Room Function

The function **room** reports information on the memory management state. Depending on the arguments, **room** reports three levels of detail.

### Example 1

```
> (room nil)
;;; 412144 words [1648576 bytes] free
NIL
```

This reports the least amount of information, namely the same sort of information that is reported in the second line of the garbage collection message.

### Example 2

```
> (room)
;;; 46376 words [185504 bytes] of dynamic storage in use.
;;; 412124 words [1648496 bytes] of free storage available before a GC.
;;; 870624 words [3482496 bytes] of free storage available if GC is
                                                               disabled.
NIL
```

The code above reports an intermediate amount of information, the same sort of information that is reported by the garbage collection message.

### Example 3

```
> (room t)
;;; 46410 words [185640 bytes] of dynamic storage in use.
;;; 412090 words [1648360 bytes] of free storage available before a GC.
;;; 870590 words [3482360 bytes] of free storage available if GC is
                                                               disabled.
;;; Semi-space Size: 1792K bytes [28 segments]
;;; Current Dynamic Area: Dynamic-0-Area
;;; GC Status: Enabled
;;; Reserved Free Space: 192K bytes [3 segments]
;;; Memory Growth Limit: 12800K bytes [200 segments], total
;;; Memory Growth Rate: 256K bytes [4 segments]
;;; Reclamation Ratio: 33% desired free after garbage collection
;;; Area Information:
;;; Name                        Size [used/allocated]
;;; ----                        ----
;;; Dynamic-0-Area              182K/1792K bytes,  3/28 segments
;;; Dynamic-1-Area              0K/1792K bytes,    0/28 segments
;;; Static-Area                 1158K/1216K bytes, 19/19 segments
;;; Readonly-Pointer-Area       488K/512K bytes,   8/8 segments
;;; Readonly-Non-Pointer-Area   2640K/2688K bytes, 42/42 segments
NIL
```

The first three lines in Example 3 contain the same type of information as the garbage collection message. The next three lines give the current size of the semi-spaces and report which of the two semi-space areas is currently in use and whether the Garbage Collector has been disabled:

```
;;; Semi-space Size: 1792K bytes [28 segments]
;;; Current Dynamic Area: Dynamic-0-Area
;;; GC Status: Enabled
```

The next four lines give the current values of parameters that affect memory expansion:

```
;;; Reserved Free Space: 192K bytes [3 segments]
;;; Memory Growth Limit: 12800K bytes [200 segments], total
;;; Memory Growth Rate: 256K bytes [4 segments]
;;; Reclamation Ratio: 33% desired free after garbage collection
```

***Reserved Free Space*** is the amount of free storage that is currently in reserve for read-only and static data. The ***Memory Growth Limit*** is the maximum total size to which the Lisp system is allowed to grow. The ***Memory Growth Rate*** is the amount of additional storage that is added each time memory is expanded. The ***Reclamation Ratio*** determines when memory is expanded by specifying the desired ratio of free storage to total storage. If garbage collection has not reached this ratio (or a higher one), then at a point at which a garbage collection would normally take place, memory is expanded instead. You may use the function **change-memory-management** to alter these parameters or to force memory expansion explicitly.

The last section of the expanded **room** report gives a breakdown of the current storage allocation by area. For each area, the report gives the total amount of storage allocated to that area and the current amount in use.

When a garbage collection does not reclaim enough room to continue consing, there are two options: to return to the top level (type **:a** to do this) or to disable the Garbage Collector and start consing in the area normally reserved for copying (type :c to do this). Here is what may be displayed after invocation of a function (`cons-a-lot`, in the example) that requires a large amount of storage:

```
>>Error: Not enough storage after GC.

CONS-A-LOT:
   Required arg 0 (CONS-COUNT): 1000000

:A    Abort to Lisp Top Level
:C    GC will be disabled:
CONSers will use the storage normally reserved
for copying currently allocated dynamic storage.
The next GC might fail.
->
```

When half of the remaining free segments have been allocated, another continuable error may be signaled:

```
>>Error: There are 1407206 words left and GC is disabled.

CONS-A-LOT:
   Required arg 0 (CONS-COUNT): 1000000

:A    Abort to Lisp Top Level
:C    GC Will Remain Disabled
->
```

When only one free segment remains, a continuable error is signaled. Continuing from this error will probably result in disaster:

```
>>Error: There are 16134 words left and GC is disabled.
This is the last warning before memory is exhausted.

CONS-A-LOT:
   Required arg 0 (CONS-COUNT): 1000000

:A    Abort to Lisp Top Level
:C    GC Will Remain Disabled
->
```

## Altering Storage Allocator Parameters

The function **change-memory-management** is used to alter the parameters affecting memory expansion. Its syntax is as follows:

**change-memory-management** `&key :growth-limit` *[Function]*
```
:growth-rate
:expand :expand-p
:reclamation-ratio
:expand-stack
:help
```

The parameters are set by specifying various keyword arguments, which are described as follows:

- **:growth-limit**

  This keyword argument specifies the maximum size of memory in segments (a segment is 64 kilobytes).

- **:growth-rate**

  This sets the number of dynamic segments that are added to each dynamic semi-space whenever memory is expanded.

- **:expand**

  This forces an immediate expansion of each semi-space of dynamic memory by the specified number of segments. If it is necessary, the current **:growth-limit** is increased.

- **:expand-p**

  A value of **t** for **:expand-p** specifies that rather than invoking a garbage collection, memory should be expanded at the next point at which additional storage is required.

- **:reclamation-ratio**

  This alters the desired ratio of free to total dynamic storage and is specified as a fraction between 0.0 and 1.0.

- **:expand-stack**

  This increases the number of segments allocated to the stack by the specified number of segments.

- **:help**

  This reminds the user of the various keyword arguments.

## Examples

```
(change-memory-management :growth-limit 246)

(change-memory-management :growth-limit 246 :growth-rate 6)

(change-memory-management :expand 20)

(change-memory-management :expand-p t)

(change-memory-management :reclamation-ratio .33)

(change-memory-management :expand-stack 5)

(change-memory-management :help t)
```

## Sample Lisp Session

```
> (change-memory-management :help t)
Supply:
:growth-limit as a number of segments,
:growth-rate as a number of segments,
:reclamation-ratio as a fraction between 0.0 and 1.0,
:expand-stack as a number of segments,
:expand as a number of segments, or
:expand-p, as T or NIL,
where 1 segment is 64K bytes
T
```

## change-memory-management

**Purpose:** The function **change-memory-management** is used to alter some of the parameters of the storage allocator. The parameters are set by specifying various keyword arguments.

**Syntax:** **change-memory-management** `&key :growth-limit :growth-rate :expand :expand-p :reclamation-ratio :expand-stack :help` *[Function]*

**Remarks:** The keyword argument **:growth-limit** specifies the maximum size of memory in segments (a segment is 64 kilobytes).

The keyword argument **:growth-rate** sets the number of dynamic segments that are added to each dynamic semi-space when memory is expanded.

The keyword argument **:expand** forces an immediate expansion of each semi-space of dynamic memory by the specified number of segments. The current **:growth-limit** is increased if necessary.

If the keyword argument **:expand-p** has a value of **t**, memory is expanded at the next point at which additional storage is required. No garbage collection is invoked.

The keyword argument **:reclamation-ratio** alters the desired ratio of free to total dynamic storage and is specified as a fraction between 0.0 and 1.0.

The keyword argument **:expand-stack** increases the number of segments allocated to the stack by the specified number of segments.

The keyword **:help** reminds the user of the various keyword argument options.

This function is an extension to Common Lisp.

**Examples:**

```
> (room t)
;;; 47390 words [189560 bytes] of dynamic storage in use.
;;; 411110 words [1644440 bytes] of free storage available before a GC.
;;; 869610 words [3478440 bytes] of free storage available if GC is
                                                           disabled.
;;; Semi-space Size: 1792K bytes [28 segments]
;;; Current Dynamic Area: Dynamic-0-Area
;;; GC Status: Enabled
;;; Reserved Free Space: 192K bytes [3 segments]
;;; Memory Growth Limit: 12800K bytes [200 segments], total
;;; Memory Growth Rate: 256K bytes [4 segments]
;;; Reclamation Ratio: 33% desired free after garbage collection
;;; Area Information:
```

```
;;; Name                         Size [used/allocated]
;;; ----                         ----
;;; Dynamic-0-Area               186K/1792K bytes,  3/28 segments
;;; Dynamic-1-Area               0K/1792K bytes,    0/28 segments
;;; Static-Area                  1158K/1216K bytes, 19/19 segments
;;; Readonly-Pointer-Area        488K/512K bytes,   8/8 segments
;;; Readonly-Non-Pointer-Area    2640K/2688K bytes, 42/42 segments
NIL
> (change-memory-management :growth-limit 202 :growth-rate 16
                            :reclamation-ratio 0.25)
T
> (room t)
;;; 47626 words [190504 bytes] of dynamic storage in use.
;;; 410874 words [1643496 bytes] of free storage available before a GC.
;;; 869374 words [3477496 bytes] of free storage available if GC is
                                                              disabled.
;;; Semi-space Size: 1792K bytes [28 segments]
;;; Current Dynamic Area: Dynamic-0-Area
;;; GC Status: Enabled
;;; Reserved Free Space: 192K bytes [3 segments]
;;; Memory Growth Limit: 12928K bytes [202 segments], total
;;; Memory Growth Rate: 1024K bytes [16 segments]
;;; Reclamation Ratio: 25% desired free after garbage collection
;;; Area Information:
;;; Name                         Size [used/allocated]
;;; ----                         ----
;;; Dynamic-0-Area               187K/1792K bytes,  3/28 segments
;;; Dynamic-1-Area               0K/1792K bytes,    0/28 segments
;;; Static-Area                  1158K/1216K bytes, 19/19 segments
;;; Readonly-Pointer-Area        488K/512K bytes,   8/8 segments
;;; Readonly-Non-Pointer-Area    2640K/2688K bytes, 42/42 segments
NIL
```

## gc

**Purpose:** The function gc invokes the Garbage Collector.

**Syntax:** gc *[Function]*

**Remarks:** The function returns three values: the amount of dynamic storage in use, the amount of free storage available before a GC, and the amount of free storage available if the Garbage Collector were to be disabled. These are the numbers displayed in the gc message when the variable ***gc-silence*** is set to **nil**, which is the default.

This function is an extension to Common Lisp.

**Examples:**

```
> (room)
;;; 47776 words [191104 bytes] of dynamic storage in use.
;;; 410724 words [1642896 bytes] of free storage available before a GC.
;;; 869224 words [3476896 bytes] of free storage available if GC is
                                                           disabled.
NIL
> (gc)
;;; GC: 472 words [1888 bytes] of dynamic storage in use.
;;; 458028 words [1832112 bytes] of free storage available before a GC.
;;; 916528 words [3666112 bytes] of free storage available if GC is
                                                           disabled.
1888
1832112
3666112
> (room)
;;; 492 words [1968 bytes] of dynamic storage in use.
;;; 458008 words [1832032 bytes] of free storage available before a GC.
;;; 916508 words [3666032 bytes] of free storage available if GC is
                                                           disabled.
NIL
```

**See Also:** ***gc-silence***

## gc-off

**Purpose:** The function **gc-off** disables the Garbage Collector. Its optional argument controls whether disabling the Garbage Collector causes periodic reconsideration of the issue. If *no-reconsideration* is set to **t**, you can use all the available storage without being asked to reconsider. The default value of *no-reconsideration* is **nil**.

**Syntax:** **gc-off** `&optional` *no-reconsideration* [*Function*]

**Remarks:** If **Dynamic-1-Area** is the current Dynamic Area, a garbage collection will occur when **gc-off** is invoked.

This function is an extension to Common Lisp.

**Examples:**

```
> (room t)
;;; 838 words [3352 bytes] of dynamic storage in use.
;;; 457662 words [1830648 bytes] of free storage available before a GC.
;;; 916162 words [3664648 bytes] of free storage available if GC is
                                                               disabled.
;;; Semi-space Size: 1792K bytes [28 segments]
;;; Current Dynamic Area: Dynamic-0-Area
;;; GC Status: Enabled
;;; Reserved Free Space: 192K bytes [3 segments]
;;; Memory Growth Limit: 12800K bytes [200 segments], total
;;; Memory Growth Rate: 576K bytes [9 segments]
;;; Reclamation Ratio: 33% desired free after garbage collection
;;; Area Information:
;;; Name                        Size [used/allocated]
;;; ----                        ----
;;; Dynamic-0-Area              4K/1792K bytes,    1/28 segments
;;; Dynamic-1-Area              0K/1792K bytes,    0/28 segments
;;; Static-Area                 1158K/1216K bytes, 19/19 segments
;;; Readonly-Pointer-Area       488K/512K bytes,   8/8 segments
;;; Readonly-Non-Pointer-Area   2640K/2688K bytes, 42/42 segments
NIL
> (gc-off)
T
> (room t)
;;; 1008 words [4032 bytes] of dynamic storage in use.
;;; 916244 words [3664976 bytes] of free storage available before a GC.
;;; 916244 words [3664976 bytes] of free storage available if GC is
                                                               disabled.
;;; Semi-space Size: 1792K bytes [28 segments]
;;; Current Dynamic Area: Dynamic-0-Area
;;; GC Status: Disabled
;;; Reserved Free Space: 192K bytes [3 segments]
;;; Memory Growth Limit: 12800K bytes [200 segments], total
;;; Memory Growth Rate: 576K bytes [9 segments]
;;; Reclamation Ratio: 33% desired free after garbage collection
;;; Area Information:
;;; Name                        Size [used/allocated]
;;; ----                        ----
;;; Dynamic-0-Area              5K/1792K bytes,     1/28 segments
;;; Dynamic-1-Area              0K/1792K bytes,     0/28 segments
;;; Static-Area                 1158K/1216K bytes, 19/19 segments
;;; Readonly-Pointer-Area       488K/512K bytes,    8/8 segments
;;; Readonly-Non-Pointer-Area   2640K/2688K bytes, 42/42 segments
NIL
```

**See Also:** **gc-on**

## gc-on

**Purpose:** This function enables the Garbage Collector.

**Syntax:** **gc-on** [*Function*]

**Remarks:** This function is an extension to Common Lisp.

**Examples:**

```
> (gc-off)
T
> (room t)
;;; 514 words [2056 bytes] of dynamic storage in use.
;;; 916738 words [3666952 bytes] of free storage available before a GC.
;;; 916738 words [3666952 bytes] of free storage available if GC is
                                                              disabled.
;;; Semi-space Size: 1792K bytes [28 segments]
;;; Current Dynamic Area: Dynamic-0-Area
;;; GC Status: Disabled
;;; Reserved Free Space: 192K bytes [3 segments]
;;; Memory Growth Limit: 12800K bytes [200 segments], total
;;; Memory Growth Rate: 576K bytes [9 segments]
;;; Reclamation Ratio: 33% desired free after garbage collection
;;; Area Information:
;;; Name                        Size [used/allocated]
;;; ----                        ----
;;; Dynamic-0-Area              3K/1792K bytes,     1/28 segments
;;; Dynamic-1-Area              0K/1792K bytes,     0/28 segments
;;; Static-Area                 1158K/1216K bytes, 19/19 segments
;;; Readonly-Pointer-Area       488K/512K bytes,    8/8 segments
;;; Readonly-Non-Pointer-Area  2640K/2688K bytes, 42/42 segments
NIL
> (gc-on)
T
> (room t)
;;; 684 words [2736 bytes] of dynamic storage in use.
;;; 457816 words [1831264 bytes] of free storage available before a GC.
;;; 916316 words [3665264 bytes] of free storage available if GC is
                                                              disabled.
;;; Semi-space Size: 1792K bytes [28 segments]
;;; Current Dynamic Area: Dynamic-0-Area
;;; GC Status: Enabled
;;; Reserved Free Space: 192K bytes [3 segments]
;;; Memory Growth Limit: 12800K bytes [200 segments], total
;;; Memory Growth Rate: 576K bytes [9 segments]
;;; Reclamation Ratio: 33% desired free after garbage collection
;;; Area Information:
```

```
;;; Name                        Size [used/allocated]
;;; ----                        ----
;;; Dynamic-0-Area              3K/1792K bytes,     1/28 segments
;;; Dynamic-1-Area              0K/1792K bytes,     0/28 segments
;;; Static-Area                 1158K/1216K bytes, 19/19 segments
;;; Readonly-Pointer-Area       488K/512K bytes,   8/8 segments
;;; Readonly-Non-Pointer-Area   2640K/2688K bytes, 42/42 segments
NIL
```

**See Also:** gc-off

## *gc-silence*

**Purpose:** If you set the variable ***gc-silence*** to **t**, all garbage collection messages are suppressed. The default value of ***gc-silence*** is **nil**.

**Syntax:** ***gc-silence*** [*Variable*]

**Remarks:** This variable is an extension to Common Lisp.

**Examples:**

```
> (room)
;;; 101682 words [406728 bytes] of dynamic storage in use.
;;; 356818 words [1427272 bytes] of free storage available before a GC.
;;; 815318 words [3261272 bytes] of free storage available if GC is
                                                               disabled.
NIL
> (let ((*gc-silence* t)) (gc))
2968
1831032
3665032
> (room)
;;; 762 words [3048 bytes] of dynamic storage in use.
;;; 457738 words [1830952 bytes] of free storage available before a GC.
;;; 916238 words [3664952 bytes] of free storage available if GC is
                                                               disabled.
NIL
```

## get-stack-remaining

**Purpose:** The function **get-stack-remaining** returns the number of words of stack space that may be used before the stack will overflow.

**Syntax:** **get-stack-remaining** [*Function*]

**Remarks:** One memory segment contains 64 kilobytes, or 16,384 words.

This function is an extension to Common Lisp.

## room

**Purpose:** The function **room** prints information about the current state of internal memory on the standard output.

If the optional argument is specified as **nil**, a terse summary is printed. If the optional argument is non-**nil**, a verbose description is given. If no argument is specified, **room** prints a moderate amount of information.

**Syntax:** **room** `&optional` *x* [*Function*]

**Remarks:** The standard output is defined by the value of the variable ***standard-output***.

# Chapter 11. The Flavor System

# Chapter 11. The Flavor System

# Introduction to Flavors

## What Is an Object?

A program can often be thought of as an algorithm for manipulating objects. A graphics program might deal with such objects as circles, polygons, line segments, and windows. An operating system might deal with files, processes, disks, and terminals.

For each of these objects, there are certain operations that can be performed on it. Given a circle, a programmer might want to know its area or diameter or might want to have it move to the right a certain number of pixels. For a window, a programmer might want to move it, to expose it, or to place an object in it at a certain point.

In object-oriented programming, this viewpoint is modified slightly. Objects are not treated as passive entities but as active participants. You do not calculate the area of a circle; rather, you request that the circle calculate its area and return the result. Similarly, you do not move a window; it moves itself or places an object at a certain point.

## What Is a Flavor?

The Flavor System is designed to provide a convenient and modular way of creating objects and of describing what each object can do.

There is a standard terminology for dealing with objects. An abstract object, such as a circle or a window, is called an ***abstract data type*** or a ***flavor***. A particular incarnation of one of these is called an ***instance***. Requesting an action or information from a particular instance is called ***sending*** the instance a ***message***. The action that is performed by an instance upon receiving a particular message is that instance's ***method*** for the message.

The following example illustrates the operation of defining a flavor.

```
(defflavor circle-flavor
    (x-center y-center radius)
    ()
    :gettable-instance-variables)

(defmethod (circle-flavor :diameter) ()
    (* 2 radius))
```

```
(defmethod (circle-flavor :area) ()
    (* pi radius radius))           ; pi is a built-in constant

(defmethod (circle-flavor :moveright) (amount)
    (setq x-center (+ x-center amount)))
```

The **defflavor** macro creates a flavor called `circle-flavor`. The next line says that a circle has three ***instance variables***: the $x$ position of its center, the $y$ position of its center, and its radius. The instance variables of a flavor are similar to the fields of a structure. They are names for the important attributes of an instance of a flavor. Every instance of a circle has these three instance variables.

The syntax of the **defmethod** in the preceding example is nearly identical to that of a **defun**. The function name is replaced by (*flavor message*). The lambda list has the full generality of a **defun** lambda list. (See the chapter "Program Structure" of the *Sun Common Lisp Reference Manual* for more information on lambda lists.)

All message names are keywords. The `:gettable-instance-variables` option to **defflavor** in the preceding example creates methods for the three messages `:radius`, `:x-center`, and `:y-center`. These methods simply return the value of the corresponding instance variable.

The next few lines describe other messages that an instance of a `circle-flavor` can receive and the methods that correspond to these messages. The first **defmethod** says that if an instance of the flavor `circle-flavor` receives a `:diameter` message, it should return a value equal to twice the radius. Similarly, if the instance receives an `:area` message, it should calculate and return $\pi$ `radius`$^2$.

The last **defmethod** says that if an instance of flavor `circle-flavor` receives a `:moveright` message, the instance should increment the value of its `x-center` instance variable by the amount of the single argument.

## Sending a Message

If the variable `a-circle` is bound to an instance of a circle, evaluating the form `(send a-circle :area)` returns the area of the circle. Evaluating `(send a-circle :moveright 15)` moves the circle to the right by 15 units.

The function **send** sends the keyword message *message* to the instance *instance*. The Flavor System finds the appropriate method corresponding to this message. The remaining arguments are passed to this method and are bound to the variables in the lambda list of the **defmethod**.

The syntax of **send** is

**send** ***instance message*** **&rest** ***args*** [*Function*]

The syntax of **send** may seem complicated for two reasons. First, the ***message*** argument, although it is a symbol in the keyword package, is not a keyword argument. (See the chapter "Program Structure" of the *Sun Common Lisp Reference Manual* for more information on keyword arguments.) Second, when the ***args*** arguments are passed to the ***instance***'s method for ***message***, the manner in which these arguments are bound depends on the lambda list of the appropriate **defmethod** and cannot be determined *a priori*.

The following examples should clarify the process of sending a message.

First, a flavor named `flavor-1` is created. It has no instance variables. The value of the variable `x1` is set to an instance of this flavor.

```
> (defflavor flavor-1 () () ) ; flavor-1 is a simple flavor;
FLAVOR-1
> (setq x1 (make-instance 'flavor-1))
;;; Compiling function...tail merging...assembling...emitting...done.
#<Instance FLAVOR-1 92D4D3>
```

The first method for this flavor has two required arguments. If any additional arguments are passed, the first is bound to the variable `c`, and a list of the remaining is bound to the variable `d`.

```
> (defmethod (flavor-1 :msg1) (a b &optional c &rest d)
     (list a b c d))
:MSG1
> (send x1 :msg1 1 2)
(1 2 NIL NIL)
> (send x1 :msg1 1 2 3 4)
(1 2 3 (4))
```

Notice that neither the `x1` argument nor the `:msg1` argument to the **send** function is used in the binding process. These two arguments are removed before the rest of the arguments are bound to the variables of the lambda list of the **defmethod**.

Here is a slightly more complicated example:

```
> (defmethod (flavor-1 :msg2) (a &optional c &key key1 key2)
     (list a c key1 key2))
:MSG2
```

The preceding method has one required argument, one optional argument, and two keyword arguments.

```
> (send x1 :msg2 3 5 :key2 'abc :key1 'xyz)
(3 5 XYZ ABC)
```

The argument 3 is bound to variable a in the lambda list, and the argument 5 is bound to the variable c in the lambda list. The remaining arguments are treated as keyword arguments. The value of 'abc gets bound to the variable key2, and the value of 'xyz gets bound to key1. Again, the first two arguments to the send function are not involved in the binding of values to variables in the lambda list. The *message* argument serves only to determine which of x1's methods to call.

```
> (send x1 :msg2 :key1 'abc :key2 'xyz)
(:KEY1 ABC NIL XYZ)
```

The :key1 argument is treated not as a keyword argument but as the first required argument. The value of 'abc gets bound to the second argument. The remaining arguments are treated as keyword-value pairs.

The following code creates another flavor, flavor-2. The variable x2 contains an instance of that flavor. It also creates a :msg2 method for flavor-2.

```
> (defflavor flavor-2 () ())
FLAVOR-2
> (setq x2 (make-instance 'flavor-2))
;;; Compiling function...tail merging...assembling...(assembling...(sharing
previous code vector))(sharing previous code vector)emitting...done.
#<Instance FLAVOR-2 95DDF3>
> (defmethod (flavor-2 :msg2)(a b c &optional d)
      (list a b c d))
:MSG2
```

Notice that flavor-2's lambda list for the message :msg2 is quite different from flavor-1's lambda list. Send the same message and arguments to x2.

```
> (send x2 :msg2 :key1 'abc :key2 'xyz)
(:KEY1 ABC :KEY2 XYZ)
```

In this case, the value of each argument gets bound to a variable of the lambda list. Not surprisingly, a different result is returned.

## Flavors Versus Structures

You can define a circle as a structure rather than as a flavor. The code for this would be the following:

```
(defstruct circle
    x-center
    y-center
    radius)

(defmacro  circle-diameter (circle)
    '(* 2 (circle-radius ,circle)))
    .
    .
    .
```

This code may be faster than code using flavors because structures use in-line substitution, whereas flavors use function calls. However, the Flavor System is designed to support modularity, extensibility, and the ability to perform generic operations.

## Modularity

When you program any large system, it is almost essential that you break the system into smaller, more manageable pieces. You should be able to deal with each piece, often called a ***module***, without paying too much attention to any of the other pieces.

The interface between modules should be thin, that is, no module should have to concern itself with the internal workings of any other module. Instead, each module has an ***interface***—a small number of functions, variables, and data types that can be accessed from outside the module. What the module does is clearly specified, but not how it does what it does.

A module may be thought of as a "black box." The module specifies what instructions it understands and what it returns in response to those instructions. The internal workings of the black box are of no concern to anything outside of it.

The Flavor System is designed to support modularity. You can think of a flavor or a group of flavors as a module. The specifications for a flavor are the methods that exist for that flavor. The user of a flavor need not know what the instance variables of the flavor are nor what the corresponding methods are.

For example, `circle-flavor` in the initial example of defining a flavor could have been defined as follows:

```
(defflavor circle-flavor (x-center y-center diameter)
    ()
    :gettable-instance-variables)

(defmethod (circle-flavor :radius) ()
    (/ diameter 2))

(defmethod (circle-flavor :area) ()
    (* 1/4 pi diameter diameter))

(defmethod (circle-flavor :moveright) (amount)
    (setq x-center (+ x-center amount)))
```

You need not recompile a program that interacts with an object of type `circle-flavor` only through the messages `:x-center`, `:y-center`, `:radius`, `:diameter`, `:area`, and `:moveright`. A future implementation of `circle-flavor` might rearrange the instance variables and add more of them or add more methods, but old programs using `circle-flavor` should not require modifications.

If structures were used in a program, every piece of code that referred to a `circle` would have to be recompiled whenever the implementation of `circle` changed. With flavors and messages, you should be able to ignore details of the flavor's implementation.

## Generic Operations

A ***generic operation*** is one that depends on the type of argument passed. This is not as unusual as it sounds. Many computers have two different instructions for multiplication of numbers—one for integers and one for floating-point numbers.

Suppose the flavor `rectangle-flavor` were defined as follows:

```
(defflavor rectangle-flavor
    (height width top-left-x top-left-y)
    ()
    :gettable-instance-variables)

(defmethod (rectangle-flavor :area) ()
    (* height width))
```

Given an object that is an instance of either `circle-flavor` or `rectangle-flavor`, you can send it an `:area` message, and no matter whether it is a circle or a rectangle, the appropriate method is used. If you were using structures, you would have to test the object to see if it was a circle or a rectangle (or any other object with an area), and you would have to decide whether to call **circle-area** or **rectangle-area**.

An important aspect of generic operations is ***inheritability***, which is the ability of one flavor to automatically incorporate all the instance variables and methods of another flavor.

Suppose you wanted to model rectangular pieces of farmland. The attributes of such pieces of farmland might include `types-of-crops`, `soil-acidity`, and the like. Such pieces might also have height, width, and all the other attributes of a rectangle. The flavor `farmland-flavor` could then be defined as follows:

```
(defflavor farmland-flavor
    (types-of-crops soil-acidity)      ; instance variables
    (rectangle-flavor)                 ; component flavors
    :gettable-instance-variables)      ; options
```

The second argument to the **defflavor** in the above example is a list of flavors that the flavor `farmland-flavor` inherits. An instance of flavor `farmland-flavor` not only has the instance variables `types-of-crops` and `soil-acidity`, but it also inherits all the instance variables of `rectangle-flavor`. More importantly, the instance inherits all the messages and corresponding methods of `rectangle-flavor`. It automatically understands an `:area` message. If you were to write a new method for `rectangle-flavor` called `:perimeter`, an instance of `farmland-flavor` would automatically understand this new message and call `rectangle-flavor`'s method for that message.

## Extensibility

To add new methods to a flavor, you need only add new code. You need not modify already existing code. Moreover, you should be able to make these additions without knowing any details of the flavor's implementation.

Suppose that another programmer has written a flavor called `circle-flavor` that understands all of the messages described in the initial example (see "What Is an Object?"). This flavor is just what you need, except that you would like a `:perimeter` message.

You can write the following code:

```
(defmethod (circle-flavor :perimeter) ()
    (* pi (send self :diameter)))
```

When the body of the **defmethod** is executed, `self` is bound to the instance to which the message is sent. By sending a message to `self`, the `:perimeter` method can find its own diameter without knowing anything about the implementation of `circle-flavor`.

## Creating Simple Instances

To actually create instances of flavors, use the **make-instance** function.

The argument to **make-instance** is the name of a flavor. There may be additional keyword arguments. What is returned is an instance, which is usually displayed as follows:

`#<Instance` *flavor-name hexadecimal-address*`>`

(To modify what is displayed, see the section "The Vanilla Flavor.")

For example, typing the expression `(setq a-circle (make-instance 'circle-flavor))` creates an instance of flavor `circle-flavor`. It returns an instance of a circle whose displayed representation looks like

```
#<Instance CIRCLE-FLAVOR 3A25DC3>
```

Since all of the instance variables of `a-circle` are unbound, it is not very interesting. Not much can be done with `a-circle`. If the **describe** function is called on it, the result is as follows:

```
(describe a-circle)
An instance of flavor CIRCLE-FLAVOR.
Instance variables:
X-CENTER unbound
Y-CENTER unbound
RADIUS unbound
 #<Instance CIRCLE-FLAVOR 3A25DC3>
```

Its `x-center`, `y-center`, and `radius` are all unbound. If you were to type the expression `(send a-circle :y-center)` the result would be

```
FLAVORS-SYSTEM::UNBOUND
```

which is a special value used by the system for marking unbound instance variables.

The **describe** function explicitly ignores modularity. It displays the names and values of all the instance variables, which are supposed to be hidden from the user of a flavor. The values of the instance variables that are displayed can be a great aid in debugging a program. Since **describe** is primarily intended for debugging, you get implementation-dependent information.

**Note:** Because the output of **describe** is implementation dependent, you should remember that it may change.

In the previous example, the output of **describe** shows that `a-circle` is a rather uninteresting circle because everything about it is unspecified. Typing

```
(send a-circle :radius)
```

returns the uninteresting value `FLAVORS-SYSTEM::UNBOUND`, and typing

```
(send a-circle :area)
```

signals an error because `FLAVORS-SYSTEM::UNBOUND` is not a number.

The original definition of `circle-flavor` can be modified slightly so that **make-instance** creates useful instances of circles.

```
(defvar *default-x-center* 0)
(defvar *default-y-center* 0)

(defflavor circle-flavor
   ((x-center *default-x-center*)
    (y-center *default-y-center*)
     radius)
   ()
   :gettable-instance-variables
   :settable-instance-variables
   :initable-instance-variables)
```

There are four things to take note of in this example.

- The `:settable-instance-variables` option creates methods for the three messages `:set-x-center`, `:set-y-center`, and `:set-radius`. Each of these methods takes a single argument and causes the corresponding instance variable to be set to the value of the argument.
- The `:gettable-instance-variables` option creates methods for the three messages `:x-center`, `:y-center`, and `:radius`. Each of these methods takes no arguments and returns the value of the corresponding instance variable.
- The `:initable-instance-variables` option allows you to initialize the values of the instance variables in the **make-instance** function call. Thus you can type

  ```
  (setq a-circle
    (make-instance 'circle-flavor :x-center 3 :y-center 5 :radius 10))
  ```

  to create a circle whose instance variables will have the values specified.
- The instance variables `x-center` and `y-center` now have default values. If no initial value is specified for either of these in the call to **make-instance**, then both get the value 0. The value of `radius` is unbound.

The :gettable-instance-variables and :initable-instance-variables options are redundant in this case because both options are implied by the :settable-instance-variables option. (See the section "Defining Flavors.") The readability and understandability of the program may be improved by including all three.

Now you can type the following:

```
> (setq a-circle
     (make-instance 'circle-flavor :y-center 3 :radius 5))
#<Instance CIRCLE-FLAVOR 369ABC>
```

At last a useful circle is created. Notice that since a new instance of circle-flavor has been created, it has a different hexadecimal number in its printed representation. You can now type the following:

```
> (describe a-circle)
An instance of flavor CIRCLE-FLAVOR.
Instance variables:
X-CENTER 0
Y-CENTER 3
RADIUS 5
 #<Instance CIRCLE-FLAVOR 369ABC>
> (send a-circle :radius)
5
> (send a-circle :diameter)
10
> (send a-circle :set-x-center (send a-circle :y-center))
3
> (send a-circle :x-center)
3
> (send a-circle :area)
78.53981
> (send a-circle :set-radius 2)
2
> (send a-circle :area)
12.56637
```

## Mixing Flavors

The power and versatility of the Flavor System is enhanced by its ability to mix flavors. You can create specialized flavors from already existing flavors.

### Simple Mixing

Suppose two flavors called **farmland-flavor** and **plist-mixin** already exist on your system. The flavor **farmland-flavor** is the one described in the example in the section "Introduction to Flavors"; **plist-mixin** provides the functionality of a property list by supplying methods for these three messages:

(**send** *instance* :putprop *property value*)

(**send** *instance* :getprop *property* **&optional** *default-value*)

(**send** *instance* :remprop *property*)

The following code is one possible implementation of **plist-mixin**:

```
(defflavor plist-mixin
   ((property-list nil))
   ())

(defmethod (plist-mixin :putprop) (property value)
   (setf (getf property-list property) value))

(defmethod (plist-mixin :getprop) (property &optional default)
   (getf property-list property default))

(defmethod (plist-mixin :remprop) (property)
   (remf property-list property))
```

To create a flavor that understands the messages dealing with farmland and the messages for a property list, you can write the following:

```
(defflavor farmland-with-plist-mixin
    ()                                  ; no new instance variables
    (farmland-flavor plist-mixin))      ; component flavors
```

This code creates a new flavor, which is named **farmland-with-plist-mixin**, that understands the messages of both **farmland-flavor** and **plist-mixin**. Its instance variables are the union of the instance variables of **farmland-flavor** and **plist-mixin**.

## Inheriting Values

A flavor may inherit the instance variables and methods of another flavor. In the section "Introduction to Flavors," `farmland-flavor` was defined so that it automatically included all of the methods and instance variables of `rectangle-flavor`. This was accomplished by including `rectangle-flavor` in the list of component flavors for `farmland-flavor`.

Inheritance lets you take a general abstract data type and build a more specialized data type on top of it. A farmer in Massachusetts could create the flavor `cranberry-growing-farmland-flavor`, and his Wisconsin counterpart could write the flavor `cattle-grazing-farmland-flavor`. Although each flavor would have its own specialized set of messages and methods, each would also have an understanding of the basic messages for `farmland-flavor`.

A flavor with components can be thought of as a tree. A flavor may have several flavors on its list of component flavors. Each of these component flavors may have its own components.

Simple flavors that provide a few convenient methods are frequently found at the very bottom of the tree. Flavors, such as `plist-mixin`, that provide a few convenient operations but that are never to be instantiated by themselves are often called ***mixins***.

## Creating the Components List

The order in which component flavors appear in the components list is important because it may affect the order in which methods are combined. The more specific flavors should appear at the beginning of the list. The less specific, more basic flavors should appear at the end.

When a flavor is compiled, the tree of components is converted into a single list of component flavors. A preorder depth-first search is performed on the tree.

Suppose the following were true:

- `flavor1`'s component flavors are `flavor2` and `flavor3`.
- `flavor2`'s component flavors are `flavor4` and `flavor5`.
- `flavor3`'s component flavors are `flavor5`, `flavor6`, and `flavor7`.
- The other flavors have no component flavors.

The tree of component flavors would look like the following diagram:

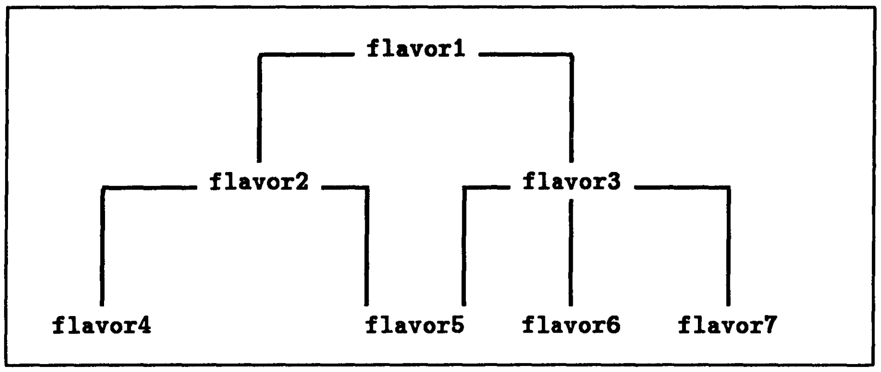

**Figure 11–1. Tree of Component Flavors**

The final list of component flavors would be the following:

**flavor1, flavor2, flavor4, flavor5, flavor3, flavor6, flavor7**

This is how the Flavor System performs the calculation. The first item on the list is **flavor1**. Each of its children is examined. The first child is **flavor2**. It is added to the list, and then its children, **flavor4** and **flavor5**, are added. Since **flavor2** has no more children, the next child of **flavor1**, namely **flavor3**, is added to the list and examined. The first child of **flavor3**, which is **flavor5**, is already on the list, so it is not added again. Then **flavor3**'s remaining two children, **flavor6** and **flavor7**, are added.

The instance variables of the resulting flavor are the union of all the instance variables of the flavors on the components list. If an instance variable **var** appears in both **flavor3** and **flavor7**, both flavors refer to the same instance variable when referring to **var**. The different component flavors can use this sharing of instance variables to communicate with each other. If an instance variable is initialized by a default value in more than one component, it receives the value specified in the component closest to the front of the list.

## Introduction to Combined Methods

Until this point in our discussion of methods, ***primary methods*** have been the only type discussed. They are the default. In the section "Introduction to Flavors," **rectangle-flavor** had primary methods for **:height**, **:width**, **:top-left-x**, **:top-left-y**, and **:area**. The flavor **farmland-flavor** had primary methods for **:types-of-crops** and **:soil-acidity**; **farmland-flavor** did not have a primary method for **:height** or **:area** but inherited **rectangle-flavor**'s method.

When several flavors on the component list provide a primary method for a given message, all but the first are ignored. Thus you can write a flavor that is identical to a second flavor except for a single message. The new flavor simply inherits the first flavor but overrides the primary method of that first flavor by writing a new method.

More often, if a new flavor is written on top of an old flavor, the new flavor does not want to override any of the old flavor's messages. There may be certain messages for which the new flavor expects the old flavor to provide a method. Instead of overriding these methods, the new flavor may want to add some preprocessing or postprocessing (or both). This means that the new flavor wants to provide what is called ***daemon method types***. The two method types **:before** and **:after** provide this flexibility.

To specify such a method type, the first argument to **defmethod** should be

*(flavor method-type message)*

where *method-type* must be either **:before** or **:after**.

When a message is sent, it is not handled by a single primary method. The Flavor System calculates a ***combined method*** to handle the message. The combined method for a message consists of all of the **:before** methods for that message, a single primary method, and all of the **:after** methods for that message. The values returned are the values returned by the primary method. The **:before** methods are called in the order in which their flavor appears on the components list. The primary method is the single method that is the first primary method to appear on the components list. The **:after** methods are called in the reverse order.

Although it may seem strange, this ordering is intentional. The components list in the **defflavor** should be from the highest-level flavor to the lowest-level. If `flavor1` is built on top of `flavor2`, then `flavor1` should not care how `flavor2` is defined. The flavor `flavor1` thinks of itself as doing some precomputation and some postcomputation. It does not concern itself with whether `flavor2` has several daemons.

## More on Combined Methods

Creating a flavor can cause several other flavors to be mixed in, and these flavors can themselves cause other flavors to be mixed in. For a given message, several component flavors may have methods associated with them. Each of these methods has one of the following types associated with it:

- **:primary** is the default if no type is given to **defmethod**.
- **:before** indicates a daemon method that should be run before the **:primary** method.

- **:after** indicates a daemon method that should be run after the **:primary** method.

A ***method combination type*** associated with each message indicates how the individual methods are to be combined to form a combined method. Each component flavor is checked to see if it has a **:method-combination** option. (See the section "Defining Flavors.") This option gives both a type and a modifier. Any message not specified in an option has the default type **:daemon** and the default modifier **:base-flavor-last**. If more than one flavor specifies the combination type for a message, these two types must be the same.

The modifier, which is **:base-flavor-first** or **:base-flavor-last**, indicates the order in which the methods are to be combined. The ***base flavor*** is the last flavor in the component list. The type modifier **:base-flavor-last** indicates that the methods should be combined in the order in which they appear on the list. The type modifier **:base-flavor-first** indicates that the methods should be combined in the reverse order.

Here are the possible method combination types:

- **:progn, :and, :or, :list, :append, :nconc**

  All of the primary methods are inserted in the appropriate function in the order indicated by the type modifier. Methods of type **:before** and **:after** are ignored.

- **:daemon**

  All of the **:before** methods are called in the order specified by the type modifier. Then a single primary method is called. If the type modifier is **:base-flavor-last** (the default), the component list is searched until a flavor with a **:primary** method is found. If the type modifier is **:base-flavor-first**, the list is searched in the reverse order. Finally, all of the **:after** methods are called in the reverse order of that specified by the modifier.

## Some Examples of Combination Types

Suppose that the list of component flavors is `a-flavor`, `b-flavor`, `c-flavor`, `d-flavor`, and `e-flavor`; `a-flavor` and `d-flavor` have **:before** methods for the message `:msg`; `b-flavor` and `d-flavor` have **:after** methods; and `c-flavor` and `e-flavor` have **:primary** methods. The methods are combined in the following ways:

- If the combination type is **:list** and the modifier is **:base-flavor-last**, the methods are called in the following order:

  - `c-flavor`'s **:primary** method. Let the value it returns be ***val1***.
  - `e-flavor`'s **:primary** method. Let the value it returns be ***val2***.

  The result returned from the combined method is the list (***val1 val2***). Note that the **:before** and **:after** daemons are ignored.

- If the combination type is **:or** and the modifier is **:base-flavor-first**, the methods are called in the following order:
  - `e-flavor`'s **:primary** method is evaluated. If it returns a non-nil value, that value is returned immediately, and no other method is evaluated.
  - `c-flavor`'s **:primary** method is evaluated. The combined method returns whatever values this method returns.
- If the combination type is **:daemon** and the modifier is **:base-flavor-last**, the methods are called in the following order:
  - `a-flavor`'s **:before** method
  - `d-flavor`'s **:before** method
  - `c-flavor`'s **:primary** method
  - `d-flavor`'s **:after** method
  - `b-flavor`'s **:after** method

  The values returned from the call to `c-flavor`'s **:primary** method are returned as the values of the combined method.

  Note that **:daemon :base-flavor-last** is the default combination type.
- If the combination type is **:daemon** and the modifier is **:base-flavor-first**, the methods are called in the following order:
  - `d-flavor`'s **:before** method
  - `a-flavor`'s **:before** method
  - `e-flavor`'s **:primary** method
  - `b-flavor`'s **:after** method
  - `d-flavor`'s **:after** method

  Again, the values returned from the combined method are the values returned from the single primary method.

## Whoppers and Wrappers

Whoppers and wrappers are a specialized means of controlling the actions of a combined method. A primary aspect of whoppers and wrappers is that they go around the combined method for a message. That is, they are wrapped around the execution of the combined method.

The difference between whoppers and wrappers is that whoppers are functions, whereas wrappers are macros. A flavor can provide both whoppers and wrappers, in which case the wrapper goes around the whopper. If more than one flavor on the component list provides a whopper and/or a wrapper, all the wrappers are wrapped around all the whoppers, and all the whoppers are wrapped around the combined method.

The syntax of a whopper is identical to that of a **defmethod** except that there is no *method-type* in the second argument.

**defwhopper** (*flavor message*) *lambda-list* [*Macro*]
{*declaration* | *documentation*}* {*form*}*

When *message* is sent to an instance, the outermost whopper is called. It is executed as if it were the method for that message. The *form* arguments are executed until one of the following three macros, which are legal only within the *form* arguments of a **defwhopper**, is called:

**continue-whopper** {*arg*}* [*Macro*]

**continue-whopper-all** [*Macro*]

**lexpr-continue-whopper** {*arg*}* *arg-list* [*Macro*]

Each of these macros causes the execution of the wrapped combined method. If there is another whopper inside of this one, it is called; otherwise the combined method is called. The macro **continue-whopper-all** causes the wrapped code to be called with the identical arguments that were passed to this execution of the whopper code.

The macro **continue-whopper** allows you to modify the arguments or to pass different ones. The **lexpr-continue-whopper** macro is similar to **continue-whopper** except that the last argument must be a list of arguments. The argument list that is passed to the wrapped code is the last argument to **lexpr-continue-whopper** appended to the end of the list of all the other arguments. The function **lexpr-continue-whopper** is analogous to the Common Lisp function **apply**. (See the chapter "Program Structure" of the *Sun Common Lisp Reference Manual* for more information on the **apply** function.)

These three macros return as their values whatever values are returned from the execution of the wrapped combined methods. The **defwhopper** may return these values unchanged, or it may examine them and return different values.

In general, a whopper is used to set up a special dynamic environment in which the combined methods may be executed. For example, a flavor's **:before** or **:after** method might call the function **throw**. A whopper can be set up to catch the **throw**. The following example demonstrates how a whopper might do this:

```
(defwhopper (a-flavor :a-message) (&rest args)
   (declare (ignore args))
   (catch 'a-throw (continue-whopper-all)))
```

Alternatively, several different component methods of a combined method might communicate via a special variable. A **defwhopper** can be set up to initialize this special variable to a default value.

A wrapper is like a whopper except that the text of the wrapper is literally wrapped around the text of the combined method. This code modifies the combined method so that it is only called if the first argument is non-**nil**. Otherwise **nil** is returned immediately.

The complete syntax of **defwrapper** is the following:

**defwrapper** *(flavor message)* *[Macro]*
*(lambda-list . body-var)*
*{declaration | documentation}* {form}**

To calculate the new combined method, the Flavor System first calculates what the combined method would be without the wrapper. The forms are then evaluated with *body-var* bound to a list of length one, whose single element is the unwrapped combined method. The value that the last form returns becomes the body of the wrapped combined method. The argument list of the wrapped combined method is the *lambda-list* argument.

The format of a wrapper is best shown by an example:

```
(defwrapper (a-flavor :a-message) ((arg1 arg2) . body)
   `(unless (null arg1)
        . ,body))
```

Notice that in the last line, `body` has a comma before it, but `arg1` does not. The reason is that the variables `arg1` and `arg2` are arguments to the combined method and are bound when the message is sent. However, `body` is bound when the *form* arguments of the **defwrapper** are being evaluated to create the combined method.

During the execution of the body of a **defwhopper** or a **defwrapper**, a special lexical environment is created in which the variable **self** is bound to the instance that received the message that caused the invocation of this whopper or wrapper.

(See the chapter "Program Structure" of the *Sun Common Lisp Reference Manual* for more information on lexical environments.) The variable **self** is also bound to the instance during the execution of the body of a **defmethod**. See the section "Defining Methods for Messages."

> **Note:** Wrappers have several advantages over whoppers. Wrappers do not cause an extra function call. Wrappers have greater generality because you can use the full power of the macro expansion facility. Finally, wrappers can set up both lexical and dynamic environments, whereas whoppers can only set up dynamic environments.
>
> Despite these advantages, wrappers have certain drawbacks. For example, wrappers may waste space. If a wrapper is used in more than one combined method, its entire code is duplicated for each. Changing a wrapper requires recompiling the entire combined method, whereas changing a whopper only requires recompiling the body of the whopper.

## Defining Flavors

The **defflavor** macro is used to define a new flavor. Its syntax is

defflavor *flavor-name* ({*var* | (*var default-form*)}*) ({*flavor*}*) {*option*}* [*Macro*]

## Required Arguments

These arguments to **defflavor** are required:

- Flavor name

  The first argument is the name of the flavor being defined. By convention, high-level flavor names end with the suffix `-flavor`. Flavors that are intended to implement a special optional feature have names that end with the suffix `-mixin`.

  **Note:** If *flavor-name* is already defined as a flavor, the Flavor System assumes that you are redefining *flavor-name* and throws away the old definition. All instances that are built on top of the old definition of *flavor-name* may cease to work correctly. Sending them a message may cause undefined behavior. You must re-instantiate each of them.

- Instance variables

  The second argument is a list. Each element of the list has either the form *var* or the form (*var default-form*).

  In both cases, *var* is declared to be an instance variable of the flavor *flavor-name*. The latter form specifies a default form to be evaluated as the initial value of *var*. See the section "Creating Instances" for more information.

- Component flavors

  The third argument is *flavor-name*'s component list. Whenever an instance of *flavor-name* is created, the instance automatically inherits the instance variables and methods of each of these flavors. These flavors must already be defined. See the section "Mixing Flavors" for more information on component flavors.

## Options

The remaining arguments to **defflavor** are options. Each option has the form (*option-name* {*arg*}*). If no arguments are being passed, the simpler form *option-name* can be used.

All of the options are effective both when an instance of *flavor-name* is being created directly and when *flavor-name* is being instantiated indirectly because it is a component of another flavor.

### Frequently Used Options

Three frequently used options are the following:

- **:settable-instance-variables**

  Each argument must be an instance variable of this flavor. For each *var* argument, a method for the `:set-var` message is created that sets the value of the instance variable argument to the supplied value. For example, evaluating `(send test-circle :set-radius 5)` sets the instance variable `radius` of `test-circle` to the value 5.

  If this option is given no arguments, then a method for the `:set-var` message is created for every instance variable *var* in the second argument to **defflavor**.

  The option (`:settable-instance-variables` *var* ...) automatically implies the options (`:gettable-instance-variables` *var* ...) and (`:initable-instance-variables` *var* ...). Thus, when the Flavor System creates the method for the `:set-var` message, it also creates a method for the `:var` message and allows **:var** as a keyword argument when **make-instance** creates an instance of this flavor. See the **:gettable-instance-variables** option and the **:initable-instance-variables** option for more information.

  If this option is given an argument list, then each argument must appear in the second argument to **defflavor**. Otherwise the Flavor System assumes that there is a misspelling and signals an error.

- **:gettable-instance-variables**

  Each argument must be an instance variable of this flavor. For each *var* argument, a method for the `:var` message is created that returns the value of that instance variable.

  If this option is given no arguments, then a method for the `:var` message is created for every instance variable *var* in the second argument to **defflavor**.

  If this option is given an argument list, then each argument must appear in the second argument to **defflavor**. Otherwise the Flavor System assumes that there is a misspelling and signals an error.

- **:initable-instance-variables**

  Each argument must be an instance variable of this flavor. For each *var* argument, a **:var** keyword option is allowed when calling **make-instance**. This option lets you initialize the value of an instance variable when creating the instance. Instance variables that can be initialized in this manner are said to be ***initializable***.

  If this option is given no arguments, then every instance variable in the second argument to **defflavor** is initializable.

  If this option is given an argument list, then each argument must appear in the second argument to **defflavor**. Otherwise the Flavor System assumes that there is a misspelling and signals an error.

## Less Frequently Used Options

Simple flavor applications usually do not need these options:

- **:init-keywords**

  The arguments to this option are the keywords that the flavor's method for the `:init` message understands. When the **make-instance** function is called, an ***initial property list*** is created. The Flavor System checks whether all the keywords on the initial property list either are arguments in one of the component flavor's **:init-keywords** options or are keywords corresponding to instance variables that have been made initializable using the **:initable-instance-variables** or **:settable-instance-variables** option.

  See the section "Creating Instances" for more information.

- **:default-init-plist**

  The argument list to this option must have the form

  *keyword1 form1 keyword2 form2* ...

  When an instance containing *flavor-name* as a component is created, an initial property list is formed from the list of keyword-value pairs passed to **make-instance**. The function **make-instance** then examines the keywords in the argument list to **:default-init-plist** in turn. If any of these keywords is already on the property list, the keyword's corresponding form is ignored. Otherwise the keyword's corresponding form is evaluated, and the keyword-value pair is added to the property list.

  With this option, a flavor can give a default value to variables or options of component flavors. The values are not used if you have provided another default.

The **:default-init-plist** pairs that initialize instance variables are not put on the initial property list passed to the `:init` message.

- **:required-instance-variables**

  Each argument is an instance variable. Any flavor that is eventually instantiated and that contains *flavor-name* as a component must have the specified arguments as instance variables. Instance variables declared as **:required-instance-variables** may be used by this flavor's methods even though they do not occur in the list of instance variables.

  The main difference between declaring an instance variable in the instance variable list and declaring it as a **:required-instance-variable** is stylistic. The former declares that a flavor is "responsible" for the variable and ensures that it is initialized properly. The latter says that *flavor-name* makes use of this variable but that some other flavor is responsible for it.

- **:required-init-keywords**

  Each argument is a keyword.

  Whenever an instance of *flavor-name* is being created, each keyword argument must appear on the initial property list that is created. Each argument must appear either as a keyword-value pair in the call to **make-instance** or as a keyword-form pair in some component flavor's **:default-init-plist**. See the section "Creating Instances" for more information.

- **:required-methods**

  Each of the arguments is the name of a message. If an attempt is made to instantiate *flavor-name*, then some component flavor must provide a method for each of these messages.

  This option can be used in a base flavor that is not intended to be a stand-alone flavor. It might send itself messages, expecting that one of the other component flavors built on top of it has provided a method for that message. This option causes the Flavor System to catch errors when the flavor is compiled rather than when an illegal message is sent at run-time.

- **:required-flavors, :included-flavors**

  For these options, each argument is a flavor name. When an instance of *flavor-name* is instantiated, all flavor arguments must also be components.

  These options differ as to what happens when one of the listed flavors does not appear in the final list of component flavors. The **:required-flavors** option signals an error. The **:included-flavors** option adds the listed flavors to the end of the component list of flavors.

Specifying a flavor as an argument to these options differs from including the flavor in the list of component flavors. By specifying a flavor as an argument, you are stating that it must appear at some point in the component list. You are not causing the flavor to occur immediately in the component list. This is useful when you want to ensure that the **:before** and **:after** daemons are called in the correct order. These declarations indicate a flavor's indirect dependence on another flavor.

- **:no-vanilla-flavor**

  Every component list usually has the flavor **vanilla-flavor** automatically added to its end. The vanilla flavor provides default methods for several useful operations. Any flavor that has *flavor-name* as a component flavor does not have **vanilla-flavor**. See the section "The Vanilla Flavor" for more information.

- **:method-combination**

  Each argument has the form (*type modifier msg1 msg2* ...).

  The *type* argument is a defined type of method combination; the *modifier* argument is either **:base-flavor-first** or **:base-flavor-last**; and *msg1*, *msg2*, and so on are message names. The message names are declared to have the type *type* and modifier *modifier*. Any component flavor may specify the method-combination type of a message. The default is **:daemon :base-flavor-last**. If more than one component specifies a method-combination type, these types must be the same.

## Defining Methods for Messages

The Flavor System allows you to dynamically modify methods. If an instance is created and one of its methods is later modified, the instance uses the most recent method.

The **defmethod** macro creates or modifies a flavor's method for a message. The **undefmethod** macro deletes a flavor's method. The syntax for these two macros is as follows:

**defmethod** (*flavor* [*method-type*] *message*) *lambda-list* [*Macro*]
{*declaration* | *documentation*}* {*form*}*

**undefmethod** (*flavor* [*method-type*] *message*) [*Macro*]

When a **defmethod** is executed, the *flavor* argument is given a new method of type *method-type* for the message *message.* If a message of that method type already exists, it is removed. The argument *method-type* must be **:primary**, **:before**, or **:after**. If *method-type* is omitted, it is assumed to be **:primary**.

When an **undefmethod** is executed, the *flavor* argument's method of type *method-type* for the message *message* is deleted. To remove a whopper or a wrapper, the *method-type* should be **:whopper** or **:wrapper** respectively. Otherwise *method-type* must be **:primary**, **:before**, or **:after**. If *method-type* is omitted, it is assumed to be **:primary**. An error is signaled if the flavor *flavor* does not have a method of type *method-type* for the *message* argument.

The *lambda-list* argument has the full generality of a lambda list for a **defun**. (See the chapter "Program Structure" of the *Sun Common Lisp Reference Manual* for more information on lambda lists.)

During the execution of the *form* arguments of a **defmethod**, a special lexical environment is created. The variable **self** is bound to the instance that received the message that caused the invocation of the method. In addition, the instance variables of **self** that are declared in the **defflavor** of *flavor* can be accessed and modified by giving their name. (For more information on lexical environments, see the chapter "Program Structure" of the *Sun Common Lisp Reference Manual.*)

Both **defmethod** and **undefmethod** examine all the existing flavors to find any that have already been compiled and that have the specified flavor as a component flavor. The Flavor System then calculates a new combined method for the *message* argument for each of these flavors.

## Subroutines

The Flavor System lets you access and modify the values of instance variables only from within the *form* arguments of a **defmethod**. Modularity demands that free referencing of the instance variables of a flavor be limited.

This design has one drawback. You may want a subroutine that is called from one or more methods. However, subroutines are defined using **defun** rather than **defmethod**. Thus the subroutine is not able to access or modify the instance variables.

There are two ways to handle this situation. If the subroutine accesses the instance variables but does not modify them, the instance variables can be passed to it as arguments. The alternative is to make the subroutine into another **defmethod**. Its message name can be `:internal-fix-labels`, for example. Any other methods that wish to use the subroutine can send **self** an `:internal-fix-labels` message. The name of the message indicates both what the method does and the fact that it is not intended for public use.

## Forcing a Combined Method to be Recalculated

You can force the Flavor System to recalculate all combined methods that depend on a particular method by calling the function **recompile-flavor**.

This function is used primarily when a method for a message contains a call to a macro. If the definition of the macro is modified, the Flavor System is not informed that the method is now modified and should be recalculated. The function **recompile-flavor** explicitly informs the Flavor System of the modification.

Its syntax is as follows:

**recompile-flavor** *flavor* `&optional` *messages do-dependents* [*Function*]

If the *messages* argument is **nil** or omitted, then all combined methods that depend on one of the *flavor* argument's methods are recalculated. Otherwise *messages* is either a single message name or a list of message names. Only those methods that depend on one of the *flavor* argument's methods for *messages* are recalculated.

If the *do-dependents* argument is non-**nil** or omitted, both the *flavor* argument and any already compiled flavors that have *flavor* as a component flavor have their combined methods for *messages* recalculated.

If *do-dependents* is **nil**, then only *flavor*'s combined methods for *messages* are recalculated. If *flavor* is not a compiled flavor, then no combined methods are modified.

Flavors that are used only as components of other flavors and that are not instantiated on their own are not recompiled because they do not have their own compiled methods.

## Resuming Calculation of Combined Methods

Aborting a calculation while the Flavor System is in the middle of calculating combined methods for a message may leave the Flavor System in an inconsistent state. The function **cleanup-all-flavors** restores the Flavor System to a consistent state by forcing the recalculation and recompilation of all combined method calculations that have been interrupted.

## Inhibiting Calculation

The macro **without-cleaning-flavors** temporarily inhibits calculation of combined methods. It is primarily used when you intend to modify the methods of a particular message for several different flavors. The combined methods for that message are not recalculated after each **defmethod**.

## Compiling a Flavor

At certain times the Flavor System automatically compiles a flavor. This means that it calculates the components list for an instance of that flavor and does error checking. It then creates and compiles all the combined methods for the messages that an instance of that flavor can receive and sets up a hash table to allow rapid access to the methods.

The Flavor System is designed so that only flavors that are to be instantiated are compiled—there is no need to compute the combined methods of a flavor that is only to be a mixin for other flavors.

**Note:** The use of the word "compile" here is a slight misnomer. Compilation here means that a flavor's combined methods, which contain the user-written methods, are calculated only once. The methods for a flavor are not recalculated every time an instance is created.

### When Compilation Occurs

A complete compilation occurs at the following times:

- When the Flavor System tries to create an instance of a flavor, but the flavor has not previously been compiled.
- When a call to the macro **compile-flavor-methods** explicitly forces compilation of a flavor.

### The Steps in Compiling a Flavor

The compiling of a flavor by the Flavor System consists of the following steps, which are not necessarily performed in the order given:

- Calculate the component list of flavors.
- Determine the instance variables.
- Determine the acceptable keyword arguments.
- Determine the list of messages accepted.
- Create and compile the combined methods.

- Check for errors.
  - Verify **:required-instance-variables**.
  - Verify **:required-methods**.
  - Verify **:required-flavors**.

## Detailed Information on Flavor Compilation

Here is more detailed information on the steps involved in the compilation of a flavor.

- Calculate the component list of flavors.

  The list of component flavors is created via a preorder depth-first search, as described in the section "Mixing Flavors." If any of the flavors has an option of the form (**:included-flavors** *flavor1 flavor2* ...), the specified flavors are added to the list of component flavors if they are not already there.

  If new flavors are added because of an **:included-flavors** option, each is checked to see whether more flavors must be added because of their component-flavors list or an **:included-flavors** option.

  Unless one of the component flavors specifies the **:no-vanilla-flavor** option, the flavor **vanilla-flavor** is added to the end of the list of component flavors.

- Determine the instance variables.

  The instance variables of the compiled flavor are the union of the instance variables of each component flavor. If an instance variable name occurs in more than one flavor, the two instance variable names refer to the same instance variable.

- Determine the acceptable keyword arguments.

  Each component flavor is checked to see what keywords it allows. A **defflavor** can do this in one of three ways:

  - The **:initable-instance-variables** option declares that certain instance variables of a flavor can be initialized with a keyword. The keyword for initializing an instance variable *var* is **:var**.

  - The **:settable-instance-variables** option declares that certain instance variables of a flavor can be set. Declaring that a variable can be set also declares that it can be initialized.

  - The **:init-keywords** option specifies that each of its arguments is a keyword that is acceptable to **make-instance**. The flavor should provide an **:init** method to interpret the keyword.

  When you use **make-instance** to create an instance of a flavor, the set of acceptable keywords is the union of acceptable keywords for each component flavor.

- Determine the list of messages accepted.

  The set of messages that an instance accepts is the union of messages for which each component flavor has a method.

- Create and compile the combined methods.

  The combined methods for each message that an instance of a flavor accepts are calculated. See the section "Mixing Flavors" for more information on the way in which the individual methods are combined.

- Check for errors.

  - If any component flavor has a **:required-instance-variables** option, all the variables specified as arguments to this option must be in the set of instance variables.
  - If any component flavor has a **:required-methods** option, each of the messages given as arguments to this option must be in the list of accepted messages.
  - If any of the component flavors has a **:required-flavors** option, all the flavors given as arguments to this option must be on the list of component flavors.

  An error is signaled if any of the conditions described is not satisfied.

## Creating Instances

The **make-instance** function creates instances of a flavor. Its syntax is as follows:

**make-instance** *flavor* {*keyword value*}* [*Function*]

The argument *flavor* is the name of the flavor that is to be instantiated. The remaining arguments are pairs of keywords and values.

The operation of **make-instance** consists of several stages:

- Compile the flavor if necessary.

  The flavor is compiled (see the section "Compiling a Flavor") if it has not yet been compiled. The function **make-instance** needs to know the flavor's component list of flavors, the acceptable keyword arguments, and the instance variables.

- Create the property list.

  The initial property list consists of the pairs of *keyword-value* pairs passed to the **make-instance** function.

  Each component flavor is checked to see if it has a **:default-init-plist** option. If it does, each *keyword-form* pair of the **:default-init-plist** option is checked to determine if the *keyword* component of the pair is already on the property list. If it is, the *keyword-form* pair is ignored. Otherwise the *form* component is evaluated, returning the value *value*. The pair *keyword-value* is added to the property list.

  Each keyword on the final property list must be one of the acceptable keywords.

  For the component flavors that have a **:required-init-keywords** option, each keyword argument must appear as a keyword on the property list.

- Give each instance variable an initial value.

  The following rules govern how an instance variable receives its initial value. The rule that applies first determines the initial value.

  1. For an instance variable *var*, if :var is a keyword on the property list, the *keyword-value* pair is removed from the property list, and the instance variable receives the value *value*.

  2. If one of the flavors on the list of components has a default-value form for the instance variable, that form is evaluated, and the instance variable receives the result as its initial value. If the variable is given a default-value form by more than one component flavor, then the default-value form of the component flavor that is closest to the front of the list is the one

evaluated. (See the section "Defining Flavors" for more information on giving an instance variable a default value.)

3. All other instance variables are set to the default value **`flavors-system::unbound`**.

- Create the instance.
- Send the instance an `:init` message.

  The message is sent only if the instance has a method for handling it. The instance thus initializes itself in a manner specified by the writer of the flavor. The `:init` method is passed the property list that has been created as a single argument. This property list has the form (*keyword1 value1 keyword2 value2* ...).

## The Vanilla Flavor

There are certain standard messages for which every flavor should provide methods. They include the following:

- **:print-self**
- **:describe**
- **:which-operations**
- **:operation-handled-p**
- **:send-if-handles**

The flavor **vanilla-flavor** is a system-provided flavor that supplies useful default methods for these standard messages. Unless a flavor or one of a flavor's component flavors contains the option **:no-vanilla-flavor**, **vanilla-flavor** is automatically added at the end of the list of component flavors that is created when a flavor is compiled.

The methods provided by the vanilla flavor are simply useful defaults. You may write more specific methods for these messages, especially for **:print-self** and **:describe**.

## Printing an Instance

The **:print-self** message is sent to an instance whenever an attempt is made to display the instance. You send an instance a **:print-self** message using the following form:

**send** *instance* **:print-self** *stream prindepth*

The *stream* argument is the output stream to which the displayed representation should be sent. The *prindepth* argument is the current depth of the list structure of the object being displayed. The vanilla flavor method for **:print-self** ignores the last argument and displays something like the following:

`#<Instance` *flavor-name hexadecimal-address*>

The flavor name tells you what kind of object the instance is. The hexadecimal address helps distinguish this object from other objects of the same flavor unless, of course, the Garbage Collector has moved the instance.

If you write your own **:print-self** method, the output by convention should start with #< and end with > so that the Lisp reader will complain if the form is accidentally read back in. You may wish to compare the value of the `prindepth` argument with the value of ***print-level*** and to examine the values of ***print-pretty*** and ***print-escape***. (See the chapter "Input/Output" of the *Sun Common Lisp Reference Manual* for more information on ***print-level***, ***print-pretty***, and ***print-escape***.)

In the following example, the instance created by the **make-instance** function receives a **:print-self** message because of Lisp's read-eval-print loop.

```
> (defmethod (circle-flavor :print-self) (stream prindepth)
     (declare (ignore prindepth))
     (format stream "#<circle of radius ~A, center (~A,~A)>"
           (send self :radius)
           (send self :x-center)
           (send self :y-center)))
:PRINT-SELF
> (make-instance 'circle-flavor :radius 5 :x-center 3 :y-center 7)
#<circle of radius 5, center (3,7)>
```

## Describing an Instance

The **describe** function sends an instance a **:describe** message without arguments to make the instance describe itself. The **:describe** method should describe the instance, displaying whatever information is appropriate. The method should write its output to the stream ***standard-output***.

The **:describe** method is called by **describe** as follows:

**send** *instance* **:describe**

The vanilla flavor method displays the flavor of the instance and the value of each of its instance variables. It returns the value of the *instance* argument.

## Determining Messages Handled by an Instance

The messages **:which-operations**, **:operation-handled-p**, and **:send-if-handles** allow you to find out whether an instance knows how to handle a given message.

You send these messages to an instance using the following forms:

**send** *instance* **:which-operations**

**send** *instance* **:operation-handled-p** *msg*

**send** *instance* **:send-if-handles** *msg* `&rest` *args*

The vanilla flavor methods for these three messages do the following:

- **:which-operations** returns a list of the messages that *instance* has methods for.
- **:operation-handled-p** returns **t** if *instance* has a method for the *msg* message, and **nil** otherwise.
- **:send-if-handles** checks to see if *instance* has a method for the *msg* message. If it does, it calls that method with the arguments it has been passed and returns whatever values that method returns. Otherwise it returns **nil**.

**Warning:** You should not write your own methods for these messages.

The vanilla flavor method for **:operation-handled-p** is equivalent to the following:

```
(defmethod (vanilla-flavor :operation-handled-p) (message)
   (not (null (member message (send self :which-operations)))))
```

The **vanilla flavor** method for **:send-if-handles** is equivalent to the following:

```
(defmethod (vanilla-flavor :send-if-handles) (message &rest args)
    (when (send self :operation-handled-p message)
       (apply #'send self message args)))
```

## Unclaimed Messages

If an instance is sent a message for which it has no method, and it has an **:unclaimed-message** method, the message and its arguments are passed to the **:unclaimed-message** method.

The vanilla flavor does not provide a default method for the message **:unclaimed-message**.

You should provide an **:unclaimed-message** method if you want special handling of unknown messages.

The following is an example:

```
> (defmethod (circle-flavor :unclaimed-message)(msg &rest args)
    (declare (ignore args))
    (format *error-output* "I'm ignoring a message of type ~A~%" msg)
    nil)
:UNCLAIMED-MESSAGE
> (send (make-instance 'circle-flavor) :youve-never-heard-of-this)
I'm ignoring a message of type :YOUVE-NEVER-HEARD-OF-THIS
NIL
```

## *all-flavor-names*

**Purpose:** The variable ***all-flavor-names*** provides a list of all defined flavors.

**Syntax:** ***all-flavor-names*** [*Variable*]

**Remarks:** This variable is an extension to Common Lisp.

**Examples:**

```
> (defflavor new-flavor (x y z) () )
NEW-FLAVOR
> (not (null (member 'new-flavor *all-flavor-names*)))
T
```

## cleanup-all-flavors

**Purpose:** The function **cleanup-all-flavors** causes all aborted computations to be performed again. Aborting the calculation and the compilation of a combined method can cause the Flavor System to be left in an inconsistent state. This function recalculates and recompiles the combined methods.

**Syntax:** **cleanup-all-flavors** [*Function*]

**Remarks:** This function is an extension to Common Lisp.

## compile-flavor-methods

**Purpose:** The macro **compile-flavor-methods** explicitly forces the compilation of a flavor, even if no instances of the flavor exist.

**Syntax:** **compile-flavor-methods** {*flavor*}* [*Macro*]

**Remarks:** The macro **compile-flavor-methods** when seen by the Compiler causes the compilation of each of the *flavor* arguments if necessary. The compiled methods and internal data structures are written out to the binary file. When the binary file is loaded, each of the *flavor* arguments is an already compiled flavor.

Normally, a flavor is compiled the first time an instance of it is created. By explicitly calling **compile-flavor-methods** on a flavor, there is no delay at run-time when an instance is created the first time (unless a method or wrapper has been added).

The **compile-flavor-methods** must appear after the methods, wrappers, whoppers, and component flavors of all the flavors being compiled have been defined.

This macro when seen by the interpreter causes the compilation of the *flavor* arguments if they have not already been compiled.

This macro is an extension to Common Lisp.

**Examples:**

```
> (defflavor flavor1 () () )          ; flavor1 is a trivial flavor.
FLAVOR1
> (defflavor flavor2 () () )          ; flavor2 is also a trivial flavor.
FLAVOR2
> (make-instance 'flavor1)            ; Instantiate flavor1.
;;; Compiling function...assembling...(assembling...)emitting...done.
#<Instance FLAVOR1 444B33>
> (compile-flavor-methods flavor2)    ; Force flavor2 to be compiled.
;;; Compiling function...assembling...(assembling...)emitting...done.
NIL
> (make-instance 'flavor2)            ; There is no recompilation.
#<Instance FLAVOR2 467343>
```

## continue-whopper, lexpr-continue-whopper, continue-whopper-all

**Purpose:** The macros **continue-whopper**, **lexpr-continue-whopper**, and **continue-whopper-all** cause the execution of the next innermost whopper or combined method. They can only appear within the *form* arguments of a whopper.

**Syntax:**

**continue-whopper** {*arg*}* [*Macro*]

**lexpr-continue-whopper** {*arg*}* *arg-list* [*Macro*]

**continue-whopper-all** [*Macro*]

**Remarks:** The macro **continue-whopper** explicitly lists the arguments that are to be passed to the next innermost function.

The macro **lexpr-continue-whopper** treats its last argument as a list. The last argument passed to **lexpr-continue-whopper** appended to the list of all the other arguments is the argument list for the next innermost function.

The macro **continue-whopper-all** passes the identical arguments that were passed to the whopper to the next innermost function.

These macros are extensions to Common Lisp.

**Examples:**

```
;; This is an example of a whopper that returns nil if the first argument
;; is 0; otherwise it lets the combined method execute normally.

> (defflavor my-flavor () ())          ; my-flavor is a trivial flavor.
MY-FLAVOR
> (defmethod (my-flavor :my-message) (x) x)
:MY-MESSAGE
> (defwhopper (my-flavor :my-message)(arg1 &rest args)
      (declare (ignore args))
      (if (eql arg1 0)
          nil
          (continue-whopper-all)))
:MY-MESSAGE
> (setq x (make-instance 'my-flavor))
;;; Compiling function...assembling...(assembling...)emitting...done.
;;; Compiling function...assembling...(assembling...)emitting...done.
#<Instance MY-FLAVOR 4B725B>
> (send x :my-message 0)
NIL
> (send x :my-message 3)
3
```

```
;; You can replace the preceding whopper with a new whopper that takes
;; a list of all its arguments and passes that list as a single argument
;; to the wrapped combined method.

> (defwhopper (my-flavor :my-message)(&rest args)
      (continue-whopper args))
;;; Warning: Redefining MY-FLAVOR-MY-FLAVOR-MY-MESSAGE-WHOPPER-WHOPPER-
PRIMARY
;;; Warning: Redefining MY-FLAVOR-MY-MESSAGE-WHOPPER
:MY-MESSAGE
> (send x :my-message 3 4 5)
(3 4 5)
> (send x :my-message)
NIL
```

**See Also:** **defwhopper**

## defflavor

**Purpose:** The macro **defflavor** creates a new flavor or redefines an already existing flavor.

**Syntax:** **defflavor** *flavor-name* ({*var* | (*var default-form*)}*) ({*flavor*}*) {*option*}* [*Macro*]

**Remarks:** The *flavor-name* argument is the name of the flavor being created or redefined.

The second argument is a list of variable names that are the instance variables of this flavor.

The third argument is a list of flavors that are the component flavors of this flavor. If a default form is specified, it is evaluated as the initial value of the form *var*.

The following arguments are options. Each argument is either the name of an option or a list whose car is the option and whose cdr is a list of arguments.

- **:default-init-plist** adds additional keyword-value pairs to the initial list of keyword-value pairs given to **make-instance**.
- **:gettable-instance-variables** creates methods to access the value of the instance variables that are arguments to this option. If no argument is given, methods are created to access the value of all the instance variables.
- **:included-flavors**. Any instance containing *flavor-name* as a component flavor must also contain each of the arguments to this option as a component flavor. If it does not contain a particular argument to **:included-flavors**, that argument is added to the end of the components list.
- **:init-keywords** declares that the keywords that are its arguments are allowed in the call to **make-instance**.
- **:initable-instance-variables** declares that the instance variables passed as arguments can have their values initialized in the call to **make-instance**. If no argument is given, then all instance variables can have their values initialized.
- **:method-combination**. Each argument to the option is a list of the form

  (*type modifier* {*msg*}*)

  The method combination type for each message *msg* is type *type* with modifier *modifier*.
- **:no-vanilla-flavor** declares that **vanilla-flavor** will not be added to any instance containing *flavor-name* as a component flavor.

- **:required-flavors**. Any instance containing *flavor-name* as a component flavor must also contain each of the arguments to this option as a component flavor.
- **:required-init-keywords**. When an instance containing *flavor-name* as a component is being created, the initial list of keyword-value pairs must have each of the keyword arguments to this option as a keyword.
- **:required-instance-variables**. Any instance that contains *flavor-name* as a component must also have the arguments to this option as instance variables.
- **:settable-instance-variables** creates methods to modify the values of the instance variables that are arguments to this option. If no argument is given, methods are created to modify all the instance variables.

This macro is an extension to Common Lisp.

Examples:

```
;; plist-mixin is the flavor described in the section "Mixing Flavors."

> (defflavor circle-with-plist-flavor
     ((x-center 0)
      (y-center 0)
      radius)
     (plist-mixin)
     :settable-instance-variables
     (:required-methods :getprop :putprop))
CIRCLE-WITH-PLIST-FLAVOR
> (setq x (make-instance 'circle-with-plist-flavor))
;;; Compiling function...assembling...(assem<bling...)emitting...done.
#<Instance CIRCLE-WITH-PLIST-FLAVOR 4EC5B3>
> (send x :x-center)
0
```

## defmethod, undefmethod

**Purpose:** The macro **defmethod** creates or modifies the method that an instance of a flavor uses to handle a message. The macro **undefmethod** deletes a flavor's method for a message of type *method-type*.

**Syntax:** **defmethod** (*flavor* [*method-type*] *message*) *lambda-list* {*declaration* | *documentation*}* {*form*}* [*Macro*]

**undefmethod** (*flavor* [*method-type*] *message*) [*Macro*]

**Remarks:** For both macros, if the *method-type* argument is omitted, it is assumed to be **:primary**.

For **defmethod**, the *method-type* argument must be **:primary**, **:before**, or **:after**.

For **undefmethod**, *method-type* must be **:primary**, **:before**, **:after**, **:whopper**, or **:wrapper**. To delete a whopper, use **undefmethod** with *method-type* **:whopper**. To delete a wrapper, use **defmethod** with *method-type* **:wrapper**.

Both **defmethod** and **undefmethod** cause the Flavor System to recalculate and to recompile the combined method for *message* of all compiled flavors that contain *flavor* as a component.

These macros are extensions to Common Lisp.

**Examples:**

```
> (defmethod (circle-flavor :print-self)(stream prindepth)
     ;; circle-flavor is the flavor defined in the section
     ;; "Introduction to Flavors."
       (declare (ignore prindepth))
       (format stream "#<circle of radius ~A>" radius))
:PRINT-SELF
> (setq x (make-instance 'circle-flavor :radius 3))
#<circle of radius 3>
> (undefmethod (circle-flavor :print-self))
:PRINT-SELF
> x
#<Instance CIRCLE-FLAVOR 56022B>
```

**See Also:** **recompile-flavor**

**without-cleaning-flavors**

## defwhopper

**Purpose:** The macro **defwhopper** creates a whopper to go around the combined method for the message *message* of any flavor that contains the *flavor* argument as a component flavor.

**Syntax:** **defwhopper** (*flavor message*) *lambda-list* {*declaration* | *documentation*}* {*form*}* [*Macro*]

**Remarks:** When a message is sent to an instance that contains the *flavor* argument as a component flavor, the arguments of the message are bound to the variables of the *lambda-list* argument, and the *form* arguments are evaluated. Whatever value is returned by the last *form* is the value of the combined method.

Within the *form* arguments, three special macros are recognized: **continue-whopper**, **lexpr-continue-whopper**, and **continue-whopper-all**. These macros cause the execution of either the next innermost whopper or the original combined method.

This macro is an extension to Common Lisp.

**Examples:**

```
;; The following examples show the use of a whopper to reverse the order of
;; the first two arguments.

> (defflavor new-flavor () () )          ; new-flavor is a trivial flavor.
NEW-FLAVOR
> (setq x (make-instance 'new-flavor))
;;; Compiling function...tail merging...assembling...emitting...done.
#<Instance NEW-FLAVOR 58B15B>
> (defmethod (new-flavor :msg) (&rest args) args)
:MSG
> (send x :msg 1 2 3 4)
(1 2 3 4)
> (defwhopper (new-flavor :msg)(arg1 arg2 &rest args)
     (lexpr-continue-whopper arg2 arg1 args))
;;; Compiling function...assembling...(assembling...)emitting...done.
:MSG
> (send x :msg 1 2 3 4)
(2 1 3 4)
```

**See Also:**

**continue-whopper**

**continue-whopper-all**

**lexpr-continue-whopper**

**undefmethod**

## defwrapper

**Purpose:** The macro **defwrapper** creates a wrapper to go around the combined method for the message *message* of any flavor that contains the *flavor* argument as a component flavor.

**Syntax:** **defwrapper** (*flavor message*) [*Macro*]
(*lambda-list* . *body-var*)
{*declaration* | *documentation*}* {*form*}*

**Remarks:** To calculate the new combined method, the Flavor System first calculates what the combined method would be without the wrapper. The forms are then evaluated with the *body-var* argument bound to a list of length one whose single element is the unwrapped combined method. The value returned by the last form becomes the body of the wrapped combined method. The argument list of the wrapped combined method is the *lambda-list* argument.

Whoppers may be easier to use than wrappers.

This macro is an extension to Common Lisp.

**Examples:**

```
> (defflavor new-flavor () () )         ; new-flavor is a trivial flavor.
NEW-FLAVOR
> (defmethod (new-flavor :message) (x) x)
:MESSAGE
> (setq x (make-instance 'new-flavor))
#<Instance NEW-FLAVOR 5B9893>
> (send x :message 4)
4
> (send x :message :key)
:KEY
> (defwrapper (new-flavor :message)((arg) . body)
     '(when (numberp arg)
         (- (progn ,@ body))))
;;; Compiling function...assembling...(assembling...)emitting...done.
:MESSAGE
> (send x :message 4)
-4
> (send x :message :key)
NIL
```

**See Also:** **undefmethod**

## flavor-allowed-init-keywords, flavor-allows-init-keyword-p

**Purpose:** The function **flavor-allowed-init-keywords** returns a list, which is sorted alphabetically, of all the keywords that are valid keyword options when creating an instance of a flavor. The specified flavor and all flavors that are components of that flavor are checked.

The function **flavor-allows-init-keyword-p** returns **nil** if the function **make-instance** does not allow *keyword* as a keyword option when creating an instance of *flavor*. Otherwise it returns *flavor*.

**Syntax:**

**flavor-allowed-init-keywords** *flavor* [*Function*]

**flavor-allows-init-keyword-p** *flavor keyword* [*Function*]

**Remarks:** These functions are extensions to Common Lisp.

**Examples:**

```
> (defflavor another-flavor
    (var1 var2)
    (circle-flavor)                  ; circle-flavor is the flavor described
    :settable-instance-variables     ; in the section "Introduction
    :gettable-instance-variables     ; to  Flavors."
    :initable-instance-variables
    (:init-keywords :key1 :key2))
ANOTHER-FLAVOR
> (flavor-allowed-init-keywords 'circle-flavor)
;;; Compiling function...tail merging...assembling...emitting...done.
(:RADIUS :X-CENTER :Y-CENTER)
> (flavor-allowed-init-keywords 'another-flavor)
;;; Compiling function...tail merging...assembling...emitting...done.
(:KEY1 :KEY2 :RADIUS :VAR1 :VAR2 :X-CENTER :Y-CENTER)
> (flavor-allows-init-keyword-p 'another-flavor :key1)
ANOTHER-FLAVOR
> (flavor-allows-init-keyword-p 'another-flavor :no-such-key)
NIL
```

## instancep

**Purpose:** The predicate **instancep** tests whether its argument *object* is an instance of a flavor.

**Syntax:** **instancep** *object* [*Function*]

**Remarks:** This function is an extension to Common Lisp.

**Examples:**

```
;; circle-flavor is defined in the section "Introduction to Flavors."

> (instancep (make-instance 'circle-flavor))
T
> (instancep 7)
NIL
```

## make-instance

**Purpose:** The function **make-instance** creates an instance of a flavor. The Flavor System compiles the flavor if it has not already been compiled.

**Syntax:** **make-instance** *flavor* {*keyword value*}* [*Function*]

**Remarks:** The acceptable values for the *keyword* argument depend on the flavor. The function **flavor-allowed-init-keywords** returns a list of the acceptable values for *keyword.*

This function is an extension to Common Lisp.

**Examples:**

```
;; circle-flavor is defined in the section "Introduction to Flavors."

> (describe (make-instance 'circle-flavor
                    :x-center 3 :y-center 7 :radius 10))
An instance of flavor CIRCLE-FLAVOR.
Instance variables:
X-CENTER 3
Y-CENTER 7
RADIUS 10
 #<Instance CIRCLE-FLAVOR 96F363>
```

**See Also:** **flavor-allowed-init-keywords**

## recompile-flavor

**Purpose:** The function **recompile-flavor** recalculates any existing combined method that contains a flavor's methods for certain messages.

**Syntax:** **recompile-flavor** *flavor* `&optional` *messages do-dependents* [*Function*]

**Remarks:** If the *messages* argument is **nil** or omitted, then all combined methods that depend on one of the flavor's methods are recalculated. Otherwise *messages* is either a single message name or a list of message names. Only those methods that depend on one of the flavor's methods for *messages* are recalculated.

If the *do-dependents* argument is non-**nil** or omitted, the specified flavor and any already compiled flavors that have the specified flavor as a component flavor have their combined methods for *messages* recalculated.

If *do-dependents* is **nil**, then only the specified flavor's combined methods for *messages* are recalculated. If the flavor is not a compiled flavor, no combined methods are modified.

This function is an extension to Common Lisp.

## self

**Purpose:** The variable **self** is used by methods to send themselves messages. It is bound to the instance receiving the message that caused the execution of the **defmethod**, the **defwrapper**, or the **defwhopper**.

**Syntax:** **self** [*Variable*]

**Remarks:** The variable **self** is used when a flavor is built on top of component flavors and when an instance wishes to make use of one of the component flavor's methods.

The variable **self** is not a special variable.

This variable is an extension to Common Lisp.

**Examples:**

```
> (defflavor rectangle-flavor
    (width height)
    ()
    :gettable-instance-variables
    :settable-instance-variables
    :initable-instance-variables)
RECTANGLE-FLAVOR
> (defflavor rectangle-with-perimeter-flavor
      ()
      (rectangle-flavor)
      (:required-methods :height :width)
         ;; The :required-methods option is used here to ensure that
         ;; rectangle-flavor has methods for the two messages :height
         ;; and :width.
   )
RECTANGLE-WITH-PERIMETER-FLAVOR
> (defmethod (rectangle-with-perimeter-flavor :perimeter) ()
    (* 2 (+ (send self :height)
            (send self :width))))
:PERIMETER
> (setq x (make-instance 'rectangle-with-perimeter-flavor
              :width 7 :height 5))
#<Instance RECTANGLE-WITH-PERIMETER-FLAVOR 69683B>
> (send x :perimeter)
24
```

## send

**Purpose:** The function **send** sends a message to an instance. Lisp determines what the specified instance's flavor is and finds its combined method for the *message* argument. It then calls that method, passing arguments to the method.

**Syntax:** **send** *instance message* &rest *args* [*Function*]

**Remarks:** This function is an extension to Common Lisp.

**Examples:**

```
;; circle-flavor is defined in the section "Introduction to Flavors."

> (setq a-circle (make-instance 'circle-flavor))
#<Instance CIRCLE-FLAVOR 57581B>
> (send a-circle :set-radius 3)
3
> (send a-circle :diameter)
6
```

## symeval-in-instance, set-in-instance

**Purpose:** The functions **symeval-in-instance** and **set-in-instance** allow you to find or to set the value of a particular instance variable within a particular instance.

The function **symeval-in-instance** finds the value of the instance variable *symbol* in the specified instance.

The function **set-in-instance** sets the value of the instance variable *symbol* of *instance* to the specified *value* argument.

**Syntax:**

**symeval-in-instance** *instance symbol* &optional *no-error-p unbound* [*Function*]

**set-in-instance** *instance symbol value* [*Function*]

**Remarks:** If there is no instance variable whose name is *symbol*, the value of *no-error-p* is examined. If it is **nil**, an error is signaled. If it is non-**nil**, **nil** is returned.

If the instance variable *symbol* is unbound in the instance, then the *unbound* argument (whose default value is **nil**) is returned.

These functions are intended for debugging and may be inefficient. Therefore production code should never use these functions.

These functions are extensions to Common Lisp.

**Examples:**

```
;; circle-flavor is described in the section "Introduction to Flavors."

> (setq a-circle (make-instance 'circle-flavor :x-center 3))
#<Instance CIRCLE-FLAVOR 56C74B>
> (symeval-in-instance a-circle 'x-center)
3
> (symeval-in-instance a-circle 'radius nil 'unknown-radius)
UNKNOWN-RADIUS
> (set-in-instance a-circle 'y-center
                   (symeval-in-instance a-circle 'x-center))
3
> (symeval-in-instance a-circle 'width t)
NIL
```

## without-cleaning-flavors

**Purpose:** The macro **without-cleaning-flavors** is designed to make the modification of several different methods more efficient. It can prevent the same combined method from being repeatedly calculated.

**Syntax:** **without-cleaning-flavors** {*form*}* [*Macro*]

**Remarks:** Each *form* argument is evaluated. No recalculation of combined methods is done until all of the forms have been evaluated.

This macro is an extension to Common Lisp.

# Chapter 12. The Window Tool Kit

# Chapter 12. The Window Tool Kit

## Introduction to the Window Tool Kit

The Window Tool Kit allows you to create and access windows, viewports, and bitmaps, and to access the mouse. It supports both character output and graphic output, and it displays characters in a variety of fonts.

## Initializing the Window Tool Kit

To use the Window Tool Kit, you must first invoke Lisp from within the **suntools** environment.

You must then initialize the Window Tool Kit. The method you choose depends on whether you want to use the Editor in the window environment. The two methods are as follows:

- Using the Editor in the Window Tool Kit

  You normally call the function **ed** to initialize the Window Tool Kit and to invoke the Editor. Type **ed** without any arguments to the top-level prompt:

  ```
  > (ed)
  ```

  The default display that **ed** returns consists of a Lisp Buffer window that occludes a scratch buffer window. The top level of Lisp is accessible through the Lisp Buffer.

  The syntax for **ed** is as follows:

  **ed** `&optional` *x* `&key :windows &allow-other-keys` [*Function*]

  The optional argument *x* is either a pathname that represents a file or a symbol that represents a function definition to be edited. When you are running Lisp from within the **suntools** environment, the value of the **:windows** keyword argument is **t** by default. You may also specify any of the keyword options that are valid for the function **initialize-windows**.

- Using the Window Tool Kit without the Editor

  If you want to use the Window Tool Kit without the Editor, call **initialize-windows** without any arguments:

  ```
  > (initialize-windows)
  ```

  The function **initialize-windows** creates a SunView system window that contains the root viewport. The top level of Lisp is still accessible from the **suntools** Shell window from which you invoked the function.

The syntax for **initialize-windows** is as follows:

```
initialize-windows &key :height :width                    [Function]
                        :screen-x :screen-y
                        :label :icon-image
                        :icon-x :icon-y
                        :icon-label :icon-font
```

See the function page at the end of this chapter for a detailed description of the keyword arguments.

## Restoring Windows in a Saved Image

If the Window Tool Kit has been initialized and the function **disksave** is invoked, the window environment is temporarily suspended. Once the **disksave** function has saved the Lisp image on disk, the windows on the running Lisp image are restored automatically to the state they were in before the **disksave** function call.

To restore the state of the windows in the newly saved image, you may call either **ed** or **initialize-windows** with no arguments. The result of the call depends on whether you were using the Editor in the Window Tool Kit before invoking **disksave**.

- Using the Window Tool Kit without the Editor

  If you used **initialize-windows** to initialize the Window Tool Kit without invoking the Editor in the saved image, you may use either of the following functions to restore the window environment:

  - **initialize-windows**

    If you use this function to restore the windows in the saved image, the windows are restored to the state they were in before the **disksave** function call.

  - **ed**

    If you use this function to restore the windows in the saved image, the windows are restored and the Editor is invoked. However, the Editor buffer windows may cover some or all of the previously initialized windows.

- Using the Editor in the Window Tool Kit

  If you initialized the Window Tool Kit with **ed** before calling **disksave**, the following command restores the window environment in the saved image:

  - **ed**

    Calling this function puts you back into the Editor and restores all previous Editor buffers.

**Note:** Do not use **initialize-windows** to restore windows if you initialized the Window Tool Kit with **ed** before you called **disksave**.

## Basic Structures

The Window Tool Kit has three basic data structures: ***positions***, ***extents***, and ***regions***.

- Positions

  A position is a data structure with two fixnum components, **x** and **y**. A position is specified in Cartesian coordinates in which the **x** component is the distance to the right of the origin, and the **y** component is the distance below the origin.

  The following operations are defined for positions:

| | |
|---|---|
| **make-position** | **position-y** |
| **position-x** | **positionp** |

- Extents

  An extent is a data structure that describes the size of a rectangular area. An extent has two nonnegative fixnum components called **width** and **height**.

  The following operations are defined for extents:

| | |
|---|---|
| **extent-height** | **extentp** |
| **extent-width** | **make-extent** |

- Regions

  A region describes a rectangular area.

  The origin of a region is its top-left point. The corner of a region is the point just below and to the right of its bottom-right point.

  The Window Tool Kit includes functions for accessing the attributes of a region, for finding the corners of a region, for testing containment and equality for regions, and for testing whether a position is inside a region.

  The following operations are defined for regions:

| | |
|---|---|
| **make-region** | **region-size** |
| **region-contains-point-p** | **region-union** |
| **region-contains-position-p** | **region-width** |
| **region-corner** | **region/=** |
| **region-corner-x** | **region<** |
| **region-corner-y** | **region<=** |
| **region-height** | **region=** |
| **region-intersection** | **region>** |
| **region-origin** | **region>=** |
| **region-origin-x** | **regionp** |
| **region-origin-y** | |

## Bitmaps

In the Window Tool Kit, all graphic operations are performed either directly or indirectly on ***bitmaps***. A bitmap is a rectangular array of bits.

Most bitmaps are created with the function **make-bitmap**. The Window Tool Kit also provides functions for accessing bitmap data structures, for setting the width or height of a bitmap, for copying bitmaps, for storing bitmap images in files, and for loading bitmap images from files.

The following operations are defined for bitmaps:

| | |
|---|---|
| **bitblt** | **copy-bitmap** |
| **bitblt-position** | **draw-circle** |
| **bitblt-region** | **draw-line** |
| **bitmap-extent** | **draw-polyline** |
| **bitmap-height** | **draw-polypoint** |
| **bitmap-p** | **load-bitmap** |
| **bitmap-value** | **make-bitmap** |
| **bitmap-width** | **store-bitmap** |
| **charblt** | **stringblt** |
| **clear-bitmap** | **with-fast-drawing-environment** |

## Fonts

A *font* is a set of character images and an associated font name. Each image specifies what the corresponding character looks like when displayed.

Each font has an underlying bitmap that holds the font's character images. Each character's image is a region of the bitmap.

Each character's image in a font has a width. The width is the width of its image's region in the bitmap. Some fonts are fixed width, which means that every character has the same width. Other fonts are variable width, which means, for example, that a "W" may be wider than an "i."

Some characters have an ascender, which is the part of the character above the baseline. Some characters, such as "j" and "q," also have a descender, which is the part of the character below the baseline. The height of a character is the combined height of its ascender and descender. The baseline height of a character is the height of just its ascender.

Every character in a Window Tool Kit font must have the same height and baseline height. Short character images must be padded on the top. Characters with short descenders or no descenders must be padded on the bottom.

## Operations on Fonts

The following operations are defined for fonts:

| | |
|---|---|
| **copy-font** | **font-name** |
| **delete-font** | **fontp** |
| **font-baseline** | **font-set-char** |
| **font-bitmap** | **load-font** |
| **font-clear-char** | **make-font** |
| **font-code-limit** | **rename-font** |
| **font-fixed-width** | **store-font** |
| **font-height** | **string-width** |

## The Font Registry

The Window Tool Kit maintains a registry of all fonts in the window system. This registry lets you access a font by name. Any time you create or load a new font into the window system, the new font is stored in the font registry.

The function **find-font** finds a font in the font registry. Its syntax is the following:

**find-font** *name* [*Function*]

The *name* argument is either a string or a symbol. If *name* is a symbol, it is converted into a string by using its print name. The function **find-font** finds the font whose name is *name* and returns that font. It returns **nil** if it cannot find a font with that name.

Any function that takes a font as an argument can be given a string or a symbol instead. The function **find-font** is called to find the corresponding font.

## Viewports

In the Window Tool Kit, a ***viewport*** is a mapping between a region of a bitmap and a region of the screen. The ***bitmap clipping region*** is the piece of a bitmap that a viewport views. The ***screen clipping region*** is the region of the screen onto which the viewport maps. The screen clipping region and the bitmap clipping region must be the same size.

Whether or not a viewport is actually displayed on the display screen depends on whether the viewport is activated and whether it is occluded (covered up) by other viewports.

The mapping between the bitmap and the screen is as follows: bits in the bitmap that are within the bitmap clipping region are mapped into the viewport's coordinate system by subtracting the origin of the clipping region. These bits are then mapped onto the screen by adding the origin of the viewport's screen clipping region. Bitmap positions that are outside of the bitmap clipping region are undefined under this mapping. Similarly, if a bitmap position maps onto a screen position that is occluded by another viewport, the screen position of the bit is undefined.

Whenever the mouse position lies on top of an unoccluded portion of some viewport, the inverse mapping carries it back to some point of that viewport's corresponding bitmap.

You can reshape viewports and move them around on the screen.

## Creating a Viewport

The following function creates a viewport:

**make-viewport**

## The Viewport Hierarchy

Viewports are arranged in a hierarchy that controls occlusion. The root of the hierarchy is the ***root viewport***, which is created when the Window Tool Kit is initialized. The function **root-viewport** returns the root viewport. The root viewport is a viewport onto a special bitmap that requires less memory but has limited capabilities. You cannot modify the bits of this special bitmap in any way without signaling an error. The root viewport covers the entire screen and cannot be reshaped or moved. All other viewports occlude the root viewport.

Every viewport except the root viewport has a ***parent viewport***. A viewport does not need to lie within its parent's region.

All viewports that are children of one viewport are called ***sibling viewports***. They may overlap on the screen.

Sibling viewports are arranged in a stack. The function **viewport-children** returns a list of a viewport's children in the order that they appear in the sibling stack, with the sibling at the top of the stack appearing at the beginning of the list. The function **expose-viewport** moves a viewport to the top of its sibling stack. The function **hide-viewport** moves a viewport to the bottom of its sibling stack. You can perform more complicated manipulations of the sibling stack using the **setf** macro with the function **viewport-children**.

A viewport may be either active or inactive. A viewport is displayed on the screen only if it is active. A viewport that is inactive is still in the viewport hierarchy, but it is not displayed. If a viewport is inactive, none of its descendants are active.

If two active viewports overlap on the screen, the following rules determine which viewport occludes the other:

- A viewport occludes all of its ancestor viewports.
- If two viewports are siblings, then the viewport that is closest to the top of the sibling stack and all of its descendants occlude the viewport that is farther down and all of its descendants.

### Accessing Viewport Data Structures

The following operations are defined for viewports:

| | |
|---|---|
| **activate-viewport** | **viewport-bitmap** |
| **deactivate-viewport** | **viewport-bitmap-offset** |
| **expose-viewport** | **viewport-bitmap-region** |
| **hide-viewport** | **viewport-bitmap-x-offset** |
| **move-viewport** | **viewport-bitmap-y-offset** |
| **reshape-viewport** | **viewport-children** |
| **root-viewport** | **viewport-parent** |
| **viewport-at-point** | **viewport-screen-region** |
| **viewport-at-position** | **viewportp** |

Any function that takes a bitmap argument can be passed a viewport argument. The function is then performed on the viewport's bitmap.

## Bitmap Output Streams

Because both input and output in Common Lisp are stream oriented, the Window Tool Kit provides a stream-oriented interface to bitmaps, the ***bitmap output stream***, which is an output stream that supports all the Common Lisp character output functions.

Each bitmap output stream maintains an output position that specifies the next available position for writing to the bitmap. You can modify this position.

Each bitmap output stream also maintains a current font and a current linefeed distance. The linefeed distance of a stream is initially the character height of the initial font. If the user does not specify a font, the value of the variable ***default-font*** becomes the initial font.

A bitmap output stream has a default operation for combining new bits with bits already in the bitmap. This operation can be any of the 16 boolean constants that can be the first argument to the function **boole**. The default value is the value of the constant **boole-xor**. (See the chapter "Numbers" in the *Sun Common Lisp Reference Manual* for more information on **boole**.)

The following operations are defined for bitmap output streams:

| | |
|---|---|
| **bitmap-output-stream-p** | **stream-linefeed-distance** |
| **make-bitmap-output-stream** | **stream-operation** |
| **stream-current-font** | **stream-position** |
| **stream-draw-circle** | **stream-x-position** |
| **stream-draw-line** | **stream-y-position** |
| **stream-draw-polyline** | |

A bitmap output stream can be used as the stream argument in any Common Lisp function. (See the chapter "Streams" in the *Sun Common Lisp Reference Manual* for more information on Common Lisp streams.)

Any function that takes a bitmap argument can be passed a bitmap output stream. The operation is performed on the bitmap output stream's underlying bitmap.

## Using the Mouse

The Window Tool Kit provides several ways in which the mouse can be accessed and used in programs. These include polling the mouse, queuing mouse events, and specifying active regions.

### The Mouse Cursor

The position of the mouse is indicated on the screen by a mouse cursor. To specify a cursor, you need to specify a bitmap of the cursor, a boolean operation, and *x*- and *y*-offsets.

The bitmap contains an image of the cursor. The maximum width and height of the bitmap are specified by the constants **maximum-cursor-width** and **maximum-cursor-height** respectively.

The operation specifies the boolean operation that combines the bits already on the screen with the bits of the bitmap. The *x*- and *y*-offsets specify which point of the bitmap should be placed on the screen at the exact location where the mouse is pointing.

Mouse cursors are manipulated by using ***mouse cursor objects***, which are specifications of mouse cursors.

The following functions create and access mouse cursor objects:

| | |
|---|---|
| **current-mouse-cursor** | **mouse-cursor-p** |
| **make-mouse-cursor** | **mouse-cursor-x-offset** |
| **mouse-cursor-bitmap** | **mouse-cursor-y-offset** |
| **mouse-cursor-operation** | |

The function **current-mouse-cursor** returns the mouse cursor object that is currently tracking the mouse on the screen. The **setf** macro can be used with **current-mouse-cursor** to change the cursor.

In addition, the following form allows you to move the cursor:

**move-mouse**

## Polling the Mouse

The most basic way to access the mouse is ***polling***—that is, having a program examine the current state of the mouse. The following functions and variables provide information about the position of the mouse and the status of the mouse buttons:

| | |
|---|---|
| **mouse-buttons** | ***mouse-buttons*** |
| **mouse-x** | ***mouse-x*** |
| **mouse-y** | ***mouse-y*** |

The functions **mouse-x** and **mouse-y** return the current $x$- and $y$-coordinates of the mouse, which are specified in terms of the root viewport. The variables ***mouse-x*** and ***mouse-y*** contain the current $x$- and $y$-coordinates of the mouse relative to the root viewport. These variables are provided for backward compatibility; you should use the corresponding functions in most instances.

The function **mouse-buttons** returns a 3-bit number—that is, a number in the range 0 to 7. If the right mouse button is depressed, the low-order bit is 1. If the middle mouse button is depressed, the middle bit is 1. If the left mouse button is depressed, the high-order bit is 1. This information is summarized in the following table:

| Value | Binary | Meaning |
|---|---|---|
| 0 | 000 | No button is depressed. |
| 1 | 001 | Right button is depressed. |
| 2 | 010 | Middle button is depressed. |
| 3 | 011 | Middle and right buttons are depressed. |
| 4 | 100 | Left button is depressed. |
| 5 | 101 | Left and right buttons are depressed. |
| 6 | 110 | Left and middle buttons are depressed. |
| 7 | 111 | Left, middle, and right buttons are depressed. |

**Figure 12–1. Interpretation of mouse-buttons**

A mouse with two buttons has only a left button and right button, and the middle bit is always 0.

The variable ***mouse-buttons*** contains a 3-bit number that has an interpretation shown in the table above. However, this variable is provided for backward compatibility; you should use the corresponding function in most instances.

## Handling Mouse Events

Queuing ***mouse events*** is a more versatile way to access the mouse.

A mouse event occurs when the mouse is moved or when one of its buttons is pressed or released. Mouse events recognized by the Window Tool Kit are the following:

| | |
|---|---|
| **:mouse-left-down** | **:mouse-enter-region** |
| **:mouse-middle-down** | **:mouse-exit-region** |
| **:mouse-right-down** | **:mouse-moved** |
| **:mouse-left-up** | **:mouse-still** |
| **:mouse-middle-up** | |
| **:mouse-right-up** | |

**Figure 12–2. Mouse Events**

The meaning of each mouse event is summarized below:

- **:mouse-left-down**

  **:mouse-middle-down**

  **:mouse-right-down**

  The corresponding button has been pressed.

- **:mouse-left-up**

  **:mouse-middle-up**

  **:mouse-right-up**

  The corresponding button has been released.

- **:mouse-enter-region**

  **:mouse-exit-region**

  The mouse has entered or exited an active region. (Active regions are discussed in the next subsection.)

- **:mouse-moved**

  The mouse has been moved.

- **:mouse-still**

  The mouse stopped moving approximately half a second ago.

If your mouse has only two buttons, the two mouse events **:mouse-middle-down** and **:mouse-middle-up** cannot occur.

A ***mouse event object*** is a special data structure that is used to encode mouse events. A mouse event object specifies what mouse event has occurred, where the mouse was when the event occurred, and which buttons were pushed at the time.

The following operations are defined for mouse event objects:

| | |
|---|---|
| **mouse-event-buttons** | **mouse-event-x** |
| **mouse-event-event-type** | **mouse-event-y** |
| **mouse-event-p** | |

Special input streams called ***mouse input streams*** can queue both character input and mouse event objects. Characters typed at the terminal and mouse events are queued on a mouse input stream in the order in which they occur.

The following functions create and access mouse input streams:

**make-mouse-input-stream**

**mouse-input-stream-p**

**mouse-input-stream-queue-mouse-events-p**

**mouse-input-stream-viewport**

When a mouse event occurs, the value of the function **mouse-input** is examined; it must be a mouse input stream. You cannot read from a mouse input stream unless the stream is the value of **mouse-input**. If the value of the expression `(mouse-input-stream-queue-mouse-events-p (mouse-input))` is true, then a mouse event object encoding the mouse event is created and queued on the stream. Otherwise the mouse event is ignored, and no mouse event object is created. The **setf** macro can be used with the function **mouse-input** to modify the mouse input stream to which mouse input is sent.

Each mouse input stream is associated with a particular viewport. The values of the functions **mouse-event-x** and **mouse-event-y** for mouse event objects queued on a mouse input stream are relative to the origin of that viewport. The value of **mouse-event-buttons** for mouse event objects is the 3-bit value of ***mouse-buttons*** at the time of the event.

A mouse input stream can be used in the same manner as other Common Lisp input streams. If any Common Lisp input function is used on a mouse input stream, mouse event objects at the front of the queue are removed and discarded. If you call the functions **listen** or **peek-char** on a mouse input stream, mouse events at the beginning of the mouse input stream are lost. (See the chapter "Input/Output" in the *Sun Common Lisp Reference Manual* for more information on **listen** and **peek-char**.)

The following operations are defined for mouse input streams:

| | |
|---|---|
| **listen-any** | **read-any-no-hang** |
| **peek-any** | **unread-any** |
| **read-any** | |

These five functions are similar to the Common Lisp functions **listen**, **peek-char**, **read-char**, **read-char-no-hang**, and **unread-char** respectively. (See the chapter "Input/Output" in the *Sun Common Lisp Reference Manual* for more information.) They differ from their Common Lisp analogues in that they check the input stream for both mouse event objects and characters.

## Active Regions

Specifying ***active regions*** is a third way to access the mouse. Active regions facilitate the creation of menus, scroll bars, and other display objects that interact with the mouse.

An active region is a region that can be attached to a bitmap and that causes that region of the bitmap to become ***mouse sensitive***. If that region of the bitmap is displayed on the display screen and the mouse enters or leaves that region of the screen, the Window Tool Kit's mouse handler calls a method specified by the active region. Similarly, if a mouse event occurs while the mouse is inside an active region displayed on the screen, a method specified by the active region is called. See Figure 12–2 for the possible mouse events.

The following operations are defined for active regions:

| | |
|---|---|
| **active-region-bitmap** | **bitmap-active-regions** |
| **active-region-method** | **clear-bitmap-active-regions** |
| **active-region-p** | **detach-active-region** |
| **attach-active-region** | **make-active-region** |

Any function that can be passed a region can be passed an active region instead.

When a mouse event occurs, the process handling the mouse determines which mouse methods, if any, are invoked. First, the viewport containing the mouse is found. Then, if the viewport's bitmap contains any active regions, they are searched. If the mouse's projected position on the viewport's bitmap is such that it falls inside one or more active regions, then the following rules apply:

- If the mouse has exited an active region, the active region's exit method is invoked.
- If the mouse has entered an active region, the active region's entry method is invoked.
- If the mouse is inside one or more active regions, each active region's method for the event is invoked.

The method is called with the following sequence of arguments:

- The viewport on which the mouse event occurred
- The active region
- The mouse event
- The *x*-coordinate of the position on which the mouse event occurred
- The *y*-coordinate of the position on which the mouse event occurred

The *x*- and *y*-coordinates are given relative to the origin of the active region's bitmap.

For all mouse events except **:mouse-exit-region**, the *x*-coordinate and *y*-coordinate arguments specify a position inside the active region. For **:mouse-exit-region**, the specified position lies outside the active region; it may also lie outside the bitmap.

You can use the macro **with-mouse-methods-preempted** to force the Window Tool Kit to ignore all active regions or to ignore all active regions except those attached to a specific bitmap.

Normally, active region methods and interrupt character methods are executed in the order that they occur, and no method is executed until the code for the previous method has finished. You can use the macro **with-asynchronous-method-invocation-allowed** inside a method to allow the execution of other methods before that method has finished execution.

## Windows

A ***window*** is a composite object that combines the functionality of a bitmap, a viewport, a bitmap output stream, and a mouse input stream. Any function that takes one of these as an argument can take a window as an argument.

The predicates **viewportp**, **bitmap-output-stream-p**, and **mouse-input-stream-p** are true for a window.

Windows are included in the viewport hierarchy and, like viewports, are mappings from a bitmap onto the screen. A window can have a border and a title. The border consists of two parts: a black strip around the edge of the window and a white strip inside the black strip. strip is displayed in the displayed in the background color.

The window's viewport and bitmap output stream write onto the area inside the border.

## Scroll Bars

You can create windows with two ***scroll bars*** by using the options provided for the function **make-window**. Scroll bars do the following:

- They indicate what portion of the bitmap is inside the viewport's bitmap clipping region.
- They let you move the bitmap clipping region with the mouse.

Scroll bars are generally used when a window's bitmap is larger than the bitmap clipping region of the window's viewport. When this is the case, you see only a portion of the bitmap at a time.

Scroll bars are two gray bars—a vertical bar that appears on the right-hand side of the window and a horizontal bar that appears at the bottom of the window. The top and bottom edges of the vertical scroll bar represent the top and bottom edges of the bitmap respectively. Similarly, the left and right edges of the horizontal scroll bar represent the left and right edges of the bitmap respectively.

Within each of the two scroll bars is a "bubble" that contains a double-edged arrow. This bubble represents the position of the bitmap clipping region within the bitmap. If the bubble is near the top of the vertical scroll bar, then the visible portion of the bitmap is near the top of the bitmap. If the bubble is near the center of the horizontal scroll bar, then the visible portion of the bitmap is about halfway between the right and left edges of the bitmap.

If the window's viewport is not smaller in a given dimension than its bitmap, or if it is so small that the bubble would fill an entire scroll bar, then the window will not have a scroll bar in that dimension. The scroll bars will only appear when suitable space constraints are met. The size of the bubble is fixed.

Scroll bars are mouse sensitive. You can move the bitmap clipping region by moving the mouse onto the right or bottom scroll bar. When you move the mouse onto a scroll bar, the mouse cursor changes into a dark block. This new cursor indicates that you can now use the mouse to move the bitmap clipping region.

To move the bitmap clipping region, you use the mouse in one of the following ways:

- If you put the mouse outside the bubble and click the right mouse button, the bubble will move so that its center is approximately where the mouse was clicked. The viewport's bitmap clipping region changes to correspond to the new bubble position.
- You can put the mouse inside the bubble, hold the right mouse button down, and then "drag" the bubble. The bubble moves with the mouse cursor. After you release the mouse button, the viewport's bitmap clipping region changes to correspond to the new bubble position.

Once you move the mouse off either of the scroll bars, the mouse cursor changes back to its former shape.

### Operations on Windows

The following operations are defined for windows:

| | |
|---|---|
| **make-window** | **window-title** |
| **window-frame** | **window-title-font** |
| **window-horizontal-scroll-ratio** | **window-vertical-scroll-ratio** |
| **window-inner-border-width** | **windowp** |
| **window-outer-border-width** | **windows-available-p** |

## Keyboard Input and Interrupt Characters

When a character is typed at the keyboard, that character is sent to the mouse input stream that is the value of the function **keyboard-input**. The **setf** macro can be used to modify the mouse input stream to which characters typed at the keyboard are sent.

Each mouse input stream can have a set of interrupt characters associated with it. When they are typed to the mouse input stream, these interrupt characters do not get queued on the stream. Instead, the Window Tool Kit immediately calls the function that is associated with that character.

The function **mouse-input-stream-interrupt-char** accesses the function that is called when a character is typed to a mouse input stream. Its syntax is the following:

**mouse-input-stream-interrupt-char** [*Function*]
*mouse-input-stream char*

This function returns **nil** if the *char* argument is not an interrupt character on the stream *mouse-input-stream.*

The **setf** macro can be used with **mouse-input-stream-interrupt-char** to modify a character's interrupt function. If you set the value to **nil**, the character is no longer an interrupt character. If you set the value to a function, the character becomes an interrupt character on that mouse input stream.

When an interrupt character is typed on the mouse input stream, the corresponding function is called with these two arguments:

- The mouse input stream that received the character.
- The character.

Normally, active region methods and interrupt character methods are executed in the order that they occur, and no method is executed until the code for the previous method has finished. You can use the macro **with-asynchronous-method-invocation-allowed** inside a method to allow the execution of other methods before that method has finished execution.

## Pop-Up Menus

A pop-up menu is a viewport that is displayed temporarily on the screen and that offers you a set of options. You can either select one of the options by placing the mouse over that item and clicking the right button or make no choice by moving the mouse off the menu. In either case, the pop-up menu then disappears.

This process is divided into two steps. The function **make-pop-up-menu** creates a new pop-up menu object. When the function **pop-up-menu-choose** is passed a pop-up menu object, that menu appears on the screen near the current location of the mouse. The function returns a value that depends on what you choose from the menu. A pop-up menu object can be passed repeatedly to the function **pop-up-menu-choose**.

You can select menu items by pressing any of the mouse buttons that are returned by the function **menu-mouse-buttons**. You can use the **setf** macro with the **menu-mouse-buttons** to specify which buttons activate menu options.

The following operations are defined for pop-up menus:

| | |
|---|---|
| **make-pop-up-menu** | **pop-up-menu-choose** |
| **menu-mouse-buttons** | **pop-up-menu-p** |

## Sample Code

The following sample code could be used to implement pop-up menus.

```
(in-package 'windows)

;;; Define the inner and outer border width for menu windows.

(defparameter *menu-inner-border-width* 2)
(defparameter *menu-outer-border-width* 1)


;;; Define the margins for a selection as a fraction of the font height.

(defparameter *menu-top-margin* 0.1)
(defparameter *menu-bottom-margin* 0.1)
(defparameter *menu-left-margin* 0.1)
(defparameter *menu-right-margin* 0.1)


;;; Define the POP-UP-MENU structure.

;;; When the menu window is finally displayed, the top-left corner
;;; of the window must in the region whose origin is the point
;;; (min-mouse-x, min-mouse-y) and whose corner is the point
;;; (max-mouse-x, max-mouse-y).

(defstruct
  (pop-up-menu
     (:constructor create-pop-up-menu)) ; Function to create a blank pop-up
                                        ; menu object.
  window                                ; The menu window.
  width                                 ; Width of the menu window.
  height                                ; Height of the menu window.
  min-mouse-x
  max-mouse-x
  min-mouse-y
  max-mouse-y
  (state 'not-choosing))                ; Current state of the menu.
```

```
;;; You call the function MAKE-POP-UP-MENU to construct a new
;;; pop-up menu.

(defun make-pop-up-menu (choice-list &optional default-value)
  (declare (special *default-font*))
  (check-type choice-list list)        ; Verify choice-list is a list.
  (let* ((root-viewport (root-viewport))
         (root-bitmap (viewport-bitmap root-viewport))
         (screen-width (bitmap-width root-bitmap))
         (screen-height (bitmap-height root-bitmap))
         (font-height (font-height *default-font*))
         (top-margin (round (* *menu-top-margin* font-height)))
         (bottom-margin (round (* *menu-bottom-margin* font-height)))
         (left-margin (round (* *menu-left-margin* font-height)))
         (right-margin (round (* *menu-right-margin* font-height)))
                                       ; Extra margin to leave on sides.
         (combined-border-width (+ *menu-inner-border-width*
                                   *menu-outer-border-width*))
         (selection-height (+ top-margin font-height bottom-margin))
                                       ; Height of each entry.
         (inner-width 0)               ; Inside width of the menu.
         (menu-width 0)                ; These two variables will
         (menu-height 0))              ; eventually hold the menu's width
                                       ; and height.

    ;; Verify that each item on the choice list is either a symbol
    ;; or a cons whose car is a string.  Also figure the maximum
    ;; width of the strings.
    ;;
    ;; Determine the height of the menu by adding selection-height
    ;; into menu-height for each item seen.
    ;;
    ;; The variable inner-width contains the width of the largest string
    ;; seen so far.


    (dolist (choice choice-list)
      ;; Verify each item.
      (unless (or (symbolp choice)
                  (and (consp choice) (stringp (car choice))))
        (error "Element ~S of choice-list ~S must be a symbol or ~
                a cons whose car is a string" choice choice-list))
      (setq inner-width
            (max inner-width (string-width (if (symbolp choice)
                                               (symbol-name choice)
                                               (car choice))
                                           *default-font*)))
      (incf menu-height selection-height))
```

```
;; The value of menu-width is the sum of the left and right margins and
;; the inner width.  Check to see if the menu is too long or too wide.

(setq menu-width (+ inner-width left-margin right-margin))
(when (> (+ menu-width (* 2 combined-border-width)) screen-width)
  (error
    "Pop-Up Menu for choice-list ~S is wider than the screen"
    choice-list))
(when (> (+ menu-height (* 2 combined-border-width)) screen-height)
  (error
    "Pop-Up Menu for choice-list ~S is taller than the screen"
    choice-list))


;; Create an inactive window.  For now, set viewport-x and
;; viewport-y to 0.  They will be changed by pop-up-menu-choose
;; before the menu is displayed.

;; Once the window is created, you have all the information needed
;; to create the pop-up menu.

;; The variable choice-number will keep track of where you are in
;; the choice-list.

(let* ((window (make-window :viewport-x 0 :viewport-y 0
                            :width menu-width :height menu-height
                            :inner-border-width
                              *menu-inner-border-width*
                            :outer-border-width
                              *menu-outer-border-width*
                            :initial-font *default-font*
                            :activate nil))
       (bitmap (viewport-bitmap window)) ;; The window's bitmap.
       (menu (create-pop-up-menu
               :window window
               :width menu-width
               :height menu-height
               :min-mouse-x (+ combined-border-width
                               (round menu-width 2))
               :max-mouse-x (- screen-width combined-border-width
                               (round menu-width 2) 1)
               :min-mouse-y combined-border-width
               :max-mouse-y (- screen-height combined-border-width
                               menu-height 1)))

       (choice-number 0))
```

```
;; Now attach active regions to the window's bitmap.  Attach one that
;; covers the entire bitmap, so that you will know if the mouse
;; moves off the bitmap.  Also make an active region for each choice
;; item so that you will know if the mouse moves onto or off that
;; choice item, or if the mouse is clicked on that item.

;; First, the active region that covers the entire bitmap.
(make-active-region
   (make-region :x 0 :y 0 :extent (bitmap-extent bitmap))
   :bitmap bitmap

   ;; If you leave the bitmap and the menu's state indicates that
   ;; a value is expected, then set the state to indicate that you
   ;; are returning a value.  THROW the value default-value.  The
   ;; function pop-up-menu-choice will set up the CATCH.

   :mouse-exit-region
   #'(lambda (viewport active-region mouse-event x y)
       (declare (ignore viewport active-region mouse-event x y))
       (when (eq (pop-up-menu-state menu) 'awaiting-choice)
         (setf (pop-up-menu-state menu) 'returning-choice)
         (throw 'pop-up-menu-choose default-value)))
```

```
;; For each item in the choice list, create an active region and
;; write an appropriate string onto the bitmap.  The active region
;; covers an area menu-width wide by selection-height high.
;; The top-left corner of the region is
;; (* choice-number selection-height).  The string is written
;; just a little below and to the right of that.

(dolist (choice choice-list)
  (let ((name (if (symbolp choice)
                  (symbol-name choice)
                  (car choice)))
        (value (if (symbolp choice) choice (cdr choice))))
    (make-active-region
       (make-region :x 0 :y (* choice-number selection-height)
                    :width menu-width :height selection-height)
       :bitmap bitmap

       ;; If you enter the region, invert it by using the bitblt
       ;; function with the operation boole-c1.

       :mouse-enter-region
       #'(lambda (viewport active-region mouse-event x y)
           (declare (ignore mouse-event x y))
           (bitblt-region (viewport-bitmap viewport) active-region
                          (viewport-bitmap viewport) active-region
                          boole-c1))

       ;; If you leave the region, reinvert it so that it will
       ;; return to a normal state.

       :mouse-exit-region
       #'(lambda (viewport active-region mouse-event x y)
           (declare (ignore mouse-event x y))
           (bitblt-region (viewport-bitmap viewport) active-region
                          (viewport-bitmap viewport) active-region
                          boole-c1))

       ;; If the right button is clicked and the menu is waiting
       ;; for a choice, indicate that a choice has been found and
       ;; THROW the appropriate value to the CATCH set up by
       ;; pop-up-menu.

       :mouse-right-down
       #'(lambda (viewport active-region mouse-event x y)
           (declare (ignore viewport active-region mouse-event x y))
           (when (eq (pop-up-menu-state menu) 'awaiting-choice)
             (setf (pop-up-menu-state menu) 'returning-choice)
             (throw 'pop-up-menu-choose value)))
    ;; Write the string out to the bitmap.
    (stringblt bitmap
               (make-position
                 (+ left-margin
                    (truncate (- inner-width
                                 (string-width name *default-font*))
                              2))
                 (+ (* choice-number selection-height)
                    top-margin
                    (font-baseline *default-font*)))
               *default-font*
               name))
  (incf choice-number))

;; Everything is finished.  Return the menu.
menu)))
```

```
;;; The user calls POP-UP-MENU-CHOOSE to display a pop-up menu object.
;;; This function is rather simple.  It exposes the pop-up menu object's
;;; viewport at an appropriate place and then goes to sleep.  One of the
;;; active regions will cause a THROW to 'pop-up-menu-choose when it has
;;; a value.

;;; In this code you may find it curious that a CATCH is being done around
;;; an invocation of SLEEP.  The reason this works is that the interrupt
;;; handler is invoked in much the same way as a function call, and thus
;;; the SLEEP is a dynamically enclosing context, just as if it had called
;;; the active region method.


(defun pop-up-menu-choose (pop-up-menu)
  (declare (special *mouse-x* *mouse-y*))
  (check-type pop-up-menu pop-up-menu)
  (catch 'pop-up-menu-choose
    (let* ((window (pop-up-menu-window pop-up-menu))

           ;; Try to position the top-left corner of the window so that
           ;; the center of the window is where the mouse is right now.
           (desired-x (- *mouse-x*
                         (round (pop-up-menu-width pop-up-menu) 2)))
           (desired-y (- *mouse-y*
                         (round (pop-up-menu-height pop-up-menu) 2)))

           ;; Set actual-x and actual-y so that they are within the
           ;; region defined by min-mouse-x, min-mouse-y, max-mouse-x
           ;; and mouse-max-y.
           (actual-x (min (max (pop-up-menu-min-mouse-x pop-up-menu)
                               desired-x)
                          (pop-up-menu-max-mouse-x pop-up-menu)))
           (actual-y (min (max (pop-up-menu-min-mouse-y pop-up-menu)
                               desired-y)
                          (pop-up-menu-max-mouse-y pop-up-menu))))
(unwind-protect
    (progn
      ;; Position the menu.
      (move-viewport window actual-x actual-y)

      ;; Indicate that this menu wants a value returned.
      (setf (pop-up-menu-state pop-up-menu) 'awaiting-choice)

      ;; Expose it before activating it, to make sure it gets put on
      ;; the screen fully visible.
      (expose-viewport window)
      (activate-viewport window)
      (sleep 1000000))            ; Good night.
(setf (pop-up-menu-state pop-up-menu) 'not-choosing)
(deactivate-viewport window))))
```

## activate-viewport, deactivate-viewport

**Purpose:** The function **activate-viewport** makes the specified viewport and all of its ancestors active. If the optional ***descendants*** argument is non-**nil**, then all of the viewport's descendants are also made active.

The function **deactivate-viewport** makes the specified viewport and all of its descendants inactive. The viewport ***viewport*** maintains its position in the display stack of its siblings. However, the viewport and its descendants do not appear on the screen until they are reactivated.

**Syntax:**

**activate-viewport** ***viewport*** `&optional` ***descendants*** [*Function*]

**deactivate-viewport** ***viewport*** [*Function*]

**Remarks:** If a viewport is active, all of its ancestors are active. If a viewport is inactive, all of its descendants are inactive.

If **deactivate-viewport** tries to deactivate a viewport that is already inactive, nothing happens.

If **activate-viewport** tries to activate a viewport that is already active, nothing happens.

These functions are extensions to Common Lisp.

## active-region-bitmap

**Purpose:** The function **active-region-bitmap** returns the bitmap to which the argument *active-region* is attached. If *active-region* is not attached to a bitmap, it returns **nil**.

**Syntax:** active-region-bitmap *active-region* [*Function*]

**Remarks:** This function is an extension to Common Lisp.

## active-region-method

**Purpose:** The function **active-region-method** accesses the method that is called when a mouse event occurs inside an active region or when the mouse enters or leaves an active region. The function returns **nil** if no method is associated with the event.

**Syntax:** **active-region-method** *active-region event-name* [*Function*]

**Remarks:** The *event-name* argument must be one of the mouse events listed in Figure 12–2.

The **setf** method for this function updates the appropriate method. If you set the value to **nil**, no method is called when the corresponding mouse event occurs.

The method is called with the following sequence of arguments:

- The viewport on which the mouse event occurred
- The active region
- The mouse event
- The *x*-coordinate of the position on which the mouse event occurred
- The *y*-coordinate of the position on which the mouse event occurred

The *x*- and *y*-coordinates are given relative to the origin of the active region's bitmap. For all mouse events except **:mouse-exit-region**, the *x*-coordinate and *y*-coordinate arguments specify a position inside the active region. For **:mouse-exit-region**, the specified position lies outside the active region; it may also lie outside the bitmap.

If the mouse's projected position on the viewport's bitmap falls inside one or more active regions, the following methods are invoked in the order given:

- If the mouse has exited an active region, the active region's exit method is invoked.
- If the mouse has entered an active region, the active region's entry method is invoked.
- If the mouse is inside one or more active regions, each active region's method for the event is invoked.

**Note:** An active region method is called inside the system's interrupt handler; no other user interrupts are permitted while the function is running. If your method has an infinite loop, there is no way to interrupt it.

This function is an extension to Common Lisp.

## active-region-p

**Purpose:** The predicate **active-region-p** tests whether its argument *object* is an active region. It returns true if *object* is an active region.

**Syntax:** **active-region-p** *object* [*Function*]

**Remarks:** This function is an extension to Common Lisp.

## attach-active-region, detach-active-region, bitmap-active-regions, clear-bitmap-active-regions

**Purpose:** The function **attach-active-region** attaches an active region to a bitmap.

The function **detach-active-region** detaches an active region from its bitmap.

The function **bitmap-active-regions** returns a list of all the active regions that are attached to a bitmap.

The function **clear-bitmap-active-regions** detaches all active regions that are attached to a bitmap.

**Syntax:**

**attach-active-region** *bitmap active-region* [*Function*]

**detach-active-region** *active-region* [*Function*]

**bitmap-active-regions** *bitmap* [*Function*]

**clear-bitmap-active-regions** *bitmap* [*Function*]

**Remarks:** When you attempt to attach an active region to a bitmap, the active region must be located in the bitmap.

If **detach-active-region** is called with an active region that is not attached to a bitmap, nothing happens.

These functions are extensions to Common Lisp.

## bitblt, bitblt-position, bitblt-region

**Purpose:** The function **bitblt** copies regions from one bitmap to another.

The function **bitblt-position** is similar to **bitblt**, except that the locations in each bitmap are expressed as positions rather than as *x*- and *y*-coordinates.

The function **bitblt-region** is similar to **bitblt**, except that the arguments explicitly specify the source and destination regions.

**Syntax:**

**bitblt** ***source-bitmap source-x source-y***
***destination-bitmap destination-x destination-y***
***width height operation***
**&key :clipping-region** [*Function*]

**bitblt-position** ***source-bitmap source-position***
***destination-bitmap destination-position***
***width height operation***
**&key :clipping-region** [*Function*]

**bitblt-region** ***source-bitmap source-region***
***destination-bitmap destination-region***
***operation*** [*Function*]

**Remarks:** The arguments ***source-bitmap*** and ***destination-bitmap*** specify the bitmap from which the copying is performed and the bitmap to which the copying is done respectively. They may be the same bitmap.

The ***source-bitmap*** region that is copied is specified by one of the following:

- The ***source-x***, ***source-y***, ***width***, and ***height*** arguments of **bitblt**. The ***source-x*** and ***source-y*** arguments specify the *x*- and *y*-coordinates respectively of the region's origin. The ***width*** and ***height*** arguments specify the region's width and height respectively.
- The ***source-position***, ***width***, and ***height*** arguments of **bitblt-position**. The ***source-position*** argument specifies the position of the region's origin. The ***width*** and ***height*** arguments specify the region's width and height respectively.
- The ***source-region*** argument of **bitblt-region**.

The ***destination-bitmap*** region that is to be modified is specified by one of the following:

- The ***destination-x***, ***destination-y***, ***width***, and ***height*** arguments of **bitblt**. The ***destination-x*** and ***destination-y*** arguments specify the *x*- and *y*-coordinates respectively of the region's origin. The ***width*** and ***height*** arguments specify the region's width and height respectively.
- The ***destination-position***, ***width***, and ***height*** arguments of **bitblt-position**. The ***destination-position*** argument specifies the position of the region's origin. The ***width*** and ***height*** arguments specify the region's width and height respectively.
- The ***destination-region*** argument of **bitblt-region**.

Each position in the source bitmap region is combined with the corresponding position in the destination bitmap region, and the result is stored in the destination bitmap. The new value of the destination bitmap is the value returned when the function **boole** is applied to these three arguments: the ***operation*** argument, the value of the bit at the source bitmap position, and the value of the bit at the destination bitmap position.

The keyword argument **:clipping-region** specifies a region of the destination bitmap. If this keyword argument is given, only the region of the destination region that is located inside the clipping region is modified.

If the ***source-region*** and ***destination-region*** arguments of **bitblt-region** are different widths, the width of the region that is actually copied is the smaller of the two. Similarly, if the ***source-region*** and ***destination-region*** arguments have different heights, the height of the region that is copied is the smaller of the two.

These functions are extensions to Common Lisp.

## bitmap-extent, bitmap-height, bitmap-width

**Purpose:** These functions access and modify information about a bitmap.

The function **bitmap-extent** creates a copy of a bitmap's extent.

The function **bitmap-height** returns the height of a bitmap.

The function **bitmap-width** returns the width of a bitmap.

**Syntax:**

**bitmap-extent** *bitmap* `&optional` *result-extent* [*Function*]

**bitmap-height** *bitmap* [*Function*]

**bitmap-width** *bitmap* [*Function*]

**Remarks:** If a *result-extent* argument is specified for **bitmap-extent**, that extent is modified to the output extent and then returned. Otherwise a new extent is created and returned.

You can use the **setf** macro with these functions. Increasing the width or height of a bitmap causes new area to appear at its boundaries. Decreasing the width or height may cause loss of data.

These functions are extensions to Common Lisp.

**Examples:**

```
> (setq my-bitmap (make-bitmap :width 100 :height 200))
#<Bitmap 100x200 25F391>
> (bitmap-extent my-bitmap)
#<Extent 100x200 25F7A7>
> (bitmap-height my-bitmap)
200
> (bitmap-width my-bitmap)
100
;; Create a 0x0 extent.
> (setq empty-extent (make-extent))
#<Extent 0x0 25F84C>
;; Copy the extent of my-bitmap into empty-extent.
> (bitmap-extent my-bitmap empty-extent)
#<Extent 100x200 25F84C>
;; Now look at the value of empty-extent.
> empty-extent
#<Extent 100x200 25F84C>
```

## bitmap-output-stream-p

**Purpose:** The predicate **bitmap-output-stream-p** tests whether its argument *object* is a bitmap output stream. It returns true if *object* is a bitmap output stream.

**Syntax:** **bitmap-output-stream-p** *object* [*Function*]

**Remarks:** This function is an extension to Common Lisp.

**Examples:**

```
> (bitmap-output-stream-p (make-bitmap-output-stream))
T
> (bitmap-output-stream-p 7)
NIL
```

## bitmap-p

**Purpose:** The predicate **bitmap-p** tests whether its argument *object* is a bitmap. It returns true if *object* is a bitmap.

**Syntax:** **bitmap-p** *object* [*Function*]

**Remarks:** This function is an extension to Common Lisp.

**Examples:**

```
> (bitmap-p (make-bitmap :height 100 :width 200))
T
> (bitmap-p 7)
NIL
```

## bitmap-value

**Purpose:** The function **bitmap-value** returns the value of a bitmap's point at a given *x*-*y* coordinate.

**Syntax:** **bitmap-value** *bitmap x y* [*Function*]

**Remarks:** The result is either 0 or 1.

You can use the **setf** macro with this function to set the value of a point in a bitmap.

This function is an extension to Common Lisp.

**Examples:**

```
;; Create a 100x200 bitmap.
> (setq bmp (make-bitmap :width 100 :height 200))
#<Bitmap 100x200 AEC25B>
;; Look at the value of a point.
> (bitmap-value bmp 23 56)
0
;; Set the point to one.
> (setf (bitmap-value bmp 23 56) 1)
1
;; Look at the value of that point.
> (bitmap-value bmp 23 56)
1
```

## charblt, stringblt

**Purpose:** The function **charblt** paints a character image from a font onto a bitmap.

The function **stringblt** paints a string of character images from a font onto a bitmap.

**Syntax:** **charblt** *bitmap position font char* `&key :operation` [*Function*]

**stringblt** *bitmap position font string* `&key :operation` [*Function*]

**Remarks:** The *font* argument must be a font, a string, or a symbol. If the argument is a string or a symbol, the function **find-font** is called to find the font whose name is the string or symbol.

The **:operation** keyword argument controls how the font is painted onto the bitmap. The new value of the destination bitmap is the value returned by applying the function **boole** to these three arguments: the **:operation** argument, the value of the font's bit, and the value of the destination bitmap position. If the **:operation** keyword argument is omitted or **nil**, the default value is the value of **boole-1**. This default value causes the bits of the font's bitmap to overwrite whatever was previously on the bitmap.

The *position* argument specifies the position at which the character or characters are output. The first character is aligned so that the left-most point of its baseline is at the point given by the *position* argument.

The function **stringblt** cannot handle tabs and other characters that have an ambiguous print representation. It can handle newline and space characters.

These functions are extensions to Common Lisp.

**See Also:** **bitblt**

## clear-bitmap

**Purpose:** The function **clear-bitmap** clears a bitmap. That is, the value of every point in the bitmap is set to 0.

**Syntax:** **clear-bitmap** *bitmap* &optional *region* [*Function*]

**Remarks:** If a *region* argument is specified, only that region of the bitmap is cleared. Otherwise the entire bitmap is cleared.

This function is an extension to Common Lisp.

**Examples:**

```
;; Create a 10x10 bitmap.
> (setq btmp (make-bitmap :width 10 :height 10))
#<Bitmap 10x10 596D5D>
;; Put ones on the diagonal of the bitmap.
> (dotimes (i 10)
    (setf (bitmap-value btmp i i) 1))
NIL
;; A point on the diagonal has a value of one.
> (bitmap-value btmp 3 3)
1
;; A point not on the diagonal has a value of zero.
> (bitmap-value btmp 3 2)
0
;; Clear a region of the bitmap.
> (clear-bitmap btmp (make-region :x 2 :y 2 :height 3 :width 3))
#<Bitmap 10x10 596D5D>
;; Look at a diagonal point that was cleared.
> (bitmap-value btmp 3 3)
0
;; Look at a diagonal point that was not cleared.
> (bitmap-value btmp 9 9)
1
```

## copy-bitmap

**Purpose:** The function **copy-bitmap** copies a bitmap.

**Syntax:** **copy-bitmap** *bitmap* [*Function*]

**Remarks:** The original bitmap and the copy can be modified without affecting each other.

This function is an extension to Common Lisp.

**Examples:**

```
;; Create a 100x200 bitmap.
> (make-bitmap :height 100 :width 200)
#<Bitmap 200x100 5D95EE>
;; Make a copy of the bitmap.
> (copy-bitmap *)
#<Bitmap 200x100 5D9937>
```

**See Also:** **store-bitmap**

## copy-font

**Purpose:** The function **copy-font** copies a font.

The new font is stored in the font registry under the name *new-name*.

**Syntax:** **copy-font** *font new-name* [*Function*]

**Remarks:** The *font* argument should be a font, a string, or a symbol. If the argument is a string or a symbol, the function **find-font** is called to find the font whose name is the string or symbol.

The original font and the copy can be modified without affecting each other.

This function is an extension to Common Lisp.

**Examples:**

```
> *default-font*
#<Fixed-Width-Font ROMAN 4FB3CE>
;; Create a copy of the default font.  Call it "NEWROMAN".
> (copy-font *default-font* 'newroman)
#<Fixed-Width-Font NEWROMAN 5DA2C3>
```

**See Also:** **find-font**

**store-font**

## current-mouse-cursor

**Purpose:** The function **current-mouse-cursor** returns the mouse cursor object that is currently tracking the mouse on the display screen.

**Syntax:** **current-mouse-cursor** *[Function]*

**Remarks:** You can use the macro **setf** with this function to modify the mouse cursor object that is tracking the mouse.

This function is an extension to Common Lisp.

**See Also:** **make-mouse-cursor**

## *default-font*

Purpose: The value of the variable ***default-font*** is used as a default value by the functions **make-bitmap-output-stream** and **make-window**.

Syntax: ***default-font*** [*Variable*]

Remarks: The initial value of ***default-font*** is the font whose name is `"ROMAN"`.

This variable is an extension to Common Lisp.

Examples:

```
;; Create a bitmap output stream.
;; Do not give an :initial-font keyword argument.
> (make-bitmap-output-stream :width 100 :height 200)
#<Output-Stream to #<Bitmap 100x200 1A45D8> 1A49E0>
;; Check to see that the stream's font is *default-font*.
> (eq (stream-current-font *) *default-font*)
T
```

See Also: **find-font**

## default-font-baseline, default-font-code-limit, default-font-height

**Purpose:** The constants **default-font-baseline**, **default-font-code-limit**, and **default-font-height** are default values for the keyword arguments of **make-font**.

The constant **default-font-baseline** is the default value for the baseline height of every character in a font.

The constant **default-font-height** is the default value for the height of every character in a font.

The constant **default-code-limit** is the default value for the number of characters in a font.

**Syntax:**

**default-font-baseline** *[Constant]*

**default-font-code-limit** *[Constant]*

**default-font-height** *[Constant]*

**Remarks:** These constants are extensions to Common Lisp.

**Examples:**

```
;; Create a font. Give no baseline, height, or code limit.
> (setq font (make-font "NEW"))    ; Create a font with no options.
#<Variable-Width-Font NEW AA66E3>
;; Check the value of the font's baseline.
> (= default-font-baseline (font-baseline font))
T
;; Check the value of the font's code limit.
> (= default-font-code-limit (font-code-limit font))
T
;; Check the value of the font's height.
> (= default-font-height (font-height font))
T
;; Note that its width is nil.
> (null (font-fixed-width font))
T
```

**See Also:** **make-font**

## delete-font

**Purpose:** The function **delete-font** deletes a font from the font registry.

**Syntax:** **delete-font** *font* [*Function*]

**Remarks:** The *font* argument must be a font, a string, or a symbol. If it is a string or a symbol, the function **find-font** is called to convert the argument to a font.

The deleted font continues to exist. However, the function **find-font** no longer recognizes its name.

This function is an extension to Common Lisp.

## draw-circle, draw-line, draw-polyline, draw-polypoint

**Purpose:** The function **draw-circle** draws a circle whose center is the position *center* and whose radius is *radius.*

The function **draw-line** draws a line segment from the position *start* to the position *end.*

The function **draw-polyline** takes a sequence of positions *positions* and connects each adjacent pair.

The function **draw-polypoint** takes a sequence of positions *positions* and draws a dot at each one.

**Syntax:**

**draw-circle** *bitmap center radius* [*Function*]
`&key :width :operation`

**draw-line** *bitmap start end* [*Function*]
`&key :width :operation`

**draw-polyline** *bitmap positions* [*Function*]
`&key :width :operation`

**draw-polypoint** *bitmap positions* [*Function*]
`&key :width :operation`

**Remarks:** If the **:width** keyword argument is given, it defines the line width that is used for drawing the line segments and circles. For **draw-polypoint**, the **:width** argument specifies the diameter of the dot. For **draw-circle**, the border is drawn so that its outer edge is at the specified radius; the width must be less than or equal to the radius. If the **:width** keyword argument is omitted or **nil**, the default value 1 is used.

The **:operation** keyword value is used to control how the values that are being written onto the bitmap combine with the values that are already present. If this keyword argument is omitted or **nil**, the default value is the value of the constant **boole-1**; this default value causes the values that are being written onto the bitmap to overwrite whatever was already on the bitmap.

These functions are extensions to Common Lisp.

## expose-viewport, hide-viewport

**Purpose:** The function **expose-viewport** moves a viewport to the top of its sibling stack. Nothing happens if the viewport is already at the top of the stack.

The function **hide-viewport** moves a viewport to the bottom of its sibling stack. Nothing happens if the viewport is already at the bottom of the stack.

**Syntax:**

**expose-viewport** *viewport* [*Function*]

**hide-viewport** *viewport* [*Function*]

**Remarks:** In complex hierarchies, **expose-viewport** may not place the viewport on the screen unoccluded because it may be occluded by its children, or because its parent may be occluded.

If two active viewports overlap on the screen, the following rules determine which viewport occludes the other:

- A viewport occludes all of its ancestor viewports.
- If two viewports are siblings, then the viewport that is closest to the top of the sibling stack and all of its descendants occlude the viewport that is farther down and all of its descendants.

These functions are extensions to Common Lisp.

**See Also:** **viewport-children**

## extent-height, extent-width

**Purpose:** The function **extent-height** returns the height of an extent.

The function **extent-width** returns the width of an extent.

**Syntax:** **extent-height** *extent* [*Function*]

**extent-width** *extent* [*Function*]

**Remarks:** You can use the **setf** macro with these functions to modify the height and width of an extent.

These functions are extensions to Common Lisp.

**Examples:**

```
> (setq x (make-extent 100 200))
#<Extent 100x200 1A4D04>
> (extent-height x)
200
> (setf (extent-width x) 300)
300
> x
#<Extent 300x200 1A4D04>
```

## extentp

**Purpose:** The predicate **extentp** tests whether its argument *object* is an extent. It returns true if *object* is an extent.

**Syntax:** **extentp** *object* [*Function*]

**Remarks:** This function is an extension to Common Lisp.

**Examples:**

```
> (extentp (make-extent))
T
> (extentp 7)
NIL
```

## find-font

**Purpose:** The function **find-font** finds the font whose name is the *name* argument and returns that font. The function returns **nil** if it cannot find a font with that name.

**Syntax:** **find-font** *name* [*Function*]

**Remarks:** The argument must be a string or a symbol. If it is a symbol, its print name is used.

The font registry initially contains three fonts that are named `"ROMAN"`, `"BOLD-ROMAN"`, and `"ITALIC"`.

This function is an extension to Common Lisp.

**See Also:**

**copy-font**

**delete-font**

**load-font**

**make-font**

**rename-font**

**store-font**

## font-baseline, font-height, font-fixed-width

Purpose: The function **font-baseline** returns the baseline height of a font. This number is the baseline height of every character in the font.

The function **font-height** returns the height of a font. This number is the height of every character in the font.

The function **font-fixed-width** returns the width of every character in a font if the font is a fixed-width font; otherwise it returns **nil**.

Syntax: **font-baseline** *font* [*Function*]

**font-height** *font* [*Function*]

**font-fixed-width** *font* [*Function*]

Remarks: The *font* argument must be a font, a string, or a symbol. If the argument is a string or a symbol, the function **find-font** is called to find the font whose name is the string or symbol.

These functions are extensions to Common Lisp.

Examples:

```
;; Create a font.  Specify the font's height, baseline, and fixed width.
> (setq font1
        (make-font "NEW-ROMAN" :height 12 :baseline 12 :fixed-width 7))
#<Fixed-Width-Font NEW-ROMAN A9D46B>
;; Look at the new font's baseline.
> (font-baseline font1)
12
;; Look at the new font's height.
> (font-height font1)
12
;; Look at the new font's width.
> (font-fixed-width font1)
7
;; Create another font, specifying no fixed width.
> (setq font2 (make-font 'blank-font))    ; This is a variable width font.
#<Variable-Width-Font BLANK-FONT A9FB93>
;; Note that the fixed width is nil.
> (font-fixed-width font2)
NIL
```

See Also: **default-font-baseline**

**default-font-height**

**find-font**

## font-bitmap, font-code-limit, font-name

**Purpose:** The function **font-bitmap** returns the underlying bitmap in which a font stores its character images.

The function **font-code-limit** returns the number of character images that can be stored in a font.

The function **font-name** returns the name of a font.

**Syntax:**

**font-bitmap** *font* [*Function*]

**font-code-limit** *font* [*Function*]

**font-name** *font* [*Function*]

**Remarks:** The *font* argument must be a font, a string, or a symbol. If the argument is a string or a symbol, the function **find-font** is called to find the font whose name is the string or symbol.

These functions are extensions to Common Lisp.

**Examples:**

```
;; Create a font, specifying code limit, height, baseline, and fixed width.
> (setq font (make-font 'new-roman :code-limit 256 :height 20
                        :baseline 16 :fixed-width 8))
#<Fixed-Width-Font NEW-ROMAN AA584B>
;; Look at the font's code limit.
> (font-code-limit font)
256
;; Look at the font's name.
> (font-name font)
"NEW-ROMAN"
;; Look at the font's bitmap.
> (font-bitmap font)
#<Bitmap 2048x20 AA4363>
```

**See Also:**

**default-font-code-limit**

**find-font**

**rename-font**

## font-set-char, font-clear-char

Purpose: The function **font-set-char** defines the character image of the character *char* in the font *font*.

The function **font-clear-char** makes the character image of *char* in *font* undefined.

Syntax: **font-set-char** *font char offset* [*Function*]
`&optional` *width bitmap position*

**font-clear-char** *font char* [*Function*]

Remarks: The *char* argument specifies the character whose image in the font bitmap is being defined or cleared. This argument must satisfy the condition that the expression `(char-code` *char*`)` is less than the number of characters in the font. The number of characters in the font is given by the expression `(font-code-limit` *font*`)`.

For both functions, the *font* argument must be a font, a string, or a symbol. If the argument is a string or a symbol, the function **find-font** is called to find the font whose name is the string or symbol.

The *offset* argument specifies that the top-left corner of the character's image in the font's bitmap is the position given by the expression `(make-position` *offset* `0)`.

The *width* argument specifies the width of the character. If the font is fixed-width, then this argument must be omitted, must be **nil**, or must be the same as the width of the font. If the font is variable-width, this argument must be a nonnegative fixnum and cannot be omitted.

The character's height is the height of the given font.

The *bitmap* argument specifies the bitmap from which the image of the character is copied. The image of the character is taken from the region whose origin is specified by the *position* argument and whose width and height are the same as the width and height respectively of the character. If the *position* argument is omitted or **nil**, the origin of the region is the top-left corner of the bitmap.

If the *bitmap* argument is omitted or **nil**, the character's initial image is blank.

These functions are extensions to Common Lisp.

See Also: **find-font**

## fontp

**Purpose:** The predicate **fontp** tests whether its argument *object* is a font. It returns true if *object* is a font.

**Syntax:** **fontp** *object* [*Function*]

**Remarks:** This function is an extension to Common Lisp.

**Examples:**

```
> (fontp *default-font*)
T
> (fontp 7)
NIL
```

## initialize-windows

**Purpose:** The function **initialize-windows** initializes the Window Tool Kit.

**Syntax:** **initialize-windows** `&key :height :width` *[Function]*

```
                    &key :height :width
                         :screen-x :screen-y
                         :label :icon-image
                         :icon-x :icon-y
                         :icon-label :icon-font
```

**Remarks:** The Window Tool Kit is normally initialized when the Editor is first entered. If you are not using the Editor, you must call the function **initialize-windows** to initialize the window system.

The keyword argument pair **:height** and **:width** specifies the size of the SunView system window that will be the root viewport. Because the actual root viewport lies within the SunView borders, it will be slightly smaller than the specified size.

The keyword argument pair **:screen-x** and **:screen-y** indicates the position of the upper left corner of the SunView window.

The keyword argument **:label** names the SunView window.

The keyword argument **:icon-image** names the file that stores the image that will be used as the Lisp screen icon when an initialized window is closed.

The keyword argument pair **:icon-x** and **:icon-y** specifies the position for the icon when the window is closed.

The keyword argument pair **:icon-label** and **:icon-font** denotes the label displayed on the icon and the label's font.

**Note:** Paired keyword arguments must be used together or not at all; when the SunView system receives one of the paired arguments, it expects the other as well.

If you try to initialize the Window Tool Kit but it has already been initialized, nothing happens.

If the Window Tool Kit has been initialized and the function **disksave** is invoked, the window environment is temporarily suspended. Once the **disksave** function has saved the Lisp image on disk, the windows on the running Lisp image are restored automatically to the state they were in before the **disksave** function call. To restore the state of the window environment in the newly saved image, call **initialize-windows** or **ed** with no arguments. (See the section "Restoring Windows in a Saved Image" for more information.)

This function is an extension to Common Lisp.

**See Also:** **leave-window-system**

## keyboard-input

**Purpose:** The function **keyboard-input** determines where keyboard input is sent. Any character typed at the keyboard is sent to the mouse input stream that is the value of this function.

**Syntax:** **keyboard-input** [*Function*]

**Remarks:** The **setf** macro can be used with this function to change the stream to which keyboard input is sent. The second argument to **setf** must be a mouse input stream.

The initial value of this function is the value of ***terminal-io***.

This function is an extension to Common Lisp.

## leave-window-system

**Purpose:** The function **leave-window-system** exits the Window Tool Kit.

**Syntax:** **leave-window-system** [*Function*]

**Remarks:** If you exit the window environment by calling **leave-window-system**, you cannot return to it. If you wish to use the Window Tool Kit after calling this function, you must set up new windows by invoking **ed** or **initialize-windows**.

This function is an extension to Common Lisp.

**See Also:** **initialize-windows**

## listen-any

**Purpose:** The predicate **listen-any** is true if a character or mouse event object can be read from the given mouse input stream; otherwise it is false.

**Syntax:** **listen-any** `&optional` *mouse-input-stream* [*Function*]

**Remarks:** The argument *mouse-input-stream* specifies a mouse input stream. If this argument is omitted or **nil**, the mouse input stream that is the value of the function **mouse-input** is used. If the *mouse-input-stream* argument is **t**, the mouse input stream that is the value of the function **keyboard-input** is used.

If end-of-file occurs, **listen-any** returns **nil**.

This function is an extension to Common Lisp.

**See Also:**

**keyboard-input**

**mouse-input**

**peek-any**

**read-any**

**read-any-no-hang**

## load-bitmap, store-bitmap

**Purpose:** The function **load-bitmap** reads in a bitmap from the file *file-name* and returns the bitmap.

The function **store-bitmap** stores a bitmap into the file *file-name*.

**Syntax:** **load-bitmap** *file-name* [*Function*]

**store-bitmap** *bitmap file-name* [*Function*]

**Remarks:** These functions are extensions to Common Lisp.

**Examples:**

```
;; Create a 10x10 bitmap with a random value at each point.
> (setq bitmap (make-bitmap :width 10 :height 10))
#<Bitmap 10x10 4B3F2B>
> (dotimes (i 10)
     (dotimes (j 10)
        (setf (bitmap-value bitmap i j) (random 2))))
NIL
;; Store this bitmap in a file named "/tmp/temp".
> (store-bitmap bitmap "/tmp/temp")
#P"/tmp/temp"
;; Read the bitmap back in.
> (setq new-bitmap (load-bitmap "/tmp/temp"))
#<Bitmap 10x10 4B65BB>
;; Print a message if the two bitmaps differ at any point.
> (dotimes (i 10)
     (dotimes (j 10)
         (if (/= (bitmap-value bitmap i j)
                 (bitmap-value new-bitmap i j))
             (format t "values differ at ~D ~D~%" i j))))
NIL
```

**See Also:** **copy-bitmap**

## load-font, store-font

**Purpose:** The function **load-font** reads in a font from the file *file-name.* The function returns the font.

The function **store-font** stores a font in the file *file-name.*

**Syntax:** **load-font** *file-name* [*Function*]

**store-font** *font file-name* [*Function*]

**Remarks:** The *font* argument to **store-font** must be a font, a string, or a symbol. If the argument is a string or a symbol, the function **find-font** is called to find the font whose name is the string or symbol. The information stored by **store-font** includes the font's name.

The font that is read in by **load-font** is stored in the font registry.

These functions are extensions to Common Lisp.

**See Also:** **copy-font**

**find-font**

## make-active-region

**Purpose:** The function **make-active-region** creates an active region for the region *region*; as an option it can attach that active region to a bitmap.

**Syntax:**

```
make-active-region region &key :bitmap                       [Function]
                               :mouse-left-down
                               :mouse-left-up
                               :mouse-middle-down
                               :mouse-middle-up
                               :mouse-right-down
                               :mouse-right-up
                               :mouse-moved
                               :mouse-still
                               :mouse-enter-region
                               :mouse-exit-region
```

**Remarks:** The **:bitmap** keyword argument is the bitmap to which this active region should be attached. If this keyword argument is omitted or **nil**, then the active region is not attached to any bitmap. Later it may be attached to a bitmap by using the function **attach-active-region**.

The rest of the keyword arguments specify the methods for each of the ten types of mouse events. The value of each keyword argument must be a function of five arguments. The method is called whenever the corresponding mouse event occurs inside the created active region.

If a mouse event keyword argument is omitted or **nil**, no method is associated with the mouse event. No function is called when the mouse event occurs inside the created active region.

The method is called with the following sequence of arguments:

- The viewport on which the mouse event occurred
- The active region
- The mouse event
- The *x*-coordinate of the position on which the mouse event occurred
- The *y*-coordinate of the position on which the mouse event occurred

The *x*- and *y*-coordinates are given relative to the origin of the active region's bitmap.

For all mouse events except **:mouse-exit-region**, the *x*-coordinate and *y*-coordinate arguments specify a position inside the active region. For **:mouse-exit-region**, the specified position lies outside the active region; it may also lie outside the bitmap.

This function is an extension to Common Lisp.

See Also: **attach-active-region**

## make-bitmap

**Purpose:** The function **make-bitmap** creates a bitmap.

**Syntax:** **make-bitmap** `&key :extent :width :height` [*Function*]

**Remarks:** The width and height of the bitmap are specified by using the **:width** and **:height** keyword arguments or by supplying an extent with the **:extent** keyword argument.

Unspecified dimensions default to 0.

This function is an extension to Common Lisp.

**Examples:**

```
> (make-bitmap :height 100 :width 200)
#<Bitmap 200x100 856EE6>
```

## make-bitmap-output-stream

**Purpose:** The function **make-bitmap-output-stream** creates a bitmap output stream. The bitmap output stream can be attached to an already existing bitmap, or it can be attached to a new bitmap created by this function.

**Syntax:**

```
make-bitmap-output-stream &key :bitmap                          [Function]
                                :extent :width :height
                                :operation
                                :initial-font
```

**Remarks:** The value of the **:bitmap** keyword argument must be a bitmap. The bitmap output stream is attached to that bitmap. If this keyword argument is omitted or **nil**, a new bitmap is created. The new bitmap's size can be specified with either the **:width** and **:height** keyword arguments or with the **:extent** keyword argument (whose value should be an extent). An unspecified width or height defaults to 0.

The **:operation** keyword argument is the boolean operation used by the stream to write onto the bitmap. Its default value is the value of the constant **boole-xor**.

The **:initial-font** keyword argument is the font in which characters are painted onto the bitmap. The value of this keyword argument must be a font, a string, or a symbol. If the argument is a string or a symbol, the function **find-font** is called to find the font whose name is the string or symbol. The default value is the value of ***default-font***.

The stream position of a newly created bitmap output stream is the position whose *x*-coordinate is 0 and whose *y*-coordinate is the baseline height of the initial font. Its linefeed distance is the height of the initial font.

**Note:** Do not attach an output stream to the bitmap that is associated with the root viewport.

This function is an extension to Common Lisp.

Examples:

```
;; Create a 100x200 bitmap and a bitmap output stream to that bitmap.
> (make-bitmap-output-stream :width 100 :height 200)
#<Output-Stream to #<Bitmap 100x200 854AF5> 854EFD>
;; Check that the stream's current font is *default-font*.
> (eq (stream-current-font *) *default-font*)
T
;; Create another 100x200 bitmap.
> (setq btmp (make-bitmap :width 100 :height 200))
#<Bitmap 100x200 85603A>
;; Create a bitmap output stream that writes onto that bitmap.
> (make-bitmap-output-stream :bitmap btmp)
#<Output-Stream to #<Bitmap 100x200 85603A> 85646A>
```

See Also: **find-font**

## make-extent

**Purpose:** The function **make-extent** creates an extent whose width is *width* and whose height is *height*.

**Syntax:** **make-extent** `&optional` *width height* [*Function*]

**Remarks:** The arguments to **make-extent** are fixnums. If either argument is omitted, the default value 0 is used.

This function is an extension to Common Lisp.

**Examples:**

```
> (make-extent)
#<Extent 0x0 855B5B>
> (make-extent 100 200)
#<Extent 100x200 855B80>
```

## make-font

**Purpose:** The function **make-font** creates a font whose name is *name*.

**Syntax:** **make-font** *name* `&key :bitmap :bitmap-width :code-limit :fixed-width :height :baseline` [*Function*]

**Remarks:** The *name* argument is either a string or a symbol. If it is a symbol, the symbol's print name is used. The newly created font is stored in the font registry under this name.

All characters in the created font must have the same height and baseline height. The font can have either a fixed width or a variable width.

The keyword argument **:code-limit** specifies the maximum number of characters in the font. If this keyword argument is omitted, the value of **default-font-code-limit** is used.

The keyword argument **:baseline** gives the baseline height of every character in the font. If this keyword argument is omitted, the value of **default-font-baseline** is used.

The keyword argument **:height** gives the height of every character in the font. If this keyword argument is omitted, the value of **default-font-height** is used.

The keyword argument **:fixed-width** gives the fixed width of every character in the font. If the keyword argument is omitted or **nil**, the font has a variable width.

The **:bitmap** keyword argument specifies a bitmap to be used as the font bitmap. If this keyword argument is omitted or **nil**, a new bitmap is created for this font. The width and height of the created bitmap are specified by the value of the **:bitmap-width** keyword argument and the font's height. For fixed-width fonts, if the **:bitmap-width** keyword argument is omitted, the default bitmap width is the fixed width (specified by the **:fixed-width** keyword argument) times the number of characters in the font (specified by the **:code-limit** keyword argument). For variable-width fonts, there is no default bitmap width; either the **:bitmap** or **:bitmap-width** keyword argument must be specified.

There is no initial assignment of character images to characters. This assignment must be done with the function **font-set-char**.

This function is an extension to Common Lisp.

Examples:

```
;; Create a variable-width font named "NEW-ROMAN".
> (make-font 'new-roman)
#<Variable-Width-Font NEW-ROMAN A97E03>
;; Create a fixed-width font named "New-font".
> (make-font "New-font" :height 12 :baseline 10 :fixed-width 6)
#<Fixed-Width-Font New-font A98603>
```

See Also:

**default-font-baseline**

**default-font-code-limit**

**default-font-height**

**find-font**

**font-set-char**

## make-mouse-cursor, maximum-cursor-height, maximum-cursor-width

**Purpose:** The function **make-mouse-cursor** creates a mouse cursor object.

The constants **maximum-cursor-height** and **maximum-cursor-width** are hardware constraints on the maximum height and width respectively of a mouse cursor.

**Syntax:** **make-mouse-cursor** *bitmap* `&key :x-offset :y-offset :operation` [*Function*]

**maximum-cursor-height** [*Constant*]

**maximum-cursor-width** [*Constant*]

**Remarks:** The *bitmap* argument is a bitmap that contains an image of the cursor.

The **:x-offset** and **:y-offset** keyword arguments specify which point of the bitmap should be placed on the screen at the exact location where the mouse is pointing. If either keyword argument is omitted or **nil**, the default value 0 is used.

The **:operation** keyword argument specifies the boolean operation that combines the bits of the mouse cursor's bitmap with the bits already on the screen. If this keyword argument is omitted or **nil**, it defaults to the value of the constant **boole-or**.

The value of the expression `(bitmap-height` *bitmap*`)` must be less than or equal to **maximum-cursor-height**.

The value of the expression `(bitmap-width` *bitmap*`)` must be less than or equal to **maximum-cursor-width**.

The function **make-mouse-cursor** and the constants **maximum-cursor-height** and **maximum-cursor-width** are extensions to Common Lisp.

**See Also:** **current-mouse-cursor**

## make-mouse-input-stream

Purpose: The function **make-mouse-input-stream** creates a mouse input stream. A mouse input stream is identical to a Common Lisp input stream except that a mouse input stream can queue both characters and mouse event objects.

Syntax:

```
make-mouse-input-stream &key :queue-mouse-events-p          [Function]
                             :viewport
```

Remarks: The **:queue-mouse-events-p** keyword argument determines whether this mouse input stream initially queues mouse event objects. The default value for this keyword argument is **nil**, which means that only characters are queued on the newly created mouse input stream.

The **:viewport** keyword argument is the viewport associated with the mouse input stream that is being created. If this keyword argument is omitted or **nil**, the mouse input stream is associated with the root viewport.

This function is an extension to Common Lisp.

See Also: **listen-any**

**mouse-input-stream-queue-mouse-events-p**

**peek-any**

**read-any**

**unread-any**

## make-pop-up-menu

**Purpose:** The function **make-pop-up-menu** creates a pop-up menu object.

**Syntax:** **make-pop-up-menu** *choice-list* `&optional` *default-value* [*Function*]

**Remarks:** The argument *choice-list* is a list. Each element of the list is either a symbol or a cons whose car is a string.

If the element is a symbol, then when the function **pop-up-menu-choose** displays the pop-up menu, the print name of the element is displayed as one of the choices. If chosen, the element is returned as the value of **pop-up-menu-choose**.

If the element is a cons, then the car of the element, which must be a string, is displayed as one of the choices. If the element is chosen, the value of **pop-up-menu-choose** is the cdr of the cons.

If the mouse is moved off the choice menu, the *default-value* argument is returned. If this argument is omitted, the default value is **nil**.

This function is an extension to Common Lisp.

**See Also:** **pop-up-menu-choose**

## make-position

**Purpose:** The function **make-position** creates a position. The coordinates of this position are the *x* and *y* arguments.

**Syntax:** **make-position** `&optional` *x* *y* [*Function*]

**Remarks:** The arguments must be nonnegative fixnums. If either argument is omitted, the default value 0 is used.

This function is an extension to Common Lisp.

**Examples:**

```
> (make-position)
#<Position (0,0) 855C2D>
> (make-position 100 200)
#<Position (100,200) 855C3E>
```

## make-region

**Purpose:** The function **make-region** creates a new region.

**Syntax:**

```
make-region &key :origin :x :y                                    [Function]
                 :extent :width :height
                 :corner :corner-x :corner-y
```

**Remarks:** To create a region, you must specify two of the following three attributes of a region: its origin, its corner, and its size. A region's origin is its top-left position. A region's corner is the point just below and to the right of its bottom-right position. A region's size is its height and width.

You specify the origin of a region by specifying the position of the origin with the **:origin** keyword argument or by specifying the *x*- and *y*-coordinates of the origin separately with the **:x** and **:y** keyword arguments.

You specify the corner of a region by specifying the position of the corner with the **:corner** keyword argument or by specifying the *x*- and *y*-coordinates of the corner separately with the **:corner-x** and **:corner-y** keyword arguments.

You specify the size of a region by specifying the region's extent with the **:extent** keyword argument or by specifying the width and height of the region separately with the **:width** and **:height** keyword arguments.

This function is an extension to Common Lisp.

**Examples:**

```
;; You can specify a region whose origin is the point (400,500)
;; and whose corner is the point (480,590) in several different ways.

;; mid-screen is the position of the origin.
> (setq mid-screen (make-position 400 500))
#<Position (400,500) 855D50>
;; ext is the size of the region.
> (setq ext (make-extent 80 90))
#<Extent 80x90 855D64>
;; Give the origin and size of the region.
> (setq reg1 (make-region :origin mid-screen :extent ext))
#<Region 80x90 at (400,500) 855D84>
;; Give the origin and size but specify each coordinate separately.
> (setq reg2 (make-region :x 400 :y 500 :width 80 :height 90))
#<Region 80x90 at (400,500) 855DBE>
```

```
;; Give the size and the corner.
> (setq reg3 (make-region :extent ext :corner (make-position 480 590)))
#<Region 80x90 at (400,500) 855DEB>
;; Verify that all three regions specify the same region.
> (region= reg1 reg2 reg3)
T
```

## make-viewport

**Purpose:** The function **make-viewport** creates a viewport. The viewport is attached to an already existing bitmap or to a newly created bitmap.

The function returns two values: the newly created viewport and the bitmap to which the viewport is attached.

**Syntax:**

```
make-viewport &key :bitmap :width :height                    [Function]
                   :bitmap-region
                   :parent :fixed
                   :screen-position
                   :screen-x :screen-y
                   :activate
```

**Remarks:** The keyword options to this function are described as follows:

- **:bitmap**

  This keyword argument specifies the bitmap to which the viewport is attached. Its value must be a bitmap made with the function **make-bitmap**.

  If this keyword argument is omitted or **nil**, the viewport is attached to a new bitmap whose dimensions are specified by **:width** and **:height**.

- **:width, :height**

  These keyword arguments specify the width and height of the bitmap to which the viewport is attached. The value of each must be a nonnegative fixnum. If either is omitted or **nil**, its default value is 0.

  You only need to use **:width** and **:height** if **:bitmap** is omitted or **nil**.

- **:bitmap-region**

  This keyword argument specifies the viewport's bitmap clipping region. Its value must be a region made with the function **make-region**.

  If this keyword argument is omitted or **nil**, the bitmap clipping region is the entire bitmap; thus, the viewport and bitmap have the same size.

- **:parent**

  This keyword argument specifies the parent viewport of the new viewport. Its value must be an existing viewport. If it is omitted or **nil**, the root viewport becomes the parent viewport. The new viewport is put at the top of its sibling stack.

- **:fixed**

  This keyword argument specifies whether the viewport can be moved independently of its parent. If it is omitted or **nil**, the viewport can be moved independently. If it is non-**nil**, the viewport cannot be moved independently; the Window Tool Kit assumes that the area of the parent viewport that the child viewport obscures need not be saved, since it can never be exposed.

- **:screen-position**

  This keyword argument specifies the position of the viewport's top-left corner. Its value must be a position made with the function **make-position**.

  The **:screen-x** and **:screen-y** keyword arguments can be used as an alternative to **:screen-position**. The default value for the $x$- and $y$-coordinates is 0.

- **:screen-x, :screen-y**

  These keyword arguments specify the coordinates of the viewport's top-left corner relative to the root viewport. The value of each must be a nonnegative fixnum. If either is omitted, its default value is 0.

- **:activate**

  This keyword argument specifies whether the viewport is active or inactive. If it is omitted or non-**nil**, the viewport is active. If it is specified and **nil**, the viewport is inactive.

Note: A viewport's screen clipping region is the region whose top-left corner is the point specified by either **:screen-position** or **:screen-x** and **:screen-y**, and whose extent is the same as that of the viewport's bitmap clipping region.

This function is an extension to Common Lisp.

Examples:

```
;; To run this example, you must have already initialized the
;; Window Tool Kit.

;; Create a 100x200 bitmap and a viewport onto that bitmap.
> (make-viewport :width 100 :height 200)
#<Viewport 100x200 at (0,0) onto #<Bitmap 100x200 85516A> 855572>
#<Bitmap 100x200 85516A>
;; Create another 100x200 bitmap.
> (setq btmp (make-bitmap :width 100 :height 200))
#<Bitmap 100x200 85560C>
;; Create a viewport onto that bitmap.  Note that the bitmap is returned
;; as the second value.
> (make-viewport :bitmap btmp)
#<Viewport 100x200 at (0,0) onto #<Bitmap 100x200 85560C> 855A2A>
#<Bitmap 100x200 85560C>
```

## make-window

**Purpose:** The function **make-window** creates and returns a window.

A window combines the functionality of a viewport, a bitmap, a bitmap output stream, and a mouse input stream. On the display screen, a window appears as a viewport. It may be surrounded by a border and may have a title. A window may also have a scroll bar.

**Syntax:**

```
make-window &key :position :x :y                                   [Function]
                 :extent :width :height
                 :viewport-x :viewport-y
                 :inner-border-width :outer-border-width
                 :viewport-width :viewport-height
                 :initial-font :operation
                 :title :title-font
                 :parent :scroll :activate
                 :calculate-vertical-scroll-ratio
                 :calculate-horizontal-scroll-ratio
                 :vertical-scroll :horizontal-scroll
```

**Remarks:** The keyword options to this function are described as follows:

- **:position**

  This keyword argument specifies the position of the window's top-left corner. Its value must be a position made with the function **make-position**.

  The **:x** and **:y** keyword arguments can be used as an alternative to **:position**. The default value for the *x*- and *y*-coordinates is 0.

- **:x, :y**

  These keyword arguments specify the coordinates of the window's top-left corner relative to the root viewport. The value of each must be a nonnegative fixnum. If either is omitted, its default value is 0.

- **:extent**

  This keyword argument specifies the size of the window's bitmap. Its value must be an extent made with the function **make-extent**.

  The **:width** and **:height** keyword arguments can be used as an alternative to **:extent**.

- **:width, :height**

  These keyword arguments specify the size of the window's bitmap. The value of each must be nonnegative fixnum. If either is omitted or **nil**, its default value is 0.

- **:viewport-x, :viewport-y**

  These keyword arguments specify the coordinates of the top-left corner of the window's viewport.

- **:inner-border-width, :outer-border-width**

  These keyword arguments specify the width of the window's inner and outer borders. The inner border is strip of white space that surrounds the viewport, and the outer border is a black box that surrounds the inner border. If **:inner-border-width** is omitted or **nil**, its default value is 2. If **:outer-border-width** is omitted or **nil**, its default value is 1.

- **:viewport-width, :viewport-height**

  These keyword arguments specify the width and height of the window's viewport. These dimensions can be different from those of the window's bitmap (specified with either **:width** and **:height** or **:extent**). However, if **:viewport-width** and **:viewport-height** are omitted or **nil**, their default values are the width and height of the bitmap respectively.

  The total width of the window is the width of the viewport plus twice the thickness of the inner border plus twice the thickness of the outer border. The total height of the window is the height of the viewport plus twice the thickness of the inner border plus twice the thickness of the outer border plus the height of the title.

- **:initial-font**

  This keyword argument specifies the initial font used by the window's bitmap output stream. Its value must be a font, a string, or a symbol. If it is omitted or **nil**, its default value is the value of the variable ***default-font***.

- **:operation**

  This keyword argument specifies the boolean operation that the bitmap output stream uses to write onto the bitmap. Its value must be an acceptable first argument to the **boole** function. If it is omitted or **nil**, its default value is the value of the constant **boole-xor**.

- **:title**

  This keyword argument specifies the title of the window. Its value must be a string. If the window has a title, it appears in a title bar at the top of the window. If this keyword argument is omitted or **nil**, the window has no title.

- **:title-font**

  This keyword argument specifies the font in which the title is displayed. Its value must be a font, a string, or a symbol. If it is omitted or **nil**, its default value is the value of the **:initial-font** keyword argument.

- **:parent**

  This keyword argument specifies the parent viewport of the new viewport. Its value must be an existing viewport. If it is omitted or **nil**, the root viewport becomes the parent viewport. The new viewport is put at the top of its sibling stack.

- **:activate**

  This keyword argument specifies whether the viewport is active or inactive. If it is omitted or non-**nil**, the viewport is active. If it is **nil**, the viewport is inactive.

- **:scroll**

  This keyword argument specifies whether the window has scroll bars. If it is **t**, the window is created with scroll bars on the right and bottom. If it is omitted or **nil**, the window is created without scroll bars. Do not give this keyword any value other than **t** or **nil**.

- **:calculate-vertical-scroll-ratio, :calculate-horizontal-scroll-ratio**

  These keyword arguments calculate the vertical and horizontal scroll ratio respectively. The scroll ratio is a Common Lisp ratio between 0 and 1. Generally, the scroll ratio is the ratio of the current location of the window to the size of the window's underlying bitmap. However, window system developers may redefine the methods for scrolling and for calculating these ratios so that scrolling may be performed over an abstract bitmap or extent. If either keyword argument is specified, it must be a function that takes the window as an argument. The functions cannot be used with the macro **setf** to specify the respective ratios; they can only return a ratio or **nil**.

- **:vertical-scroll, :horizontal-scroll**

  These keyword arguments replace the default scrolling methods for the window. If either is given, it must be a function that takes two arguments: the window to be scrolled and a vertical or horizontal scroll ratio that describes the location of scrolling.

## make-window

This function is an extension to Common Lisp.

Examples:

```
;; To run this example, you must have already initialized the
;; Window Tool Kit.

> (setq w (make-window :width 100 :height 200 :title "hello"))
#<WINDOW 4AC0AB>
> (windowp w)
T
```

See Also:

**window-vertical-scroll-ratio**

**window-horizontal-scroll-ratio**

## menu-mouse-buttons

**Purpose:** The function **menu-mouse-buttons** returns a list of keywords that indicate which mouse buttons select menu items.

**Syntax:** **menu-mouse-buttons** *[Function]*

**Remarks:** The default list returned by **menu-mouse-buttons** is `(:left :middle :right)`.

When you press a mouse button that is included in the list returned by this function, a menu item is selected. If the mouse button is not listed, no selection is made.

You can use the **setf** macro with this function to set the mouse buttons that select menu items.

This function is an extension to Common Lisp.

## mouse-buttons, mouse-x, mouse-y

**Purpose:** The function **mouse-buttons** returns an integer in the range 0 to 7. This integer encodes the mouse buttons that are currently depressed.

The functions **mouse-x** and **mouse-y** return the current *x*- and *y*-coordinates of the mouse. These positions are relative to the root viewport.

**Syntax:**

**mouse-buttons** [*Function*]

**mouse-x** [*Function*]

**mouse-y** [*Function*]

**Remarks:** See Figure 12–1 for the possible values returned by **mouse-buttons**.

These functions are extensions to Common Lisp.

**See Also:**

***mouse-buttons***

***mouse-x***

***mouse-y***

**mouse-event-buttons**

**mouse-event-x**

**mouse-event-y**

**move-mouse**

## *mouse-buttons*, *mouse-x*, *mouse-y*

**Purpose:** The variable ***mouse-buttons*** contains an integer in the range 0 to 7. This integer encodes the mouse buttons that are currently depressed.

The variables ***mouse-x*** and ***mouse-y*** contain the current *x*- and *y*-coordinates of the mouse. These positions are relative to the root viewport.

**Syntax:**

***mouse-buttons*** [*Variable*]

***mouse-x*** [*Variable*]

***mouse-y*** [*Variable*]

**Remarks:** See Figure 12–1 for the possible values of ***mouse-buttons***.

**Note:** These variables are provided for backward compatibility; you should use the functions **mouse-buttons**, **mouse-x**, and **mouse-y** in most instances.

These variables are extensions to Common Lisp.

**See Also:**

**mouse-buttons**

**mouse-event-buttons**

**mouse-event-x**

**mouse-event-y**

**mouse-x**

**mouse-y**

**move-mouse**

## mouse-cursor-bitmap, mouse-cursor-x-offset, mouse-cursor-y-offset, mouse-cursor-operation

| | |
|---|---|
| Purpose: | These functions return components of a mouse cursor object. |
| | The function **mouse-cursor-bitmap** returns a mouse cursor object's bitmap. |
| | The function **mouse-cursor-x-offset** returns a mouse cursor object's *x*-offset. |
| | The function **mouse-cursor-y-offset** returns a mouse cursor object's *y*-offset. |
| | The function **mouse-cursor-operation** returns the boolean constant that is used to write the mouse cursor object onto the display screen. |
| Syntax: | **mouse-cursor-bitmap** *mouse-cursor-object* [*Function*] |
| | **mouse-cursor-x-offset** *mouse-cursor-object* [*Function*] |
| | **mouse-cursor-y-offset** *mouse-cursor-object* [*Function*] |
| | **mouse-cursor-operation** *mouse-cursor-object* [*Function*] |
| Remarks: | You can use the **setf** macro with these functions to modify a mouse cursor object. |
| | These functions are extensions to Common Lisp. |

## mouse-cursor-p

**Purpose:** The predicate **mouse-cursor-p** tests whether its argument *object* is a mouse cursor object. It returns true if *object* is a mouse cursor object.

**Syntax:** **mouse-cursor-p** *object* [*Function*]

**Remarks:** This function is an extension to Common Lisp.

## mouse-event-p

**Purpose:** The predicate **mouse-event-p** tests whether its argument *object* is a mouse event object. It returns true if *object* is a mouse event object.

**Syntax:** **mouse-event-p** *object* [*Function*]

**Remarks:** This function is an extension to Common Lisp.

## mouse-event-x, mouse-event-y, mouse-event-event-type, mouse-event-buttons

**Purpose:** These functions access the fields of a mouse event object.

The functions **mouse-event-x** and **mouse-event-y** give the *x*- and *y*-coordinates of the mouse when the mouse event occurred that created the mouse event object. These coordinates are relative to the viewport that owns the mouse input stream on which the mouse event object was read.

The function **mouse-event-event-type** returns a keyword that indicates what mouse event created a particular mouse event object. See Figure 12–2 for a list of the possible mouse events.

The function **mouse-event-buttons** returns the value of the variable ***mouse-buttons*** at the time the mouse event occurred. See Figure 12–1 for the values of ***mouse-buttons*** and their meanings.

**Syntax:**

| | |
|---|---|
| **mouse-event-x** *mouse-event-object* | [*Function*] |
| **mouse-event-y** *mouse-event-object* | [*Function*] |
| **mouse-event-event-type** *mouse-event-object* | [*Function*] |
| **mouse-event-buttons** *mouse-event-object* | [*Function*] |

**Remarks:** These functions are extensions to Common Lisp.

## mouse-input

**Purpose:** The function **mouse-input** determines where mouse input is sent.

**Syntax:** **mouse-input** [*Function*]

**Remarks:** The value of the function **mouse-input** is examined when a mouse event occurs. If the expression

```
(mouse-input-stream-queue-mouse-events-p (mouse-input))
```

is non-**nil**, an object encoding the mouse event is appended to the mouse input stream that is the value of the expression `(mouse-input)`.

The **setf** macro can be used with this function to change the stream to which mouse input is sent. The second argument to **setf** must be a mouse input stream.

This function is an extension to Common Lisp.

## mouse-input-stream-interrupt-char

**Purpose:** The function **mouse-input-stream-interrupt-char** returns the function that is called when the given character is typed to a mouse input stream.

**Syntax:** **mouse-input-stream-interrupt-char** *mouse-input-stream char* [*Function*]

**Remarks:** The function returned by **mouse-input-stream-interrupt-char** takes two arguments: *mouse-input-stream* and *char*. The function is called as soon as *char* is typed to *mouse-input-stream*.

If the *char* argument is not an interrupt character, this function returns **nil**. If the character causes the Debugger to be entered, the function returns the keyword value **:debugger**.

You can use the **setf** macro to modify a character's interrupt handler. If you set the function value to **nil**, *char* is no longer an interrupt character on *mouse-input-stream*. If you set the function value to a function of two arguments, *char* becomes an interrupt character, and the function is called when *char* is typed. If you set the function value to the keyword value **:debugger**, typing this character causes the Debugger to be entered.

**Note:** The function corresponding to an interrupt character is called inside the system's interrupt handler; no other user interrupts are permitted while the function is running. If your function has an infinite loop, there is no way to interrupt it.

This function is an extension to Common Lisp.

## mouse-input-stream-p

**Purpose:** The predicate **mouse-input-stream-p** tests whether its argument *object* is a mouse input stream. It returns true if *object* is a mouse input stream.

**Syntax:** **mouse-input-stream-p** *object* [*Function*]

**Remarks:** This function is an extension to Common Lisp.

## mouse-input-stream-queue-mouse-events-p

**Purpose:** When a mouse event occurs, the predicate **mouse-input-stream-queue-mouse-events-p** is called on the mouse input stream that is the value of the function **mouse-input**. If the value returned is non-**nil**, a mouse event object encoding the mouse event is queued on the mouse input stream that is the value of the function **mouse-input**.

**Syntax:** **mouse-input-stream-queue-mouse-events-p** *mouse-input-stream* [*Function*]

**Remarks:** You can use the **setf** macro with this function to cause a mouse input stream to start or stop queueing mouse event objects.

The initial value for this function can be set in the function **make-mouse-input-stream** with the **:queue-mouse-events-p** keyword argument.

This function is an extension to Common Lisp.

**See Also:** **make-mouse-input-stream**

**read-any**

## mouse-input-stream-viewport

**Purpose:** The function **mouse-input-stream-viewport** returns the viewport that is associated with a mouse input stream.

**Syntax:** **mouse-input-stream-viewport** *mouse-input-stream* [*Function*]

**Remarks:** This function is an extension to Common Lisp.

## move-mouse

**Purpose:** The function **move-mouse** moves the mouse cursor from its current position to the position specified by the *x* and *y* arguments.

**Syntax:** **move-mouse** *x y* [*Function*]

**Remarks:** This function is an extension to Common Lisp.

**See Also:** ***mouse-x***

***mouse-y***

## move-viewport

**Purpose:** The function **move-viewport** moves a viewport's origin so that its top-left corner is at the point whose screen coordinates are specified by the *x* and *y* arguments.

**Syntax:** **move-viewport** *viewport x y* [*Function*]

**Remarks:** The root viewport cannot be moved.

This function is an extension to Common Lisp.

## peek-any

**Purpose:** The function **peek-any** peeks at and returns the next character or mouse event object in a mouse input stream without reading it. The character or mouse event object is read at a later time.

You can also use **peek-any** for skipping over characters and mouse event objects in the input stream until a particular character is encountered.

**Syntax:** **peek-any** `&optional` *peek-type mouse-input-stream eof-error-p eof-value recursive-p* [*Function*]

**Remarks:** The argument *mouse-input-stream* specifies a mouse input stream. If this argument is omitted or **nil**, the mouse input stream that is the value of the function **mouse-input** is used. If the *mouse-input-stream* argument is **t**, the mouse input stream that is the value of the function **keyboard-input** is used.

If end-of-file occurs, an error is signaled if the *eof-error-p* argument is omitted or true. If the *eof-error-p* argument is **nil**, then the *eof-value* argument is returned on end-of-file.

If the call to **peek-any** comes not from the top-level, but from within some function that itself has been called from **read** or a similar input function, the argument *recursive-p* must be non-**nil**. If this argument is omitted, the default value **nil** is used.

The *peek-type* argument specifies the type of object searched for on the mouse input stream. If *peek-type* is specified, it must be either **nil**, **t**, or a character. If this argument is omitted, the *peek-type* argument defaults to **nil**.

If the *peek-type* argument is **nil**, **peek-any** looks at and returns the next character or mouse event object in the mouse input stream without reading it from the stream.

If the *peek-type* argument is **t**, then **peek-any** reads and discards any white space characters at the front of the mouse input stream and returns the first mouse event object or character that is not white space without reading it from the stream. Note that comments are not discarded. (See the chapter "Input/Output" in the *Sun Common Lisp Reference Manual* for more information on white space characters.)

If the *peek-type* argument is a character, then **peek-any** discards characters and mouse event objects from the front of the input stream until it encounters a character that is equal to (**char=**) the *peek-type* argument. That character is returned without being read from the stream.

This function is an extension to Common Lisp.

## peek-any

| | |
|---|---|
| **See Also:** | **keyboard-input** |
| | **listen-any** |
| | **mouse-event-p** |
| | **mouse-input** |
| | **read-any** |
| | **read-any-no-hang** |

## pop-up-menu-choose

**Purpose:** The function **pop-up-menu-choose** displays a pop-up menu specified by the *pop-up-menu-object* argument. The menu appears on the display screen near the current position of the mouse. You can choose one of the objects on the menu by clicking the right button on top of the selected item, or you can move the mouse off the menu.

**Syntax:** **pop-up-menu-choose** *pop-up-menu-object* [*Function*]

**Remarks:** Once you have made a choice or moved the mouse off the menu, the menu disappears and two values are returned. The first value is the item that you selected, and the second is a keyword that indicates which button you used to select the item. If you did not make a selection and the menu has a default value, the default value and **nil** are returned; if the menu does not have a default value, both of the values returned are **nil**.

This function is an extension to Common Lisp.

**See Also:** **make-pop-up-menu**

## pop-up-menu-p

**Purpose:** The predicate **pop-up-menu-p** tests whether its argument *object* is a pop-up menu. It returns true if *object* is a pop-up menu.

**Syntax:** **pop-up-menu-p** *object* [*Function*]

**Remarks:** This function is an extension to Common Lisp.

## position-x, position-y

**Purpose:** The functions **position-x** and **position-y** return the *x*- and *y*-coordinates respectively of a position.

**Syntax:** **position-x** *position* [*Function*]

**position-y** *position* [*Function*]

**Remarks:** You can use the **setf** macro with the functions **position-x** and **position-y** to set the *x*- and *y*-coordinates of a position.

These functions are extensions to Common Lisp.

**Examples:**

```
> (setq pos (make-position 100 200))
#<position (100,200) 595ba3>
> (position-x pos)
100
> (setf (position-y pos) 300)
300
> pos
#<position (100,300) 595ba3>
```

## positionp

**Purpose:** The predicate **positionp** tests whether its argument *object* is a position. It returns true if *object* is a position.

**Syntax:** **positionp** *object* [*Function*]

**Remarks:** This function is an extension to Common Lisp.

**Examples:**

```
> (positionp (make-position 100 200))
T
> (positionp 7)
NIL
```

## read-any, read-any-no-hang

**Purpose:** The functions **read-any** and **read-any-no-hang** read either a single character or a single mouse event object from a mouse input stream.

**Syntax:**

**read-any** &optional *mouse-input-stream* *eof-error-p eof-value recursive-p* [*Function*]

**read-any-no-hang** &optional *mouse-input-stream* *eof-error-p eof-value recursive-p* [*Function*]

**Remarks:** The argument *mouse-input-stream* specifies a mouse input stream. If this argument is omitted or **nil**, the mouse input stream that is the value of the function **mouse-input** is used. If the *mouse-input-stream* argument is **t**, the mouse input stream that is the value of the function **keyboard-input** is used.

If there is no character or mouse event object ready to be input, the function **read-any** waits until a character is typed to the stream or a mouse event occurs on the stream *mouse-input-stream*. In this same situation, the function **read-any-no-hang** returns the value **nil** without waiting.

For both functions, when end-of-file occurs, an error is signaled if the *eof-error-p* argument is omitted or true. If the *eof-error-p* argument is **nil**, then the *eof-value* argument is returned when end-of-file occurs.

If the call to **read-any** or **read-any-no-hang** comes not from the top-level but from within some function that itself has been called from **read** or a similar input function, the argument *recursive-p* must be non-**nil**. If this argument is omitted, the default value **nil** is used.

These functions are extensions to Common Lisp.

**See Also:**

**keyboard-input**

**listen-any**

**make-mouse-input-stream**

**mouse-event-p**

**mouse-input**

**mouse-input-stream-queue-mouse-events-p**

**peek-any**

**unread-any**

## region-contains-point-p, region-contains-position-p

**Purpose:** The predicates **region-contains-point-p** and **region-contains-position-p** test whether a given position is in a given region.

The predicate **region-contains-point-p** is true if the position whose coordinates are *x* and *y* is in the given region.

The predicate **region-contains-position-p** is true when the position *position* is in the given region.

**Syntax:** **region-contains-point-p** *region x y* [*Function*]

**region-contains-position-p** *region position* [*Function*]

**Remarks:** These functions are extensions to Common Lisp.

**Examples:**

```
;; Create a region whose origin is (100,100) and whose corner is (400,300).
> (setq reg (make-region :x 100 :y 100 :width 300 :height 200))
#<region 300x200 at (100,100) 596507>
> (region-contains-point-p reg 150 299)
T
> (region-contains-position-p reg (make-position 150 300))
NIL
```

## region-corner, region-corner-x, region-corner-y, region-height, region-width, region-origin, region-origin-x, region-origin-y, region-size

**Purpose:** Each of these functions returns a component of a region.

**Syntax:**

| | |
|---|---|
| **region-corner** *region* `&optional` *result-position* | [*Function*] |
| **region-corner-x** *region* | [*Function*] |
| **region-corner-y** *region* | [*Function*] |
| **region-height** *region* | [*Function*] |
| **region-width** *region* | [*Function*] |
| **region-origin** *region* `&optional` *result-position* | [*Function*] |
| **region-origin-x** *region* | [*Function*] |
| **region-origin-y** *region* | [*Function*] |
| **region-size** *region* `&optional` *result-extent* | [*Function*] |

**Remarks:** You can use the macro **setf** with all these functions.

If a *result-position* argument is given for **region-corner** and **region-origin**, that position is modified to the region's corner position or origin position and returned. Otherwise a new position is created and returned.

If a *result-extent* argument is given for **region-extent**, that extent is modified to the region's extent and then returned. Otherwise a new extent is created and returned.

These functions are extensions to Common Lisp.

**Examples:**

```
;; Create a region whose origin is (400,500) and whose corner is (480,590).
> (setq r (make-region :x 400 :y 500 :width 80 :height 90))
#<region 80x90 at (400,,500) ada243>
> (region-corner r)
#<position (480,590) ada2e3>
> (region-corner-x r)
480
> (region-corner-y r)
590
> (region-height r)
90
> (region-width r)
80
```

```
> (region-origin r)
#<position (400,500) adb1db>
> (region-origin-x r)
400
> (region-origin-y r)
500
> (region-size r)
#<extent 80x90 adb253>
```

## region-intersection, region-union

**Purpose:** The function **region-intersection** returns the region covered in common by all of the given regions. If there is no intersection, it returns **nil**.

The function **region-union** returns the smallest region that contains all of the supplied regions.

**Syntax:** **region-intersection** *region region* `&rest` *regions* [*Function*]

**region-union** *region region* `&rest` *regions* [*Function*]

**Remarks:** These functions are extensions to Common Lisp.

**Examples:**

```
> (setq r1 (make-region :x 0 :y 0 :width 100 :height 200))
#<region 100x200 at (0,0) 5ac098>
> (setq r2 (make-region :x 50 :y 150 :width 100 :height 100))
#<region 100x100 at (50,150) 5ac0bd>
> (region-union r1 r2)
#<region 150x250 at (0,0) 5ac0ce>
> (region-intersection r1 r2)
#<region 50x50 at (50,150) 5ac0e3>
```

## region<, region<=, region=, region/=, region>, region>=

Purpose: These functions test containment and equality for regions.

The predicate **region<** is true if each argument except the last is contained in the following argument.

The predicate **region<=** is true if each argument except the last is contained in or equals the following argument.

The predicate **region=** is true if every argument is the same region.

The predicate **region/=** is true if no two arguments are the same region.

The predicate **region>** is true if each argument except the last contains the argument that follows it.

The predicate **region>=** is true if each argument except the last contains or is equal to the argument that follows it.

Syntax:

**region<** *region region* &rest *regions* [*Function*]

**region<=** *region region* &rest *regions* [*Function*]

**region=** *region region* &rest *regions* [*Function*]

**region/=** *region region* &rest *regions* [*Function*]

**region>** *region region* &rest *regions* [*Function*]

**region>=** *region region* &rest *regions* [*Function*]

Remarks: These functions are extensions to Common Lisp.

Examples:

```
> (setq region1 (make-region :x   0 :y   0 :corner-x 100 :corner-y 100)
      region2 (make-region :x   0 :y   0 :corner-x 100 :corner-y 101)
      region3 (make-region :x  50 :y  50 :corner-x 200 :corner-y 300)
      region4 (make-region :x 200 :y 300 :corner-x 500 :corner-y 700)
      region5 (make-region :x 150 :y 299 :corner-x 500 :corner-y 701)
      region6 (make-region :x 150 :y 299 :corner-x 500 :corner-y 700))
#<Region 350x401 at (150,299) 4850CB>
> (region< region1 region2)
T
> (region< region1 region1)
NIL
> (region<= region1 region2)
T
```

```
> (region<= region1 region1)
T
> (region> region2 region1)
T
> (region> region2 region2)
NIL
> (region>= region2 region1)
T
> (region>= region2 region2)
T
;; For region/= to be true, the regions must be all different.
> (region/= region1 region2 region3 region4 region5 region6)
T
> (region/= region1 region2 region3 region4 region5 region6 region1)
NIL
> (region= region1 region1 region1 region1 region1)
T
> (region= region1 region2)
NIL
```

## regionp

**Purpose:** The predicate **regionp** tests whether its argument *object* is a region. It returns true if *object* is a region.

**Syntax:** **regionp** *object* [*Function*]

**Remarks:** This function is an extension to Common Lisp.

**Examples:**

```
> (regionp (make-region :x 0 :y 0 :width 100 :height 200))
T
> (regionp 8)
NIL
```

## rename-font

**Purpose:** The function **rename-font** changes the name of the font *font* to the name *new-name*. The name of the font in the registry is changed from its previous name to the new name.

**Syntax:** **rename-font** *font new-name* [*Function*]

**Remarks:** The *font* argument must be a font, a string, or a symbol. If the argument is a string or a symbol, the function **find-font** is called to find the font whose name is the string or symbol.

The *new-name* argument is either a string or a symbol. If it is a symbol, the symbol's print name is used.

This function is an extension to Common Lisp.

**See Also:** **font-name**

## reshape-viewport

**Purpose:** The function **reshape-viewport** moves and reshapes a viewport so that its screen region is the region specified by the keyword arguments.

**Syntax:** **reshape-viewport** *viewport* &key `:region :x :y` [*Function*]

```
                                    :width :height
                                    :corner-x :corner-y
```

**Remarks:** The keyword arguments are used to specify the new region. All coordinates are given in terms of the root viewport. You must specify enough keyword arguments to identify the region uniquely.

The **:x** and **:y** keyword arguments specify the *x*- and *y*-coordinates respectively of the top-left corner of the region.

The **:corner-x** and **:corner-y** keyword arguments specify the *x*- and *y*-coordinates respectively of the point just below and to the right of the region.

The **:width** and **:height** keyword arguments specify the width and height respectively of the region.

Moving a viewport also moves all of its descendants.

The root viewport cannot be reshaped.

This function is an extension to Common Lisp.

## root-viewport

**Purpose:** The function **root-viewport** returns the root viewport.

**Syntax:** **root-viewport** [*Function*]

**Remarks:** The root viewport is a viewport onto a special bitmap that requires less memory but has limited capabilities. You cannot modify the bits of this special bitmap in any way without signaling an error.

This function is an extension to Common Lisp.

## stream-current-font

**Purpose:** The function **stream-current-font** returns the current font of a bitmap output stream.

**Syntax:** **stream-current-font** *bitmap-output-stream* [*Function*]

**Remarks:** You can use the macro **setf** to modify the stream's current font. The second argument to **setf** must be a font, a string, or a symbol. If the argument is a string or a symbol, the function **find-font** is called to find the font whose name is the string or symbol. Changing a stream's current font does not modify the stream's linefeed distance.

This function is an extension to Common Lisp.

**Examples:**

```
;; Create a 100x200 bitmap.
> (setq btmp (make-bitmap :height 100 :width 200))
#<Bitmap 200x100 5ACEC3>
;; Create a bitmap output stream to that bitmap.
> (setq b-o-s (make-bitmap-output-stream :bitmap btmp))
#<Output-Stream to #<Bitmap 200x100 5ACEC3> 5AD21D>
;; The bitmap output stream's current font is *default-font*.
> (eq (stream-current-font *) *default-font*)
T
```

**See Also:** **stream-linefeed-distance**

## stream-draw-circle, stream-draw-line, stream-draw-polyline

Purpose: The function **stream-draw-circle** draws a circle of radius *radius* around a bitmap output stream's current position.

The function **stream-draw-line** draws a line segment from a bitmap output stream's current position to the position *end*. The bitmap output stream's new current position becomes the position *end*.

The function **stream-draw-polyline** draws a series of connected line segments, starting at a bitmap output stream's current position and then going through each position in a sequence of positions. The current position of the bitmap output stream is left at the final position.

Syntax:

**stream-draw-circle** *bitmap-output-stream radius* `&key :width :operation` [*Function*]

**stream-draw-line** *bitmap-output-stream end* `&key :width :operation` [*Function*]

**stream-draw-polyline** *bitmap-output-stream positions* `&key :width :operation` [*Function*]

Remarks: If the **:width** keyword argument is specified, it defines the line width that is used for drawing the line segments and circles. For the function **stream-draw-circle**, the border is drawn so that its outer edge is at the specified radius; the width must be less than or equal to the radius. If this keyword argument is omitted or **nil**, the default value 1 is used.

The value of the keyword argument **:operation** controls how the bits that are written onto the bitmap are combined with the bits that are already there. If this keyword argument is omitted or **nil**, the bitmap output stream's operation is used. You can find that operation by using the function **stream-operation**.

These functions are extensions to Common Lisp.

See Also: **draw-circle**

**draw-line**

**draw-polyline**

**stream-operation**

## stream-linefeed-distance

**Purpose:** The function **stream-linefeed-distance** accesses the linefeed distance of a bitmap output stream.

**Syntax:** **stream-linefeed-distance** *bitmap-output-stream* [*Function*]

**Remarks:** You can use the **setf** macro with this function to modify a bitmap output stream's linefeed distance.

When a bitmap output stream is created, its linefeed distance is the height of the initial font.

A stream's linefeed distance is used when one of the Common Lisp output functions sends a newline character to the bitmap output stream. The bitmap output stream's *y*-coordinate is incremented by the linefeed distance, and the *x*-coordinate is set to 0.

This function is an extension to Common Lisp.

## stream-operation

**Purpose:** The function **stream-operation** returns a bitmap output stream's default **bitblt** operation, which is used in writing characters or figures to the bitmap output stream's bitmap.

**Syntax:** **stream-operation** *bitmap-output-stream* [*Function*]

**Remarks:** You can use the macro **setf** with this function to set a new value. The new value must be an acceptable first argument to the **boole** function.

This function is an extension to Common Lisp.

**Examples:**

```
;; Create a 100x200 bitmap.
> (setq x (make-bitmap :height 100 :width 200))
#<Bitmap 200x100 AD476B>
;; Create a bitmap output stream to that bitmap.  Specify an operation.
> (setq b-o-s (make-bitmap-output-stream :bitmap x :operation boole-1))
#<Output-Stream to #<Bitmap 200x100 AD476B> AD54B3>
;; The stream operation of b-o-s is the specified operation.
> (eql (stream-operation b-o-s) boole-1)
T
```

## stream-position, stream-x-position, stream-y-position

**Purpose:** These functions return the output position of a bitmap output stream. The output position specifies the next position for writing to the bitmap output stream's bitmap.

The function **stream-position** returns a position.

The function **stream-x-position** returns the $x$-coordinate of the position.

The function **stream-y-position** returns the $y$-coordinate of the position.

**Syntax:**

**stream-position** ***bitmap-output-stream*** [*Function*]
`&optional` ***result-position***

**stream-x-position** ***bitmap-output-stream*** [*Function*]

**stream-y-position** ***bitmap-output-stream*** [*Function*]

**Remarks:** If a ***result-position*** argument is given for the function **stream-position**, that position is modified to the output position and then returned. Otherwise a new position is created and returned.

You can use the **setf** macro with these functions to modify a bitmap output stream's position.

When a bitmap output stream is created, its stream position is the position whose $x$-coordinate is 0 and whose $y$-coordinate is the baseline height of the initial font.

These functions are extensions to Common Lisp.

**Examples:**

```
;; Create a bitmap and a bitmap output stream.
> (setq b-o-s (make-bitmap-output-stream :width 100 :height 200))
#<Output-Stream to #<Bitmap 100x200 AD75BB> AD8463>
;; Check the initial value.
> (and (= 0 (stream-x-position b-o-s))
       (= (font-baseline *default-font*) (stream-y-position b-o-s)))
T
;; Set the x position to a new value.
> (setf (stream-x-position b-o-s) 50)
50
```

## string-width

**Purpose:** The function **string-width** determines how many bits wide the string *string* is when printed in the font *font.*

**Syntax:** **string-width** *string font* [*Function*]

**Remarks:** The *font* argument must be a font, a string, or a symbol. If the argument is a string or a symbol, the function **find-font** is called to find the font whose name is the string or symbol.

The function may give false results if the string contains any characters that cannot be printed, such as the newline character. The space character is a printable character.

This function is an extension to Common Lisp.

## unread-any

**Purpose:** The function **unread-any** returns a character or mouse event object to the front of a mouse input stream's queue. The character or mouse event object must be the same object that was last read from the queue.

**Syntax:** **unread-any** *char-or-mouse-event* &optional *mouse-input-stream* [*Function*]

**Remarks:** The argument *mouse-input-stream* specifies a mouse input stream. If this argument is omitted or **nil**, the mouse input stream that is the value of the function **mouse-input** is used. If the *mouse-input-stream* argument is **t**, the mouse input stream that is the value of the function **keyboard-input** is used.

This function is an extension to Common Lisp.

**See Also:** **keyboard-input**

**mouse-input**

**read-any**

## viewport-at-point, viewport-at-position

**Purpose:** The function **viewport-at-point** interprets *x* and *y* as the coordinates of a point on the screen. It returns as its value the viewport that is displayed at that point on the screen.

The function **viewport-at-position** is identical to **viewport-at-point** except that it is passed a single position argument rather than *x*- and *y*-coordinates.

**Syntax:** **viewport-at-point** *x y* [*Function*]

**viewport-at-position** *position* [*Function*]

**Remarks:** These functions are extensions to Common Lisp.

## viewport-bitmap

**Purpose:** The function **viewport-bitmap** returns a viewport's underlying bitmap.

**Syntax:** **viewport-bitmap** *viewport* [*Function*]

**Remarks:** This function is an extension to Common Lisp.

## viewport-bitmap-offset, viewport-bitmap-x-offset, viewport-bitmap-y-offset

**Purpose:** The function **viewport-bitmap-offset** returns the position that represents the offset (from the bitmap's origin) of a viewport's origin. This offset indicates what part of the bitmap is being displayed in the viewport.

The functions **viewport-bitmap-x-offset** and **viewport-bitmap-y-offset** return the $x$- and $y$-coordinates of the offset respectively.

**Syntax:**

**viewport-bitmap-offset** *viewport* `&optional` *result-position* [*Function*]

**viewport-bitmap-x-offset** *viewport* [*Function*]

**viewport-bitmap-y-offset** *viewport* [*Function*]

**Remarks:** If a *result-position* argument is given for the function **viewport-bitmap-offset**, that position is modified to the viewport's offset and returned. Otherwise a new position containing the viewport's offset is created and returned.

You can use the **setf** macro with these functions to change the viewport's offset. In particular, modifying **viewport-bitmap-y** causes vertical scrolling.

Any attempt to set the offset so that part of the viewport's clipping region falls outside the bitmap boundaries results in an error.

These functions are extensions to Common Lisp.

## viewport-bitmap-region, viewport-screen-region

Purpose: The function **viewport-bitmap-region** returns a copy of a viewport's bitmap clipping region.

The function **viewport-screen-region** returns a copy of a viewport's screen clipping region.

Syntax: **viewport-bitmap-region** *viewport* `&optional` *result-region* [*Function*]

**viewport-screen-region** *viewport* `&optional` *result-region* [*Function*]

Remarks: If a region is passed as the second argument to these functions, the result is copied into that region object and returned; otherwise a new region is created.

These functions are extensions to Common Lisp.

## viewport-children, viewport-parent

**Purpose:** The function **viewport-children** returns a list of a viewport's children. The list is in the same order as the children's sibling stack.

The function **viewport-parent** returns a viewport's parent.

**Syntax:**

**viewport-children** *viewport* [*Function*]

**viewport-parent** *viewport* [*Function*]

**Remarks:** The **setf** macro for **viewport-parent** changes a viewport's parent. You can only change a viewport's parent when the viewport is inactive. The viewport is put at the top of the sibling stack of its new parent's children.

The **setf** macro can be used with the **viewport-children** function. The second argument to **setf** must be a list of inactive viewports. All of the *viewport* argument's children must also be inactive. The *viewport* argument's list of children is modified to be the second argument; children in the sibling stack will appear in the same order as in the viewport list. Any viewport in the viewport list whose parent was not the *viewport* argument is modified so that its parent becomes the *viewport* argument. Any viewport whose parent was formerly the *viewport* argument but that is not in the viewport list becomes detached from the tree.

The parent of the root viewport is **nil**.

These functions are extensions to Common Lisp.

## viewportp

**Purpose:** The predicate **viewportp** tests whether its argument *object* is a viewport. It returns true if *object* is a viewport.

**Syntax:** **viewportp** *object* [*Function*]

**Remarks:** This function is an extension to Common Lisp.

**Examples:**

```
;; To run this example, you must have already initialized the
;; Window Tool Kit.

;; Create a 10x10 bitmap and a viewport onto that bitmap.
;; Note that make-viewport returns two values, the viewport and the bitmap.
> (multiple-value-setq (vwpt btmp)(make-viewport :width 10 :height 10))
#Viewport 10x10 at (0,0) onto #<Bitmap 10x10 40D4E5> 40D51D>
> (viewportp vwpt)
T
> (viewportp btmp)
NIL
```

## window-frame

**Purpose:** The function **window-frame** returns the window frame that is associated with a given window.

**Syntax:** **window-frame** *window* [*Function*]

**Remarks:** The window frame can be thought of as a special bitmap; the image that this bitmap displays on the screen is the window's frame. You cannot modify the bits of this special bitmap in any way without signaling an error. You may attach active regions to window frames in the same way that you normally attach active regions to bitmaps.

You cannot use the **setf** macro with this function.

This function is an extension to Common Lisp.

## window-inner-border-width, window-outer-border-width

**Purpose:** The function **window-inner-border-width** returns the width of the inner border of the window.

The function **window-outer-border-width** returns the width of the outer border of the window.

The window border consists of two strips: the black strip around the edge of the window is the outer border, and the white strip inside the black strip is the inner border.

**Syntax:** **window-inner-border-width** *window* [*Function*]

**window-outer-border-width** *window* [*Function*]

**Remarks:** The **setf** macro can be used with these functions to modify the widths of the inner and outer borders of a window.

These functions are extensions to Common Lisp.

## window-title, window-title-font

**Purpose:** The function **window-title** returns the title of a window as a string.

The function **window-title-font** returns the font in which a window's title is displayed.

**Syntax:** **window-title** *window* [*Function*]

**window-title-font** *window* [*Function*]

**Remarks:** You can use the **setf** macro with the function **window-title** to modify the title of a window. The second argument to **setf** must be a string.

You can use the **setf** macro with the function **window-title-font**. Doing so redraws the title of the window in the new font. The second argument to setf must be a font, a string, or a symbol. If the argument is a string or a symbol, the function **find-font** is called to find the font whose name is the string or symbol.

These functions are extensions to Common Lisp.

**Examples:**

```
;; To run this example, you must have already initialized the
;; Window Tool Kit.

;; Create a window with the title "hello".
> (setq w (make-window :width 100 :height 200 :title "hello"))
#<WINDOW 4ACOAB>
> (window-title w)
"hello"
;; Note that the title font is *default-font*.
> (eq (window-title-font w) *default-font*)
T
;; Modify the title.
> (setf (window-title w) "new-name")
"new-name"
> (window-title w)
"new-name"
```

## window-vertical-scroll-ratio, window-horizontal-scroll-ratio

Purpose: The functions **window-vertical-scroll-ratio** and **window-horizontal-scroll-ratio** return the vertical and horizontal scroll ratio respectively of a given window.

Syntax: **window-vertical-scroll-ratio** *window* [*Function*]

**window-horizontal-scroll-ratio** *window* [*Function*]

Remarks: The scroll ratio is a Common Lisp ratio between 0 and 1. Generally the scroll ratio is the ratio of the current location of the window to the size of the window's underlying bitmap. However, window system developers may redefine the methods for scrolling and for calculating these ratios so that scrolling may be performed over an abstract bitmap or extent.

These functions may be used with the macro **setf** to specify a vertical or horizontal ratio for the given window. If the value of the ratio is set to **nil**, the window cannot scroll and the scroll bars disappear.

These functions are extensions to Common Lisp.

See Also: **make-window**

## windowp

**Purpose:** The predicate **windowp** tests whether its argument *object* is a window. It returns true if *object* is a window.

**Syntax:** **windowp** *object* [*Function*]

**Remarks:** This function is an extension to Common Lisp.

**Examples:**

```
;; To run this example, you must have already initialized the
;; Window Tool Kit.

> (setq w (make-window :width 100 :height 200 :title "hello"))
#<WINDOW 4ACOAB>
> (windowp w)
T
> (windowp 7)
NIL
```

## windows-available-p

**Purpose:** The function **windows-available-p** checks the Lisp environment for a window system that is capable of supporting the Window Tool Kit.

**Syntax:** **windows-available-p** [*Function*]

**Remarks:** The function **windows-available-p** returns a non-**nil** value if a window system is available for the Window Tool Kit; otherwise it returns **nil**.

This function is an extension to Common Lisp.

## with-asynchronous-method-invocation-allowed

**Purpose:** The macro **with-asynchronous-method-invocation-allowed** allows active region methods and interrupt character methods to occur asynchronously rather than sequentially.

It is only used inside the body of an active region method or an interrupt character method. The *form* arguments are evaluated. Any pending active region methods and any interrupt character methods that would normally be queued until the current method terminated are instead run immediately.

**Syntax:** **with-asynchronous-method-invocation-allowed** {*form*}* [*Macro*]

**Remarks:** Normally, all active region and interrupt character methods are executed sequentially. However, sometimes an active region or interrupt character method needs to wait for the action taken by another active region method or interrupt character method to occur. The macro **with-asynchronous-method-invocation-allowed** provides for this.

The *form* arguments are evaluated in an an environment where pending active region methods and interrupt character methods are allowed to run. These methods are executed sequentially with respect to each other unless one of them contains a **with-asynchronous-method-invocation-allowed** form.

Here is an example of the use of the **with-asynchronous-method-invocation-allowed** macro. If you wanted an active region method to display a pop-up menu, the code would be similar to the following:

```
(with-asynchronous-method-invocation-allowed
   (pop-up-menu-choose a-menu))
```

The **with-asynchronous-method-invocation-allowed** macro is necessary because pop-up menus use active regions that must be allowed to execute immediately.

This macro is an extension to Common Lisp.

## with-fast-drawing-environment

**Purpose:** The macro **with-fast-drawing-environment** groups display operations. Overhead operations that are required to produce output are executed only once rather than for every display operation in the group.

**Syntax:** **with-fast-drawing-environment** {*form*}* [*Macro*]

**Remarks:** The output for the group may not appear on the screen until the macro is exited. Therefore, you should not use this macro to group operations that require user input or that will run for a long time.

This macro is an extension to Common Lisp.

## with-mouse-methods-preempted

**Purpose:** The **with-mouse-methods-preempted** macro evaluates each of its *form* arguments. While these forms are being evaluated, any active region that is not attached to the *bitmap* argument is disabled. Its methods are not called even if a mouse event occurs inside it.

The *bitmap* argument can also be **nil**. In this case, all active regions are disabled.

**Syntax:** **with-mouse-methods-preempted** *bitmap* {*form*}* [*Macro*]

**Remarks:** The results of evaluating the last form are returned as the results of **with-mouse-methods-preempted**.

This macro is an extension to Common Lisp.

# Chapter 13. The Editor

# Chapter 13. The Editor

## Introduction to the Editor

The text editor for Sun Common Lisp is based on EMACS and its various descendants. The Editor is a ***display editor***: it exhibits on the screen the piece of text that you are editing. Somewhere on the screen it places a ***cursor***, an indication of a location of current importance. You issue commands to move the cursor or to modify the text near the cursor. Many commands are typed as a small number of keystrokes, sometimes in sequence and sometimes as chords.

To understand the commands, you will need to become familiar with the definitions discussed in the section that follows. After practicing the basic movement and modification commands, you will be ready to try the more specialized commands, such as those used for searching and replacing.

Certain editing capabilities are useful for Lisp programming. Sun Common Lisp provides commands for formatting Lisp and for interacting with the outside Lisp environment.

Advanced users should read the final section of this chapter, which deals with customizing the Editor.

## Definitions and Notation

Several terms used throughout this chapter have special meanings. They are gathered here for easy reference. Note in particular that "region" and "window" in this chapter refer to Editor objects and not to window system objects.

### Point

In the Editor, the ***point*** is the center of attention. The point lies between two characters. It is represented in the display by the cursor, which is on the character just after the point. Most Editor commands involve the point in some way. Here are some examples. The command to insert a character inserts that character at the point. There are two commands to delete characters: one deletes the character before the point, and the other deletes the character after the point.

### Mark and Region

The ***mark*** is another important location. Many commands store the point into the mark. The ***region*** extends from the point to the mark. Many commands limit their effects to the region. It is common to move the point to one end of an intended region, store it in the mark, move the point to the other end, and then execute the regional command. The section "Mark Commands" discusses marks and regions in more detail.

### Buffer, File, and Editor Window

The terms "buffer," "file," and "Editor window" can all be used to refer to the text that is being edited. The three terms emphasize different aspects of that text. A ***buffer*** is the Editor's internal representation of text. A ***file*** is the machine's representation of text outside the Editor. An ***Editor window*** might be considered the user's representation of text; it is what appears on the screen. The section "Buffers, Files, and Editor Windows" contains a more detailed discussion of these objects.

### Typing Control Keys and Meta Keys

The Editor interprets various chords and sequences of keystrokes as single keys. You may already be accustomed to some of these. "Capital B" is a single keystroke, although you type it by holding down **Shift** and typing **B**.

***Control*** keys are similar. Type "control-b" by holding down **Ctrl** and typing **B**. This chord is called **Ctrl-B**.

***Meta*** keys are entirely different. They use **Esc** as a prefix. To type "meta-b," press **Esc**, release it, and then type **B**.

Most ***meta-control*** keys can be typed by combining the ways of typing meta keys and control keys. Another way to type these keys uses **Ctrl-Z** as a prefix; to type "meta-control-b," type **Ctrl-Z** followed by **B**.

**Ctrl-X** is also used as a prefix for keys. **Ctrl-XB** is typed as **Ctrl-X** followed by **B**.

## Functions and Commands

The Editor performs many different functions. You control which function the Editor performs by issuing ***commands***. You may issue commands by typing a key that is ***bound*** to the command or by typing an ***extended command***.

Commonly used commands are bound to individual keys. Typing that key issues the associated command. This allows you to give the command with just a few keystrokes. The section "Customizing the Editor" explains how to modify key bindings.

Any command can be given as an extended command. Each command has a name. You issue an extended command by using the **Extended Command** command. (That command itself is invoked using its key binding to *meta*-**X**.)

**Extended Command** *meta*-**X** [*Command*]

The **Extended Command** command prompts for a command name and then executes that command. The section "Other Arguments" discusses commands that simplify the typing of the command name. In particular, **Ctrl-G** exits from **Extended Command** without issuing another command.

## Entering and Leaving the Editor

You can enter the Editor by typing the function **ed** to the top-level Lisp prompt:

```
> (ed)
```

If you have invoked Lisp from an environment that supports a window system, typing the function **ed** puts you in the Editor and initializes the Window Tool Kit.

**Exit Editor** **Ctrl-XCtrl-Z** [*Command*]

This command returns you to the top level of Lisp. It does not save any changes you may have made to the buffer; the file version of the text is left unchanged. The section "Buffers, Files, and Editor Windows" explains how to save modifications so that they are visible from outside the Editor.

After using this exit command, typing the function **ed** to the top-level prompt returns you to the same place in the Editor; it is as if you never left the Editor.

**Note:** If you use **ed** to initialize the Window Tool Kit, you must continue to use the Editor in the window environment for subsequent editing sessions. Similarly, if you started the Editor as a terminal editor, you cannot integrate the Editor with the Window Tool Kit in subsequent editing sessions; you must continue to use the Editor as a terminal editor.

## The Display

The Editor divides the screen into three important parts: the Editor window, the mode line, and the echo area.

- The ***Editor window*** is a large rectangular area that displays text. It occupies most of the screen.
- The ***mode line*** is one line of information just below the Editor window. Normally, the mode line is displayed in reverse video for emphasis.
- The ***echo area*** is the bottom line of the screen.

The functions of the parts of the display are described in the following sections.

## The Editor Window

The Editor window shows a portion of the current buffer. In general, each line of the buffer appears on one line of the Editor window. However, if the line is too long to fit on one line of the screen, it continues on the next line of the Editor window. The character backslash (\) appears in the last column of a line to indicate that the next line of the Editor window is a continuation.

The cursor shows the current location of the point. The cursor is on top of the character following the point. Certain characters are displayed on the screen as more than one character. In particular, a tab character can appear as some number of space characters, and various nonprinting control characters appear in a two-character form. For example, the newpage character (control-L) appears as `^L`. When the point precedes one of these characters, the cursor appears on the first character of the representation.

If you are using the Editor from within the Window Tool Kit, each window has a pop-up menu that describes the available commands. You can access this menu by clicking the right mouse button on the window you wish to manipulate. The menu lists the options for moving a window and for changing its size. By clicking the right mouse button, you can select an option. To mark text and to move the cursor, you click the left mouse button. See the chapter "The Window Tool Kit" for more detailed information.

You can use the following commands to create, to move to, or to delete windows or to redisplay a window that looks different from what is described in the section "The Display."

**Refresh Screen** **Ctrl-L** [*Command*]
This command clears the screen and then redisplays what was supposed to be on the screen.

**Next Window** **Ctrl-XN** **Ctrl-XO** [*Command*]
When there are multiple Editor windows, the point is in only one of them. This command moves the point to the next window.

**Delete Next Window** **Ctrl-X1** [*Command*]
This command deletes the next window. Repeated use of the command makes the current window the only window.

**New Window** **Ctrl-X2** [*Command*]
This command creates a new Editor window.

## The Mode Line

The mode line displays important information about the buffer, including the name of the editor, the buffer name, the buffer status, and the editor mode. Here is an example of a mode line:

```
-**--Ed: test.file        (Fundamental)-----------------------------------
```

This particular mode line states that the editor is the Common Lisp Editor (`Ed:`). Thus, confusion with other programs that have similar displays (EMACS, for example) is avoided. The mode line's Editor window contains a buffer named `test.file`. The characters `**` indicate that the buffer has been modified since it was last written. If the characters `--` rather than the characters `**` appear in that position on the mode line, the buffer has not been modified since it was last written. The Editor is in fundamental mode (`(Fundamental)`).

**Fundamental Mode** [*Command*]
The normal behavior of certain commands may change somewhat depending on the current mode. Fundamental mode is the basic mode of the Editor. If the Editor somehow gets into another mode, the command **Fundamental Mode** returns the Editor to its normal behavior. The section "Editing Lisp" discusses another mode.

## The Echo Area

The echo area is located at the bottom of the screen. It is used for communication with the user. The Editor sometimes prints out messages about what it is doing. It also prints error messages there when you ask it to do something that it cannot do. For example, if you issue a command to move the point backwards one line from the first line of the buffer, the Editor prints `No previous line` in the echo area.

### Prefix Arguments

A ***prefix argument*** is a number that can be supplied to a command. The prefix argument is set just before issuing the command. The most common use of the prefix argument is as a repeat count. In general, a negative prefix argument causes a command to be executed "backwards." (There is a command to move the point to the next line of the buffer. With a negative prefix argument, that command moves the point toward the beginning of the buffer.)

**Argument Digit** *meta-digit* [*Command*]

This command is used to set the prefix argument. Typing ***meta-4meta-3*** sets the prefix argument to 43. This command is intended to be invoked only through a key binding.

**Negative Argument** *meta--* [*Command*]

This command is used to generate a negative prefix argument. It negates any numeric prefix argument that follows it, or it sets the prefix argument to −1 if there is no following argument.

**Universal Argument** **Ctrl-U** [*Command*]

**Universal Argument Default** [*Variable*]

The command **Universal Argument** sets or modifies the prefix argument. If it is followed by a number, it sets the prefix argument to that number. Typing **Ctrl-U43** also sets the prefix argument to 43.

If the command **Universal Argument** is followed immediately by another command, then it multiplies its own prefix argument by **Universal Argument Default**, a variable whose default value is 4. Typing **Ctrl-UCtrl-F** would normally pass a prefix argument of 4 to the command bound to **Ctrl-F**, and typing **Ctrl-UCtrl-UCtrl-F** would normally pass that command an argument of 16.

## Other Arguments

Many commands prompt for arguments. Both the prompt and the characters you type appear in the echo area. These arguments are of two types: single-character arguments and string arguments.

Single-character arguments are relatively simple. You type a character and the corresponding command is executed.

String arguments are more complicated. Often the string must be one of a small set of possible answers. Many times there is a default value for the string. Most of the motion commands and modification commands are available for editing string arguments. (In particular, **Quoted Insert**, which is bound to **Ctrl-Q**, can be used to put unusual characters in the string.) In addition, you can use the following commands to type and edit arguments.

**Help on Prompt** **Ctrl-_** *meta*-**H** [*Command*]

Most commands that prompt for an argument set a help string that explains in more detail what the command needs. This command prints that string. Moreover, if there are only a few, it prints a list of possible completions of the current string. This command ignores the prefix argument.

**Complete Keyword** **Tab** [*Command*]

**Complete Field** **Space** [*Command*]

These commands attempt to complete some of the current string argument. If they cannot complete that part, then they beep the terminal. If there are only a few possible completions, then they display those completions. **Complete Keyword** tries to finish the entire string. **Complete Field** only tries to complete it up to the next separator.

**Confirm Parse** **Return** **Linefeed** [*Command*]

This finishes the prompting. If nothing has been typed to the prompt, then it returns the default value. Otherwise it checks whether the string is a valid answer to the prompt and either returns the string or signals an error. This command ignores the prefix argument.

**Next Parse** **Ctrl-N** [*Command*]

**Previous Parse** **Ctrl-P** [*Command*]

The Editor maintains a record of previous attempts at string arguments. These commands replace the current incomplete attempt with another try. The prefix argument is used to choose an old string argument that is not adjacent to the current one. If the current string is not the empty string, it is added to the list of old strings. If the prefix argument is 0, it displays the record of strings.

**Beginning Of Parse** **Ctrl-A** [*Command*]

This command moves the point to the beginning of the string argument. It is careful not to do anything to the prompt. It ignores the prefix argument.

**Kill Parse** **Ctrl-W** [*Command*]

This command removes all the text from the current attempt at a string. It ignores the prefix argument. (See the section "Killing and Unkilling.")

**Insert Parse Default** [*Command*]

This command inserts the default string at the point. This allows you to edit that string to get a string that is similar to it. This command ignores the prefix argument.

## The Help Facility

The Editor includes a list of all the commands, a brief description of each command, and information about what keys are bound to which commands. This information is available through the command **Help**.

**Help** *meta*-**?** *meta*-**H** **Ctrl-_** [*Command*]

This command is the help facility. It prompts for a single character that determines what sort of help to give. These are the choices:

| | |
|---|---|
| **A** | List commands whose names include a specified string. |
| **C** | Describe the command bound to a particular key. |
| **D** | Give the documentation for a particular Editor command. |
| **G** | Give the documentation for a particular Editor object. |
| **H**<br>**?** | Explain the options in the help command. |
| **L** | List the last 60 characters typed. |
| **N**<br>**Q**<br>**Backspace**<br>**Delete** | Quit (from help) without doing anything. |
| **W** | List all the key bindings for a particular Editor command. |

**Apropos** *[Command]*
This command prompts for a string. It then displays the name of each command that includes that string as part of its name.

**Describe Key** *[Command]*
This command prompts for a keystroke. It then shows the name of the function, if any, that is bound to that key.

**Describe Command** *[Command]*
This command prompts for the name of a command. It then prints out the documentation string associated with that command.

**Generic Describe** *[Command]*
This command uses two user-supplied arguments. First it asks what sort of thing it is supposed to describe: a command, a key, a variable, or a character attribute. Then it asks which particular thing it is supposed to describe. After it collects all this information, it displays the corresponding documentation string.

**View Lossage** *[Command]*
This command prints out the last 60 characters typed. This is useful when something unusual has just occurred and you want to know what you actually typed.

**Where Is** *[Command]*
This command prompts for the name of a command. It prints out a list of all keys that are bound to that command.

## Other Useful Information

Many commands interpret **Ctrl-G** as a signal to stop execution and return (unfinished). This signal is most often used to get out of prompting.

You may use the *meta*-**H** key to get some sort of help message in many circumstances. Many commands that prompt for input also type a help message in response to *meta*-**H**.

It is important to remember the difference between a buffer and a file. After editing text, the buffer (that is, the edited version) differs from the corresponding file. The command **Save File** should be used to make your changes permanent. If, in retrospect, the changes were a mistake, the command **Revert Buffer** can be used to restore the buffer to its earlier state if it has not been saved in the corresponding file. (See the section "Buffers, Files, and Editor Windows" for more information.)

There are commands that remove large portions of text in just a few keystrokes. That text can be recovered. (See the section "Killing and Unkilling.")

## Motion Commands

Perhaps the most frequently used commands are those that move the point around in the buffer. They are divided into two groups: keyboard commands that move the point, and keyboard commands that manipulate the point and the mark.

### Keyboard Commands

The following commands move the point around in the buffer. The point is always displayed in the Editor window. If the point is moved outside the Editor window, then the Editor window is changed so that the new location of the point is inside the Editor window.

| | | |
|---|---|---|
| **Next Line** | **Ctrl-N** | [*Command*] |
| **Previous Line** | **Ctrl-P** | [*Command*] |

These commands move the point to the same position in another line. Without a repeat count, the point moves to an adjacent line. With a repeat count, the point is positioned that many lines away. A negative repeat count moves the point in the opposite direction. If the destination line is too short to have the same position, the point is put at the end of the line.

**Next Line** with no argument or with an argument of 1, and with the point on the last line of the buffer, lengthens the buffer by one line. Other attempts to move the point outside the current bounds of the buffer fail; the point moves to the beginning or end of the buffer, but no farther.

| | | |
|---|---|---|
| **Forward Character** | **Ctrl-F** | [*Command*] |
| **Backward Character** | **Ctrl-B** | [*Command*] |
| **Echo Area Backward Character** | **Ctrl-B** | [*Command*] |

These commands move the point one character in the buffer. **Forward Character** moves the point forward one character in the buffer. With a repeat count, it moves the point forward that many characters. A negative repeat count moves the point backward in the buffer. Moving forward from the end of a line puts the point at the beginning of the next line. This command does not increase the size of the buffer; the point stops at the end of the buffer. **Backward Character** is similar except that it moves the point backward in the buffer. **Echo Area Backward Character** is used during prompting; it is particularly careful not to move into the prompt.

**Forward Word** *meta*-**F** [*Command*]
**Backward Word** *meta*-**B** [*Command*]
**Echo Area Backward Word** *meta*-**B** [*Command*]

These commands move the point one word in the buffer. A **word** is a collection of consecutive letters and numbers. **Forward Word** with no repeat count moves the point forward to the end of the first word it finds. (If the point is already in a word, then the command finds that word.) With a repeat count, it moves the point to the far end of the indicated word. **Backward Word** moves the point toward the beginning of the buffer to the far end of the indicated word. With a negative prefix argument, these commands move the point in the opposite direction. **Echo Area Backward Word** is careful to avoid damaging the prompt.

**Beginning of Line** **Ctrl-A** [*Command*]
**End of Line** **Ctrl-E** [*Command*]

These commands position the point at one end of a line. With no argument or with an argument of 0, the point is repositioned on the current line. If the argument is -1, the point is repositioned on the previous line. All negative arguments position the point on previous lines. If the argument is 1, then the point is put on the following line. Larger arguments move the point to later lines. These commands do not make the buffer larger; they move the point to the beginning or end of the buffer, but not beyond.

**Beginning of Buffer** *meta*-**<** [*Command*]
**End of Buffer** *meta*-**>** [*Command*]

These commands position the point at one end of the buffer. They save the point by pushing it onto the mark stack (see the section "Mark Commands"). They ignore the prefix argument.

**Move to Window Line** *meta*-**R** [*Command*]

This command moves the point to the beginning of a particular line in the window. The prefix argument specifies which line. A value of 0 indicates the top line of the window, and larger numbers indicate successive lines. A value of -1 indicates the bottom line, and larger (negative) numbers indicate earlier lines. The prefix argument for this command defaults to 0.

**Back to Indentation** *meta*-**M** [*Command*]
*meta*-**Ctrl-M**

This command moves the point to the first printing character on the current line.

**Scroll Window Down** **Ctrl-V** [*Command*]
**Scroll Window Up** *meta*-**V** [*Command*]
**Scroll Overlap** [*Variable*]

These commands show the next or the previous screenful of text in the buffer. **Scroll Overlap** determines how many lines from one screenful are also in the next screenful. If the new screenful does not contain the point, then the point is moved to a new position that is just barely in the Editor window. If a prefix argument is supplied, then these commands scroll the Editor window by that many lines.

**Next Page** **Ctrl-X]** [*Command*]
**Previous Page** **Ctrl-X[** [*Command*]
Pages are separated by the newpage character (control-L). These commands place the point at the beginning of a page. **Next Page** places it at the beginning of the next page, and **Previous Page** places it at the beginning of the current page. The prefix argument determines which page boundary is the new location of the point.

**Count Lines Page** **Ctrl-XL** [*Command*]
This command displays in the echo area three numbers: the number of lines on the current page, the number of lines on the current page before the point, and the number of lines on the current page after the point. This command ignores the prefix argument.

## Mark Commands

Various commands manipulate the mark in some manner. The Editor actually maintains the mark as a stack of marks ten deep, the top of which is the mark itself. Most commands that change the mark do so by pushing a new value for the mark onto the stack.

The mark and the point together define the region. Use these commands to make sure the region is correct before using a region command.

**Set/Pop Mark** **Ctrl-@** [*Command*]
This command performs various functions, depending on its prefix argument. It examines **Universal Argument Default** to interpret its prefix argument. If you supply a prefix argument whose value is different from the value of the **Universal Argument Default** or its square, an error is signaled.

**Ctrl-@** sets the mark to the current location of the point.

**Ctrl-UCtrl-@** moves the point to the mark and then pops the mark off the stack.

**Ctrl-UCtrl-UCtrl-@** pops the mark off the stack.

**Exchange Point and Mark** **Ctrl-XCtrl-X** [*Command*]
This command exchanges the point with the mark. It pops the old mark off the stack; it does not push the point on top of the old value.

**Mark Page** **Ctrl-XCtrl-P** [*Command*]
This command places the point at the beginning of a page and the mark at the end of the same page. Normally, this command uses the current page. A positive prefix argument indicates which following page to use; a negative one chooses a previous page.

**Mark Whole Buffer** **Ctrl-XH** [*Command*]

First, this command pushes the current location of the point onto the mark stack. Then it places the point at one end of the buffer and the mark at the other end. Normally, the point is placed at the beginning and the mark at the end. If there is a prefix argument, then the mark is placed at the beginning and the point at the end.

**Count Lines Region** [*Command*]

This command displays in the echo area the number of lines in the region. This command ignores the prefix argument.

## Modification Commands

This section describes the commands for changing the contents of a buffer. Roughly speaking, the commonly used commands come first. The commands for case modification, transposition, and modifying white space are useful, but occasions for using them arise less often.

### Insertion Commands

These commands put new characters at the point. (Remember that the point is insertion between characters.) The new location of the point is usually after the new character in the buffer. This means that inserting large blocks of text is just like typing in text in the normal way.

**Self Insert** *[Command]*

This command inserts the character into the buffer at the point. With a repeat count, that many copies of the character are inserted at the point. It is bound to all printing characters and to Space. This command is intended to be invoked only through a key binding.

**Quoted Insert** **Ctrl-Q** *[Command]*

Many characters, particularly control characters, have a special meaning to the Editor. This command is used to insert those characters into the buffer. It reads the next character from the keyboard and inserts that character, whatever it is, into the buffer at the point. If it has a repeat count, then it inserts that many copies of the character at the point.

**New Line** **Return** *[Command]*

This command moves the point to the beginning of a new line. This usually works like inserting a newline character at the point. However, if the point is at the end of the current line and the next two lines are blank, then this command moves the point to the beginning of the next line and cleans away any white space characters on that line. This command interprets a prefix argument as a repeat count.

**Open Line** **Ctrl-O** *[Command]*

This command inserts a newline character after the point. The prefix argument tells how many newline characters to insert.

## Deletion Commands

These commands are used to remove a small piece of text from the buffer. They all use the prefix argument as a count of how many characters to remove. To remove a large section of text, consider using a kill command instead.

**Delete Next Character** **Ctrl-D** [*Command*]
**Delete Previous Character** **Delete** **Backspace** [*Command*]
**Echo Area Delete Previous Character** **Delete** **Backspace** [*Command*]

These commands remove from the buffer a character that is next to the point. (Remember that the point is between two characters.) The echo-area version is particularly careful to avoid damaging the prompt.

**Delete Previous Character Expanding Tabs** [*Command*]

This command is exactly like **Delete Previous Character** except for its treatment of tab characters. This command converts a preceding tab character into some number of space characters before removing a character from the buffer.

## Killing and Unkilling

Like the deletion commands, the commands for killing remove text from the buffer. However, there is an important difference between these commands. The commands for killing save the text in a structure called the kill ring, which means that the text can be recovered. The recovery process, called unkilling, puts the text back at the point. Consecutive commands for killing save their text in the same chunk, so all that text can be restored with a single command.

When used with motion commands, commands for killing and unkilling can move chunks of text around in a buffer.

**Kill Line** **Ctrl-K** [*Command*]
**Backward Kill Line** [*Command*]

These commands remove a line of text. Without a prefix argument, **Kill Line** kills the text after the point on the current line. If the line is blank after the point, then the command kills any white space at the end of the line and kills the following newline character. Therefore, if the point is at the beginning of a nonblank line, issuing **Kill Line** twice completely removes the line. **Backward Kill Line** either removes all characters preceding the point on the current line or, if there are no characters, removes the preceding newline character and any trailing white space from the previous line.

The prefix argument is not really a repeat count for these commands. Instead, it tells how many lines to kill. A prefix argument of 16 causes 16 lines to be killed. If it were a repeat count, it would cause somewhere from 8 to 16 lines to be killed, depending on just how many of them were blank.

**Kill Next Word** *meta*-**D** [*Command*]
**Kill Previous Word** *meta*-**Delete** *meta*-**Backspace** [*Command*]
**Echo Area Kill Previous Word** *meta*-**Delete** *meta*-**Backspace** [*Command*]

These commands are related to **Forward Word** and **Backward Word**. The motion commands move the point to a new location. The killing commands make the point and the new location identical by killing the intervening text. The echo-area version of the command, as usual, is careful about the prompt.

**Kill Region** **Ctrl-W** [*Command*]
**Save Region** *meta*-**W** [*Command*]

These commands remove and restore regions of text. **Kill Region** kills the current region. It is a way to remove large pieces of text. **Save Region** makes a copy of the current region and stores that in the kill ring.

**Un-Kill** **Ctrl-Y** [*Command*]

This command restores the most recently killed portion of text. Using this command more than once produces multiple copies of the most recently killed text. A prefix argument indicates which portion of killed text should be restored.

**Rotate Kill Ring** *meta*-**Y** [*Command*]

This command kills the current region and replaces it with the next most recently killed portion of text. A prefix argument indicates which portion of text should be restored.

## Miscellaneous Modification Commands

You may use the following commands for other common modifications.

### Case Modification Commands

**Uppercase Word** *meta*-**U** [*Command*]
**Lowercase Word** *meta*-**L** [*Command*]
**Capitalize Word** *meta*-**C** [*Command*]

These commands modify the case of a word, as indicated. A prefix argument tells how many words to modify. A negative prefix argument says to modify words before the point.

## Transposition Commands

**Transpose Characters** **Ctrl-T** [*Command*]
**Transpose Words** *meta*-**T** [*Command*]
**Transpose Lines** **Ctrl-XCtrl-T** [*Command*]

These commands reorder pieces of text; they exchange the next piece with the previous piece. (If the point is inside an appropriate piece of text, that piece is used instead of one that follows.) **Transpose Characters** trades the letters before and after the point. The point is advanced one character position, so repeated use drags a character through the following text. **Transpose Words** trades the next word with the word before that. It affects only the words themselves; punctuation and white space are preserved. The point is put at the end of the new second word. **Transpose Lines** trades the current line and the previous line. The point is put at the beginning of the line following the two that are exchanged. These commands interpret the prefix argument as a repeat count.

## White Space Commands

**Delete Horizontal Space** [*Command*]

This command removes all space characters and tab characters on either side of the point.

**Delete Indentation** *meta*-**^** [*Command*]
*meta*-**Ctrl-^**

This command joins the current line to the previous line by deleting the newline character and any extra white space between them. Normally, exactly one space character is left in place of the newline character. However, there are three situations in which no white space is left behind: when the space character would be the first character of the line, when the space character would immediately follow a left parenthesis, or when the space character would immediately precede a right parenthesis.

**Just One Space** [*Command*]

This command deletes the space surrounding the point. Then it inserts one space character. The prefix argument tells how many space characters to insert at the point.

**Indent Rigidly** **Ctrl-XTab** [*Command*]

This command modifies the indentation for each line in the current region. The prefix argument tells how many space characters to insert in the indentation. A negative prefix argument says to delete space from the indentation. If possible, space characters in the indentation are converted to tab characters.

## Searching, Replacing, and Filtering

Search commands move the point to a place in the buffer that matches a given pattern. Replacement commands substitute one block of text for another uniformly in the buffer. Both search commands and replacement commands come in two varieties: interactive and noninteractive.

Filtering is a more general form of replacement. A ***filter*** is a Lisp function that maps strings into strings. The filter command uses a filter on several consecutive lines of the buffer.

### Search Commands

**Forward Search** [*Command*]
**Reverse Search** [*Command*]

These commands search for a given string in the current buffer. First they prompt for the string. Then they move the point to the far end of the string. These commands, particularly in large buffers, are faster than the interactive search commands. These commands push their starting location onto the mark stack. These commands ignore the prefix argument.

**Incremental Search** **Ctrl-S** [*Command*]
**Reverse Incremental Search** **Ctrl-R** [*Command*]

These commands invoke an interactive search mechanism. They prompt for a search string. After each character of the search string is typed, the search mechanism moves the point to the first instance in the appropriate direction of the current search string. If there is no instance of the search string, then the search fails and the point stays in the same place. Printing characters, those bound to **Self Insert**, are added to the search string. Almost any other character stops the searching and issues another command through its key binding. The following characters have special meaning to the search routines:

| | |
|---|---|
| **Ctrl-S** or **Ctrl-R** | If the current search string is null, insert the default search string. (The default is whatever was used last time.) Then look for the next instance of the current search string in the indicated direction—forward for **Ctrl-S** or backward for **Ctrl-R**. |
| **Ctrl-Q** | Append the next character to the search string and continue searching. This prevents the next character from being interpreted as a command. |
| **Delete** | Undo the effect of the last character typed. |

|  |  |
|---|---|
| **Ctrl-G** | If the search is currently succeeding, that is, if it found something after the last character was typed, end the searching and return the point to its location when the search was started. If the search is currently failing, return the point to the last location where a successful search ended. |
| **Esc** | Exit search at the current location. |

## Replacement Commands

**Replace String** [*Command*]

This command prompts for old text and then for new text. It replaces all instances of the old text following the point with the new text. A prefix argument specifies how many substitutions to perform. If there are not enough instances of the old text to allow that many substitutions, the Editor makes as many substitutions as it can.

**Query Replace** *meta-%* [*Command*]

This command prompts for old text, then for new text. It looks for the old text in the current buffer after the point. Whenever it finds the old text, it prompts you for the next action. These are the available choices:

|  |  |
|---|---|
| **Y**<br>**Space** | Do the replacement and keep searching. |
| **N**<br>**Delete** | Do not perform the replacement but keep searching. |
| **!** | Replace this instance and all later ones. |
| **.** | Replace this instance but do not look for any more. |
| **Esc** | Do not replace this one; do not look for any more. |
| **?**<br>**H** | Show a list of available options. |

## Filtering

Filtering can cause regular changes that are more complicated than just substitution. A filter is a Lisp function that takes a single string as an argument and returns a string as its result.

**Filter Region** [*Command*]

This command prompts for a filter. It then replaces each line in the region with the string that is returned as a result of calling the filter. This command ignores the prefix argument.

## Buffers, Files, and Editor Windows

The Editor supports multiple buffers and Editor windows. Each buffer may be displayed in zero or one or more Editor windows. Different Editor windows can display different parts of the same buffer. The current Editor window is the Editor window that contains the point. The current buffer is displayed in the current Editor window.

### Buffer Commands

**Select Previous Buffer** *meta*-**Ctrl-L** [*Command*]
**Select Buffer** **Ctrl-XB** [*Command*]

These commands find a different buffer to display in the current Editor window. **Select Previous Buffer** chooses the buffer that was most recently displayed. **Select Buffer** prompts for the name of a buffer to use. The default choice of buffer is the previous buffer. **Select Buffer** creates a new buffer if the indicated buffer does not exist.

**Kill Buffer** **Ctrl-XK** [*Command*]

This command destroys a buffer. It prompts for the buffer to destroy. Since the contents of the buffer disappear forever when the buffer is destroyed, the Editor first gives you the opportunity to save the buffer in a file. If the current buffer is destroyed, the previous buffer replaces the current buffer in all Editor windows that displayed the current buffer. If any noncurrent buffer is destroyed, all Editor windows that displayed that buffer are destroyed also.

**List Buffers** **Ctrl-XCtrl-B** [*Command*]

This command creates or selects the buffer ***buffer list***. It puts a list of all the existing buffers into that buffer along with certain useful information about each buffer. Finally, it displays that buffer in another Editor window.

**Buffer Not Modified** *meta-~* [*Command*]

A buffer is considered modified if it is different from the most recent permanent version of the buffer. This means that a buffer is unmodified right after reading it in from a file or right after writing it out to a file. Some sequences of commands leave the buffer exactly the same but convince the Editor that the buffer is now modified. For example, killing a portion of text and then immediately unkilling it leaves the buffer the same as it was before. The Editor offers you an opportunity to save modified buffers before destroying them.

The command **Buffer Not Modified** tells the Editor that the current buffer has not really changed.

**Revert Buffer** *[Command]*

This command discards the text in the buffer and replaces it with the text from the corresponding file. (There must be a corresponding file.) It first queries you to make sure this is not a mistake. It ignores its prefix argument.

**Rename Buffer** *[Command]*

This command prompts for a new name for the current buffer. The default value for the name is the same as the name of the associated file, if any. This command then changes the name of the current buffer to the indicated value.

**Insert Buffer** *[Command]*

This command prompts for the name of a buffer. It then inserts the contents of the indicated buffer at the point.

## File Commands

**Visit File** **Ctrl-XCtrl-V** *[Command]*

This command replaces the contents of the current buffer with the contents of the indicated file. If the current buffer is modified, this command gives you a chance to save the changes. Then it prompts for the name of a file. Finally, it reads the contents of that file into the current buffer.

**Find File** **Ctrl-XCtrl-F** *[Command]*

This command displays the indicated file in a buffer. First it prompts for the name of a file to display. If that file already has an associated buffer, then it chooses that buffer and displays it. (There are several options available if that buffer contains an old version of the file.) Otherwise, the Editor creates a new buffer and reads the indicated file into that buffer. There are additional choices if the default name for that buffer is already in use.

**Backup File** *[Command]*

**Save File** **Ctrl-XCtrl-S** *[Command]*

**Write File** **Ctrl-XCtrl-W** *[Command]*

These commands write the contents of the current buffer into a file. **Save File** writes it in the file associated with the buffer, if there is one; otherwise it prompts for the name of the file to use. **Write File** always prompts for the name of the file. It uses the name of the associated file as a default. Both commands set the associated file to the one used. They also change the name of the buffer to match if that buffer name is not already in use. **Backup File** prompts like **Write File** but does not change the associated file or the name of the buffer.

**Save All Files** *[Command]*

**Save All Files and Exit** **Ctrl-XCtrl-C** *[Command]*

These commands save modified buffers in their associated files. If a buffer is not marked as modified, it is not saved; if a buffer does not have an associated file, it is not saved. After all files are saved, **Save All Files and Exit** returns you to Lisp.

**Insert File** *[Command]*

This command inserts the contents of a file at the point. It prompts for the name of the file.

## File System Commands

**Delete File** *[Command]*

This command prompts for the name of a file and then tries to delete that file from the external file system. It ignores the prefix argument.

**Rename File** *[Command]*

This command prompts for the current name of a file and for a new name for that file. It then tries to rename that file in the external file system. This command ignores the prefix argument.

**Copy File** *[Command]*

This command prompts for the name of an existing file and for the name of a new file. It then tries to copy the first file into the second file. This command ignores its prefix argument.

**Directory** *[Command]*

This command prompts for a pathname, which may contain wildcards. Then it tries to print a directory list of the indicated files. This command ignores the prefix argument.

## Editor Window Commands

The Editor divides the screen into Editor windows. If you have used the function **ed** to invoke the Editor from the **suntools** environment, the Editor windows are actually Window Tool Kit windows and can be manipulated as such. (See the chapter "The Window Tool Kit.") Each editor window displays part of some buffer and has a mode line that describes some details of that buffer.

**Refresh Screen** **Ctrl-L** *[Command]*

This command combines two functions. It clears and redisplays the entire screen, and it repositions the current window. With no prefix argument, it centers the current line in the current window. A prefix argument specifies the new location in the current window of the current line; nonnegative prefix arguments indicate lines counting from the top of the window (0 means the first line), and negative prefix arguments indicate lines counting from the bottom of the window (-1 means the bottom line).

**Next Window** **Ctrl-XN** [*Command*]
**Ctrl-XO**
**Previous Window** **Ctrl-XP** [*Command*]

These commands move the point to another Editor window. The windows are kept in a circular list. **Next Window** moves through the list in one direction, and **Previous Window** moves in the other. These commands ignore the prefix argument.

**New Window** **Ctrl-X2** [*Command*]

This command splits the currently selected Editor window on the screen into two pieces. The new window displays the current buffer. This command ignores the prefix argument.

**Delete Window** **Ctrl-XD** [*Command*]
**Delete Next Window** **Ctrl-X1** [*Command*]

These commands delete Editor windows from the display. The area of the deleted Editor windows is added to adjacent Editor windows. These commands ignore the prefix argument.

**Enlarge Window** **Ctrl-X^** [*Command*]

This command makes the current Editor window larger. The prefix argument, if present, specifies how much to expand the Editor window.

**Line to Top of Window** [*Command*]
**Line to Center of Window** [*Command*]

These commands modify the display so that the current line is in the indicated position of the current Editor window. These commands ignore the prefix argument.

**Scroll Window Down** **Ctrl-V** [*Command*]
**Scroll Window Up** ***meta*-V** [*Command*]
**Scroll Next Window Down** [*Command*]
**Scroll Next Window Up** [*Command*]

These commands display an adjacent part of the buffer in the designated editor window. **Scroll Window Down** and **Scroll Window Up** are explained in detail in the section "Motion Commands." The "next window" versions of these commands behave in the same manner but affect the next Editor window instead of the current Editor window.

## Editing Lisp

This section describes commands to move around the buffer by units of Lisp expressions instead of by units of words or lines. It also describes commands that modify and format Lisp code.

**LISP Mode** [*Command*]

This command sets the major mode of the Editor to Lisp mode. This alters some of the key bindings. Certain commands are made more easily available for formatting Lisp programs. This command ignores the prefix argument.

**Forward List** *meta*-**Ctrl-N** [*Command*]

**Backward List** *meta*-**Ctrl-P** [*Command*]

These commands operate on balanced sets of parentheses. Note that the balanced sets of parentheses may not be lists in the Lisp sense; some of the parentheses may be in strings or in comments. **Forward List** searches forward for a parenthesis. If it finds a right parenthesis, then it places the point just after that parenthesis; if it finds a left parenthesis, then it places the point just after the corresponding right parenthesis. **Backward List** works similarly, but in the opposite direction. Both commands interpret the prefix argument as a repeat count.

**Forward Form** *meta*-**Ctrl-F** [*Command*]

**Backward Form** *meta*-**Ctrl-B** [*Command*]

These commands try to find the right form in the buffer and then move the point to the far end of that form. For these commands, a form is considered to be a list, a symbol, or a quoted string. Each form may be preceded by certain characters that are significant to Lisp and that are a part of the form. Again, the Editor does not really understand strings and comments. The prefix argument is a repeat count for these commands.

**Backward Up List** *meta*-**Ctrl-(** [*Command*]
*meta*-**Ctrl-U**

**Forward Up List** *meta*-**Ctrl-)** [*Command*]

These commands move the point to a place just outside a containing list. The prefix argument is a repeat count.

**Down List** *meta*-**Ctrl-D** [*Command*]

This command moves the point forward past a left parenthesis. The prefix argument is a repeat count.

**Move Over )** *meta*-**)** [*Command*]

This command searches for the next right parenthesis. It deletes any horizontal white space just before the parenthesis and inserts a newline character just after the parenthesis. This command ignores the prefix argument.

**Beginning of Defun** *meta*-**Ctrl-[** *[Command]*
*meta*-**Ctrl-A**

**End of Defun** *meta*-**Ctrl-]** *[Command]*
*meta*-**Ctrl-E**

These commands act on sets of balanced parentheses whose first left parenthesis is in column 1 of the buffer. (It need not actually be a **defun**; the Editor does not examine the first element of the list.) These commands place the point just outside the appropriate object. The prefix argument is a repeat count.

**Insert ()** *meta*-**(** *[Command]*

This command introduces a balanced pair of parentheses. The left parenthesis is inserted at the point. If there is no prefix argument, the right parenthesis is inserted right after it. If there is a positive prefix argument, the closing parenthesis is inserted after that many following forms. A negative prefix argument causes an error.

**Lisp Insert )** **)** *[Command]*

**Paren Pause Period** *[Variable]*

This command inserts a right parenthesis at the point. It searches backward for the balancing left parenthesis. If the left parenthesis is displayed on the screen, the cursor is moved to it for **Paren Pause Period** seconds. If it is not displayed on the screen, then the line of text containing the balancing left parenthesis is displayed in the echo area. This command ignores its prefix argument.

**Lisp New Line** **Linefeed** *[Command]*

This command deletes horizontal space at the point, inserts a newline character, and moves the point to the current indentation on the new line. The prefix argument controls how many newline characters are inserted.

**Mark Form** *meta*-**Ctrl-@** *[Command]*

This command pushes a new mark on the mark stack. It does not move the point. The new mark is where **Forward Form**, with the same prefix argument, would place the point.

**Mark Defun** *meta*-**Ctrl-H** *[Command]*

This command places the point just before a top-level form and places the mark just after the top-level form. (It pushes the mark on the mark stack.) If the point is inside a top-level form when this command is issued, that form is the one that is used; otherwise the next top-level form after the point is used. This command ignores the prefix argument.

**Forward Kill Form** *meta*-**Ctrl-K** *[Command]*

**Backward Kill Form** *meta*-**Ctrl-Delete** *[Command]*

These commands are related to **Forward Form** and **Backward Form**. They kill the text from the point to the place where the corresponding motion command would put the point. The prefix argument is used as a repeat count. A negative prefix argument changes the direction of the killing. Notice that this is different from passing the prefix argument to the motion command in a case where the motion command would fail.

**Extract List** *meta*-**Ctrl-X** [*Command*]

This command finds the smallest list containing the point and saves that list. Then it finds the next larger surrounding list and kills that list. The smaller list is then inserted where the bigger list used to be. If there is a prefix argument, it controls how many times **Extract List** finds the next larger surrounding list before killing the chunk of text.

**Transpose Forms** *meta*-**Ctrl-T** [*Command*]

This command exchanges the next or current form and the previous form. The point is placed after the new second form. The prefix argument is interpreted as a repeat count.

**Defindent** [*Command*]

Normally, all of the arguments in a form are indented the same amount. Certain symbols cause the first few arguments to have different (greater) indentation. For example, in a form beginning with the symbol **do**, the first argument is a list of variable bindings and the second is a termination condition; these arguments are generally indented farther than the remaining arguments.

The command **Defindent** sets the number of special arguments that are to be associated with the symbol at the beginning of the current list. It sets that value to the prefix argument, if present; otherwise it sets the value to 0.

**Indent for Lisp** **Tab** [*Command*]

This command indents the current line according to Lisp conventions. If there is a prefix argument, it indents that many lines.

**Indent Form** *meta*-**Ctrl-Q** [*Command*]

This command indents the current form according to Lisp conventions. It ignores the prefix argument.

**Lisp Indent Region** *meta*-**Ctrl-\** [*Command*]

This command indents all lines between the point and mark according to Lisp convention. It ignores the prefix argument.

## Interacting with Lisp

This section describes the commands that allow you to use Lisp from the Editor. The compilation and evaluation commands allow you to test changes to code without leaving the Editor. The top-level commands let you use the Editor as an intermediary between you and Lisp. They let you store a record of the Lisp session and bring the full strength of the Editor to typing and modifying expressions.

### Evaluation and Compilation

**Process File Options** [*Command*]
**Set Buffer Package** [*Command*]

Each buffer has associated with it a Lisp package that is current for any Lisp compilations or evaluations. The default package is the same as the current package in the running Lisp image. The default can be changed in various ways. Whenever it reads a Lisp file into a buffer, the Editor tries to parse the first line as a comment specifying the correct Editor mode for the corresponding buffer and the correct package to associate with the buffer.

The command **Process File Options** forces the Editor to reread the first line of the buffer to set those values. It ignores the prefix argument.

The command **Set Buffer Package** prompts for the name of a package and sets it for the current buffer. This command overrides anything that the other mechanisms may have set. It ignores the prefix argument.

**Evaluate Defun** **Ctrl-CCtrl-E** [*Command*]

This command uses Lisp to evaluate the current or next top-level form and prints the result in the echo area. It ignores the prefix argument.

**Evaluate Expression** *meta*-**Esc** [*Command*]

This command prompts for an expression. That expression is evaluated in Lisp environment, and the result is printed in the echo area. This command ignores the prefix argument.

**Evaluate Region** [*Command*]

This command passes the current region to Lisp for evaluation. The results are not printed to the screen. This command ignores the prefix argument.

**Evaluate Buffer** [*Command*]

This command passes the entire buffer to Lisp for evaluation. The output from that evaluation is printed in the echo area. This command ignores the prefix argument.

**Compile Defun** **Ctrl-CCtrl-C** [*Command*]
This command finds the current or next top-level form and passes it to Lisp for compilation. This command ignores the prefix argument.

**Compile Region** [*Command*]
This command passes the current region to Lisp for compilation. This command ignores the prefix argument.

**Compile Buffer** [*Command*]
This command passes the entire buffer to Lisp for compilation. It ignores the prefix argument.

**Compile File** [*Command*]
This command compiles a file and directs diagnostic output to the buffer ***Compiler Warnings***. Normally, the file chosen for compilation is the one associated with the current buffer. (The buffer is saved first if it has been modified.) If there is a prefix argument or if the current buffer has no associated file, this command prompts for the name of the file to compile.

**Load File** [*Command*]
**Load Pathname Defaults** [*Variable*]
This command prompts for the name of a file and causes Lisp to load that file. **Load Pathname Defaults** is the default value for the name of the file. It is normally whatever the name was in the last call. This command ignores the prefix argument.

## Top-Level Mode

Top-level mode makes the current buffer behave like a window into Lisp. You edit the end of the buffer (either by inserting text or by using more complicated procedures). When you give the appropriate command, the last part of the buffer is passed to Lisp for evaluation. The output of the evaluation is inserted at the end of the buffer. Then you can edit the end of the buffer.

The key **Ctrl-C** receives special treatment in this mode. It interrupts any ongoing evaluation and invokes the Debugger in the current window.

**Top-Level Mode** [*Command*]
This command changes the minor mode of the current buffer. If the buffer is not in top-level mode, this command puts it in that mode; if it is in top-level mode, this command takes it out of that mode. This command ignores the prefix argument.

**Top-Level Eval** **Ctrl-]** [*Command*]
This command finds the appropriate piece of the end of the buffer and passes it to Lisp for evaluation. Any output from that evaluation is appended to the end of the buffer. Then a prompt is printed in the buffer.

The appropriate piece of the buffer is normally everything from the last prompt to the end of the buffer. However, if the point is somewhere before that prompt, **Top-Level Eval** chooses the last form that ends on the current line. Any text after the last prompt is killed, and that form is copied to the end of the buffer. This command ignores the prefix argument.

**Previous Top-Level Input** *meta*-**P** [*Command*]
**Next Top-Level Input** *meta*-**N** [*Command*]
The Editor maintains a ring of recent inputs to the top level of Lisp. These commands insert the requested input at the point. (The point should be after the most recent prompt.) The prefix argument determines which input is inserted. A prefix argument of 0 displays the ring of recent inputs.

**Kill Top-Level Input** **Ctrl-W** [*Command*]
This command kills all the text between the point and the most recent prompt. The point should not precede the prompt. This command ignores the prefix argument.

**Top-Level Beginning of Line** **Ctrl-A** [*Command*]
This command is very much like **Beginning of Line**. In the special case when the point is on the same line with the prompt and the point is being moved to the beginning of that line, the point is placed just after the prompt instead.

**Line Buffered Input** [*Variable*]
This variable determines when input in top-level mode is read and evaluated by Lisp. If its value is **t**, input is read and evaluated by Lisp after either the **New Line** command or the **Top-Level Eval** command is executed; the default value is **t**. If the value of this variable is **nil**, input is read and evaluated by Lisp as soon as a complete expression is typed.

## Customizing the Editor

There are several ways to modify the Editor's behavior. Most Editor commands are issued by using bindings between commands and particular keys. The Editor has commands to change these key bindings. Several details in the Editor's operation are controlled by variables, and there are commands that can change these variables. Finally, there are commands that define and execute keyboard macros. These allow you to define your own commands.

**Bind Key** [*Command*]
**Delete Key Binding** [*Command*]

These commands alter specific key bindings. **Bind Key** prompts for the name of a command. Then it prompts for the particular key to bind to this command and for the scope of the particular binding. The binding may be global, or it may be restricted either to a particular mode or to a particular buffer. If the binding is to be restricted to a particular buffer, the command prompts for the particular buffer involved. If the restriction is to a particular mode, then **Bind Key** prompts for that mode.

**Delete Key Binding** prompts for a key and then for a scope. It then removes any applicable key binding for that key.

While these commands can alter any binding, certain bindings should be left intact. Binding a key that is a prefix of another key sequence masks those bindings. Do not bind **Ctrl-X** or *meta*-**X** or **Ctrl-Z**. For a similar reason, do not bind *meta*-[; it is a prefix for function keys on certain terminals. Do not rebind *meta*-**O**; it is a prefix for function keys on the VT-100.

Do not bind **Ctrl-G** (or **Ctrl-C** in the Editor's top-level mode). These keys are treated specially.

**Set Variable** [*Command*]

This command prompts for the name of an Editor variable and then prompts for a new value for that variable. After collecting the information, it sets the indicated variable to the desired value.

**Define Keyboard Macro** **Ctrl-X(** [*Command*]
**End Keyboard Macro** **Ctrl-X)** [*Command*]

These commands are used to define a keyboard macro. A keyboard macro is a sequence of commands that are bundled together as a single unit. **Define Keyboard Macro** starts the definition process and **End Keyboard Macro** finishes it. The commands in between are executed as they are typed and are stored away for future use. These commands ignore their prefix arguments.

**Keyboard Macro Query** Ctrl-XQ [*Command*]

This command gives you some control over the execution of a macro. During the definition of a keyboard macro, this command has no effect. During the execution of a keyboard macro, this command prompts for a single character that determines whether the rest of the macro is to be executed this time and whether the macro is to be executed again after it reaches completion. These characters are interpreted just like the characters in **Query Replace**. This command ignores its prefix argument.

**Last Keyboard Macro** Ctrl-XE [*Command*]

This command executes the most recently defined keyboard macro. The prefix argument tells how many times to execute the macro. An Editor error stops all execution of keyboard macros.

**Name Keyboard Macro** [*Command*]

This command defines a new Editor command. It prompts for the name of the new command, and that name becomes the name of the most recently defined keyboard macro, which can then be bound to a key. This allows you to have more than one keyboard macro available at the same time. This command ignores its prefix argument.

## Editor Commands and Key Bindings

| | | |
|---|---|---|
| **Apropos** | | [*Command*] |
| **Argument Digit** | *meta-digit* | [*Command*] |
| **Back to Indentation** | *meta*-**M**<br>*meta*-**Ctrl-M** | [*Command*] |
| **Backup File** | | [*Command*] |
| **Backward Character** | **Ctrl-B** | [*Command*] |
| **Backward Form** | *meta*-**Ctrl-B** | [*Command*] |
| **Backward Kill Form** | *meta*-**Ctrl-Delete** | [*Command*] |
| **Backward Kill Line** | | [*Command*] |
| **Backward List** | *meta*-**Ctrl-P** | [*Command*] |
| **Backward Up List** | *meta*-**Ctrl-(**<br>*meta*-**Ctrl-U** | [*Command*] |
| **Backward Word** | *meta*-**B** | [*Command*] |
| **Beginning Of Parse** | **Ctrl-A** | [*Command*] |
| **Beginning of Buffer** | *meta*-**<** | [*Command*] |
| **Beginning of Defun** | *meta*-**Ctrl-[**<br>*meta*-**Ctrl-A** | [*Command*] |
| **Beginning of Line** | **Ctrl-A** | [*Command*] |
| **Bind Key** | | [*Command*] |
| **Buffer Not Modified** | *meta*-~ | [*Command*] |
| **Capitalize Word** | *meta*-**C** | [*Command*] |
| **Compile Buffer** | | [*Command*] |
| **Compile Defun** | **Ctrl-CCtrl-C** | [*Command*] |
| **Compile File** | | [*Command*] |
| **Compile Region** | | [*Command*] |
| **Complete Field** | **Space** | [*Command*] |
| **Complete Keyword** | **Tab** | [*Command*] |
| **Confirm Parse** | **Return**<br>**Linefeed** | [*Command*] |
| **Copy File** | | [*Command*] |
| **Count Lines Page** | **Ctrl-XL** | [*Command*] |
| **Count Lines Region** | | [*Command*] |
| **Defindent** | | [*Command*] |
| **Define Keyboard Macro** | **Ctrl-X(** | [*Command*] |
| **Delete File** | | [*Command*] |
| **Delete Horizontal Space** | | [*Command*] |
| **Delete Indentation** | *meta*-^<br>*meta*-**Ctrl-^** | [*Command*] |

| | | |
|---|---|---|
| **Delete Key Binding** | | [*Command*] |
| **Delete Next Character** | **Ctrl-D** | [*Command*] |
| **Delete Next Window** | **Ctrl-X1** | [*Command*] |
| **Delete Previous Character Expanding Tabs** | | [*Command*] |
| **Delete Previous Character** | **Delete** **Backspace** | [*Command*] |
| **Delete Window** | **Ctrl-XD** | [*Command*] |
| **Describe Command** | | [*Command*] |
| **Describe Key** | | [*Command*] |
| **Directory** | | [*Command*] |
| **Down List** | *meta*-**Ctrl-D** | [*Command*] |
| **Echo Area Backward Character** | **Ctrl-B** | [*Command*] |
| **Echo Area Backward Word** | *meta*-**B** | [*Command*] |
| **Echo Area Delete Previous Character** | **Delete** **Backspace** | [*Command*] |
| **Echo Area Kill Previous Word** | *meta*-**Delete** *meta*-**Backspace** | [*Command*] |
| **End Keyboard Macro** | **Ctrl-X)** | [*Command*] |
| **End of Buffer** | *meta*-**>** | [*Command*] |
| **End of Defun** | *meta*-**Ctrl-]** *meta*-**Ctrl-E** | [*Command*] |
| **End of Line** | **Ctrl-E** | [*Command*] |
| **Enlarge Window** | **Ctrl-X^** | [*Command*] |
| **Evaluate Buffer** | | [*Command*] |
| **Evaluate Defun** | **Ctrl-CCtrl-E** | [*Command*] |
| **Evaluate Expression** | *meta*-**Esc** | [*Command*] |
| **Evaluate Region** | | [*Command*] |
| **Exchange Point and Mark** | **Ctrl-XCtrl-X** | [*Command*] |
| **Exit Editor** | **Ctrl-XCtrl-Z** | [*Command*] |
| **Extended Command** | *meta*-**X** | [*Command*] |
| **Extract List** | *meta*-**Ctrl-X** | [*Command*] |
| **Filter Region** | | [*Command*] |
| **Find File** | **Ctrl-XCtrl-F** | [*Command*] |
| **Forward Character** | **Ctrl-F** | [*Command*] |
| **Forward Form** | *meta*-**Ctrl-F** | [*Command*] |
| **Forward Kill Form** | *meta*-**Ctrl-K** | [*Command*] |
| **Forward List** | *meta*-**Ctrl-N** | [*Command*] |
| **Forward Search** | | [*Command*] |
| **Forward Up List** | *meta*-**Ctrl-)** | [*Command*] |
| **Forward Word** | *meta*-**F** | [*Command*] |

| | | |
|---|---|---|
| **Fundamental Mode** | | [*Command*] |
| **Generic Describe** | | [*Command*] |
| **Help on Prompt** | **Ctrl-_**<br>***meta*-H** | [*Command*] |
| **Help** | ***meta*-?**<br>***meta*-H**<br>**Ctrl-_** | [*Command*] |
| **Incremental Search** | **Ctrl-S** | [*Command*] |
| **Indent Form** | ***meta*-Ctrl-Q** | [*Command*] |
| **Indent Rigidly** | **Ctrl-XTab** | [*Command*] |
| **Indent for Lisp** | **Tab** | [*Command*] |
| **Insert ()** | ***meta*-(** | [*Command*] |
| **Insert Buffer** | | [*Command*] |
| **Insert File** | | [*Command*] |
| **Insert Parse Default** | | [*Command*] |
| **Just One Space** | | [*Command*] |
| **Keyboard Macro Query** | **Ctrl-XQ** | [*Command*] |
| **Kill Buffer** | **Ctrl-XK** | [*Command*] |
| **Kill Line** | **Ctrl-K** | [*Command*] |
| **Kill Next Word** | ***meta*-D** | [*Command*] |
| **Kill Parse** | **Ctrl-W** | [*Command*] |
| **Kill Previous Word** | ***meta*-Delete**<br>***meta*-Backspace** | [*Command*] |
| **Kill Region** | **Ctrl-W** | [*Command*] |
| **Kill Top-Level Input** | **Ctrl-W** | [*Command*] |
| **LISP Mode** | | [*Command*] |
| **Last Keyboard Macro** | **Ctrl-XE** | [*Command*] |
| **Line to Center of Window** | | [*Command*] |
| **Line to Top of Window** | | [*Command*] |
| **Lisp Indent Region** | ***meta*-Ctrl-\** | [*Command*] |
| **Lisp Insert )** | **)** | [*Command*] |
| **Lisp New Line** | **Linefeed** | [*Command*] |
| **List Buffers** | **Ctrl-XCtrl-B** | [*Command*] |
| **Load File** | | [*Command*] |
| **Lowercase Word** | ***meta*-L** | [*Command*] |
| **Mark Defun** | ***meta*-Ctrl-H** | [*Command*] |
| **Mark Form** | ***meta*-Ctrl-@** | [*Command*] |
| **Mark Page** | **Ctrl-XCtrl-P** | [*Command*] |
| **Mark Whole Buffer** | **Ctrl-XH** | [*Command*] |
| **Move Over )** | ***meta*-)** | [*Command*] |

| | | |
|---|---|---|
| **Move to Window Line** | *meta*-**R** | [*Command*] |
| **Name Keyboard Macro** | | [*Command*] |
| **Negative Argument** | *meta*-- | [*Command*] |
| **New Line** | **Return** | [*Command*] |
| **New Window** | **Ctrl-X2** | [*Command*] |
| **Next Line** | **Ctrl-N** | [*Command*] |
| **Next Page** | **Ctrl-X]** | [*Command*] |
| **Next Parse** | **Ctrl-N** | [*Command*] |
| **Next Top-Level Input** | *meta*-**N** | [*Command*] |
| **Next Window** | **Ctrl-XN** | [*Command*] |
| | **Ctrl-XO** | |
| **Open Line** | **Ctrl-O** | [*Command*] |
| **Previous Line** | **Ctrl-P** | [*Command*] |
| **Previous Page** | **Ctrl-X[** | [*Command*] |
| **Previous Parse** | **Ctrl-P** | [*Command*] |
| **Previous Top-Level Input** | *meta*-**P** | [*Command*] |
| **Previous Window** | **Ctrl-XP** | [*Command*] |
| **Process File Options** | | [*Command*] |
| **Query Replace** | *meta*-**%** | [*Command*] |
| **Quoted Insert** | **Ctrl-Q** | [*Command*] |
| **Refresh Screen** | **Ctrl-L** | [*Command*] |
| **Rename Buffer** | | [*Command*] |
| **Rename File** | | [*Command*] |
| **Replace String** | | [*Command*] |
| **Reverse Incremental Search** | **Ctrl-R** | [*Command*] |
| **Reverse Search** | | [*Command*] |
| **Revert Buffer** | | [*Command*] |
| **Rotate Kill Ring** | *meta*-**Y** | [*Command*] |
| **Save All Files and Exit** | **Ctrl-XCtrl-C** | [*Command*] |
| **Save All Files** | | [*Command*] |
| **Save File** | **Ctrl-XCtrl-S** | [*Command*] |
| **Save Region** | *meta*-**W** | [*Command*] |
| **Scroll Next Window Down** | | [*Command*] |
| **Scroll Next Window Up** | | [*Command*] |
| **Scroll Window Down** | **Ctrl-V** | [*Command*] |
| **Scroll Window Up** | *meta*-**V** | [*Command*] |
| **Select Buffer** | **Ctrl-XB** | [*Command*] |
| **Select Previous Buffer** | *meta*-**Ctrl-L** | [*Command*] |
| **Self Insert** | | [*Command*] |

|  |  |  |
|---|---|---|
| **Set Buffer Package** |  | [*Command*] |
| **Set Variable** |  | [*Command*] |
| **Set/Pop Mark** | **Ctrl-@** | [*Command*] |
| **Top-Level Beginning of Line** | **Ctrl-A** | [*Command*] |
| **Top-Level Eval** | **Ctrl-]** | [*Command*] |
| **Top-Level Mode** |  | [*Command*] |
| **Transpose Characters** | **Ctrl-T** | [*Command*] |
| **Transpose Forms** | *meta*-**Ctrl-T** | [*Command*] |
| **Transpose Lines** | **Ctrl-XCtrl-T** | [*Command*] |
| **Transpose Words** | *meta*-**T** | [*Command*] |
| **Un-Kill** | **Ctrl-Y** | [*Command*] |
| **Universal Argument** | **Ctrl-U** | [*Command*] |
| **Uppercase Word** | *meta*-**U** | [*Command*] |
| **View Lossage** |  | [*Command*] |
| **Visit File** | **Ctrl-XCtrl-V** | [*Command*] |
| **Where Is** |  | [*Command*] |
| **Write File** | **Ctrl-XCtrl-W** | [*Command*] |

# Appendix A. Alphabetical Listing of Common Lisp Functions

This appendix is a listing of all the Common Lisp functions, macros, constants, variables, special forms, and extensions to Common Lisp described in the *Sun Common Lisp Reference Manual.*

| | |
|---|---|
| ***** &rest *numbers* | [*Function*] |
| ***** | [*Variable*] |
| ****** | [*Variable*] |
| ******* | [*Variable*] |
| **+** &rest *numbers* | [*Function*] |
| **+** | [*Variable*] |
| **++** | [*Variable*] |
| **+++** | [*Variable*] |
| **−** *number* &rest *more-numbers* | [*Function*] |
| **−** | [*Variable*] |
| **/** *number* &rest *more-numbers* | [*Function*] |
| **/** | [*Variable*] |
| **//** | [*Variable*] |
| **///** | [*Variable*] |
| **/=** *number* &rest *more-numbers* | [*Function*] |
| **1+** *number* | [*Function*] |
| **1−** *number* | [*Function*] |
| **<** *number* &rest *more-numbers* | [*Function*] |
| **<=** *number* &rest *more-numbers* | [*Function*] |
| **=** *number* &rest *more-numbers* | [*Function*] |
| **>** *number* &rest *more-numbers* | [*Function*] |
| **>=** *number* &rest *more-numbers* | [*Function*] |

| | |
|---|---|
| **abort** `&optional` *status* | [*Function*] |
| **abs** *number* | [*Function*] |
| **acons** *key datum a-list* | [*Function*] |
| **acos** *number* | [*Function*] |
| **acosh** *number* | [*Function*] |
| **adjoin** *item list* `&key :test :test-not :key` | [*Function*] |
| **adjust-array** *array new-dimensions* `&key :element-type :initial-element :initial-contents :fill-pointer :displaced-to :displaced-index-offset` | [*Function*] |
| **adjustable-array-p** *array* | [*Function*] |
| **alpha-char-p** *char* | [*Function*] |
| **alphanumericp** *char* | [*Function*] |
| **and** {*form*}* | [*Macro*] |
| **append** `&rest` *lists* | [*Function*] |
| **apply** *function arg* `&rest` *more-args* | [*Function*] |
| **applyhook** *function args evalhookfn applyhookfn* `&optional` *env* | [*Function*] |
| ***applyhook*** | [*Variable*] |
| **apropos** *string* `&optional` *package* | [*Function*] |
| **apropos-list** *string* `&optional` *package* | [*Function*] |
| **aref** *array* `&rest` *subscripts* | [*Function*] |
| **array-dimension** *array axis-number* | [*Function*] |
| **array-dimension-limit** | [*Constant*] |
| **array-dimensions** *array* | [*Function*] |
| **array-element-type** *array* | [*Function*] |
| **array-has-fill-pointer-p** *array* | [*Function*] |
| **array-in-bounds-p** *array* `&rest` *subscripts* | [*Function*] |
| **array-rank** *array* | [*Function*] |
| **array-rank-limit** | [*Constant*] |
| **array-row-major-index** *array* `&rest` *subscripts* | [*Function*] |

| | |
|---|---|
| **array-total-size** *array* | [*Function*] |
| **array-total-size-limit** | [*Constant*] |
| **arrayp** *object* | [*Function*] |
| **ash** *integer count* | [*Function*] |
| **asin** *number* | [*Function*] |
| **asinh** *number* | [*Function*] |
| **assert** *test-form* [({*place*}*) [*format-string* {*arg*}*]] | [*Macro*] |
| **assoc** *item a-list* `&key :test :test-not :key` | [*Function*] |
| **assoc-if** *predicate a-list* | [*Function*] |
| **assoc-if-not** *predicate a-list* | [*Function*] |
| **assq** *object a-list* | [*Function*] |
| **atan** *number1* `&optional` *number2* | [*Function*] |
| **atanh** *number* | [*Function*] |
| **atom** *object* | [*Function*] |
| **bit** *bit-array* `&rest` *subscripts* | [*Function*] |
| **bit-and** *bit-array1 bit-array2* `&optional` *result-bit-array* | [*Function*] |
| **bit-andc1** *bit-array1 bit-array2* `&optional` *result-bit-array* | [*Function*] |
| **bit-andc2** *bit-array1 bit-array2* `&optional` *result-bit-array* | [*Function*] |
| **bit-eqv** *bit-array1 bit-array2* `&optional` *result-bit-array* | [*Function*] |
| **bit-ior** *bit-array1 bit-array2* `&optional` *result-bit-array* | [*Function*] |
| **bit-nand** *bit-array1 bit-array2* `&optional` *result-bit-array* | [*Function*] |
| **bit-nor** *bit-array1 bit-array2* `&optional` *result-bit-array* | [*Function*] |
| **bit-not** *bit-array* `&optional` *result-bit-array* | [*Function*] |
| **bit-orc1** *bit-array1 bit-array2* `&optional` *result-bit-array* | [*Function*] |
| **bit-orc2** *bit-array1 bit-array2* `&optional` *result-bit-array* | [*Function*] |
| **bit-vector-p** *object* | [*Function*] |
| **bit-xor** *bit-array1 bit-array2* `&optional` *result-bit-array* | [*Function*] |
| **block** *name* {*form*}* | [*Special Form*] |
| **boole** *op integer1 integer2* | [*Function*] |
| **boole-1** | [*Constant*] |

| | |
|---|---|
| **boole-2** | [*Constant*] |
| **boole-and** | [*Constant*] |
| **boole-andc1** | [*Constant*] |
| **boole-andc2** | [*Constant*] |
| **boole-c1** | [*Constant*] |
| **boole-c2** | [*Constant*] |
| **boole-clr** | [*Constant*] |
| **boole-eqv** | [*Constant*] |
| **boole-ior** | [*Constant*] |
| **boole-nand** | [*Constant*] |
| **boole-nor** | [*Constant*] |
| **boole-orc1** | [*Constant*] |
| **boole-orc2** | [*Constant*] |
| **boole-set** | [*Constant*] |
| **boole-xor** | [*Constant*] |
| **both-case-p** *char* | [*Function*] |
| **boundp** *symbol* | [*Function*] |
| **break** `&optional` *format-string* `&rest` *args* | [*Function*] |
| ***break-on-warnings*** | [*Variable*] |
| **butlast** *list* `&optional` *n* | [*Function*] |
| **byte** *size position* | [*Function*] |
| **byte-position** *bytespec* | [*Function*] |
| **byte-size** *bytespec* | [*Function*] |
| **caaaar** *list* | [*Function*] |
| **caaadr** *list* | [*Function*] |
| **caaar** *list* | [*Function*] |
| **caadar** *list* | [*Function*] |
| **caaddr** *list* | [*Function*] |
| **caadr** *list* | [*Function*] |
| **caar** *list* | [*Function*] |

| | |
|---|---|
| **cadaar** *list* | [*Function*] |
| **cadadr** *list* | [*Function*] |
| **cadar** *list* | [*Function*] |
| **caddar** *list* | [*Function*] |
| **cadddr** *list* | [*Function*] |
| **caddr** *list* | [*Function*] |
| **cadr** *list* | [*Function*] |
| **call-arguments-limit** | [*Constant*] |
| **car** *list* | [*Function*] |
| **case** *keyform* {({({*key*}*) \| *key*} {*form*}*)}* | [*Macro*] |
| **catch** *tag* {*form*}* | [*Special Form*] |
| **ccase** *keyplace* {({({*key*}*) \| *key*} {*form*}*)}* | [*Macro*] |
| **cdaaar** *list* | [*Function*] |
| **cdaadr** *list* | [*Function*] |
| **cdaar** *list* | [*Function*] |
| **cdadar** *list* | [*Function*] |
| **cdaddr** *list* | [*Function*] |
| **cdadr** *list* | [*Function*] |
| **cdar** *list* | [*Function*] |
| **cddaar** *list* | [*Function*] |
| **cddadr** *list* | [*Function*] |
| **cddar** *list* | [*Function*] |
| **cdddar** *list* | [*Function*] |
| **cddddr** *list* | [*Function*] |
| **cdddr** *list* | [*Function*] |
| **cddr** *list* | [*Function*] |
| **cdr** *list* | [*Function*] |
| **ceiling** *number* `&optional` *divisor* | [*Function*] |
| **cerror** *continue-format-string error-format-string* `&rest` *args* | [*Function*] |
| **char** *string index* | [*Function*] |

**char-bit** *char name* [*Function*]

**char-bits** *char* [*Function*]

**char-bits-limit** [*Constant*]

**char-code** *char* [*Function*]

**char-code-limit** [*Constant*]

**char-control-bit** [*Constant*]

**char-downcase** *char* [*Function*]

**char-equal** *character* &rest *more-characters* [*Function*]

**char-font** *char* [*Function*]

**char-font-limit** [*Constant*]

**char-greaterp** *character* &rest *more-characters* [*Function*]

**char-hyper-bit** [*Constant*]

**char-int** *char* [*Function*]

**char-lessp** *character* &rest *more-characters* [*Function*]

**char-meta-bit** [*Constant*]

**char-name** *char* [*Function*]

**char-not-equal** *character* &rest *more-characters* [*Function*]

**char-not-greaterp** *character* &rest *more-characters* [*Function*]

**char-not-lessp** *character* &rest *more-characters* [*Function*]

**char-super-bit** [*Constant*]

**char-upcase** *char* [*Function*]

**char/=** *character* &rest *more-characters* [*Function*]

**char<** *character* &rest *more-characters* [*Function*]

**char<=** *character* &rest *more-characters* [*Function*]

**char=** *character* &rest *more-characters* [*Function*]

**char>** *character* &rest *more-characters* [*Function*]

**char>=** *character* &rest *more-characters* [*Function*]

**character** *object* [*Function*]

**characterp** *object* [*Function*]

**check-type** *place typespec* &optional *string* [*Macro*]

| | |
|---|---|
| **cis** *radians* | [*Function*] |
| **clear-input** `&optional` *input-stream* | [*Function*] |
| **clear-output** `&optional` *output-stream* | [*Function*] |
| **close** *stream* `&key :abort` | [*Function*] |
| **clrhash** *hash-table* | [*Function*] |
| **code-char** *code* `&optional` (*bits* 0) (*font* 0) | [*Function*] |
| **coerce** *object result-type* | [*Function*] |
| **commonp** *object* | [*Function*] |
| **compile** *name* `&optional` *definition* | [*Function*] |
| **compile-file** *input-pathname* `&key :output-file :messages :warnings :fast-entry :tail-merge :notinline :target` | [*Function*] |
| **compiled-function-p** *object* | [*Function*] |
| **compiler-let** ({*var* \| (*var value*)}*) {*form*}* | [*Special Form*] |
| **complex** *realpart* `&optional` *imagpart* | [*Function*] |
| **complexp** *object* | [*Function*] |
| **concatenate** *result-type* `&rest` *sequences* | [*Function*] |
| **cond** {(*test* {*form*}*)}* | [*Macro*] |
| **conjugate** *number* | [*Function*] |
| **cons** *object1 object2* | [*Function*] |
| **consp** *object* | [*Function*] |
| **constantp** *object* | [*Function*] |
| **copy-alist** *list* | [*Function*] |
| **copy-list** *list* | [*Function*] |
| **copy-readtable** `&optional` *from-readtable to-readtable* | [*Function*] |
| **copy-seq** *sequence* | [*Function*] |
| **copy-symbol** *symbol* `&optional` *copy-props* | [*Function*] |
| **copy-tree** *object* | [*Function*] |

| | |
|---|---|
| **cos** *radians* | [*Function*] |
| **cosh** *number* | [*Function*] |
| **count** *item sequence* &key :from-end :test :test-not :start :end :key | [*Function*] |
| **count-if** *test sequence* &key :from-end :start :end :key | [*Function*] |
| **count-if-not** *test sequence* &key :from-end :start :end :key | [*Function*] |
| **ctypecase** *keyplace* {(*type* {*form*}*)}* | [*Macro*] |
| ***debug-io*** | [*Variable*] |
| **decache-eval** | [*Function*] |
| **decf** *place* [*delta*] | [*Macro*] |
| **declare** {*decl-spec*}* | [*Special Form*] |
| **decode-float** *float* | [*Function*] |
| **decode-universal-time** *universal-time* &optional *time-zone* | [*Function*] |
| ***default-pathname-defaults*** | [*Variable*] |
| **defconstant** *name initial-value* [*documentation*] | [*Macro*] |
| **define-function** *name function* | [*Function*] |
| **define-macro** *name function* | [*Function*] |
| **define-modify-macro** *name lambda-list function* [*documentation*] | [*Macro*] |
| **define-setf-method** *access-fn lambda-list* {*declaration* \| *documentation*}* {*form*}* | [*Macro*] |
| **defmacro** *name lambda-list* {*declaration* \| *documentation*}* {*form*}* | [*Macro*] |
| **defparameter** *name initial-value* [*documentation*] | [*Macro*] |
| **defsetf** *access-fn* {*update-fn* [*documentation*] \| *lambda-list* (*store-variable*) {*declaration* \| *documentation*}* {*form*}*} | [*Macro*] |
| **defstruct** *name-and-options* [*documentation*] {*slot-description*}* | [*Macro*] |
| **deftype** *name lambda-list* {*declaration* \| *documentation*}* {*form*}* | [*Macro*] |
| **defun** *name lambda-list* {*declaration* \| *documentation*}* {*form*}* | [*Macro*] |
| **defvar** *name* [*initial-value* [*documentation*]] | [*Macro*] |

**delete** *item sequence* &key :from-end :test :test-not :start :end :count :key [*Function*]

**delete-duplicates** *sequence* &key :from-end :test :test-not :start :end :key [*Function*]

**delete-file** *file* [*Function*]

**delete-if** *test sequence* &key :from-end :start :end :count :key [*Function*]

**delete-if-not** *test sequence* &key :from-end :start :end :count :key [*Function*]

**delete-package** *package* [*Function*]

**denominator** *rational* [*Function*]

**deposit-field** *newbyte bytespec integer* [*Function*]

**describe** *object* [*Function*]

**digit-char** *weight* &optional (*radix* 10) (*font* 0) [*Function*]

**digit-char-p** *char* &optional (*radix* 10) [*Function*]

**directory** *pathname* [*Function*]

**directory-namestring** *pathname* [*Function*]

**disassemble** *name-or-compiled-function* [*Function*]

**do** ({*var* | (*var* [*init* [*step*]])}*) (*end-test* {*form*}*) {*declaration*}* {*tag* | *statement*}* [*Macro*]

**do*** ({*var* | (*var* [*init* [*step*]])}*) (*end-test* {*form*}*) {*declaration*}* {*tag* | *statement*}* [*Macro*]

**do-all-symbols** (*var* [*result-form*]) {*declaration*}* {*tag* | *statement*}* [*Macro*]

**do-external-symbols** (*var* [*package* [*result-form*]]) {*declaration*}* {*tag* | *statement*}* [*Macro*]

**do-symbols** (*var* [*package* [*result-form*]]) {*declaration*}* {*tag* | *statement*}* [*Macro*]

**documentation** *symbol doc-type* [*Function*]

**dolist** (*var listform* [*result*]) {*declaration*}* {*tag* | *statement*}* [*Macro*]

**dotimes** (*var countform* [*result*]) {*declaration*}* {*tag* | *statement*}* [*Macro*]

**double-float-epsilon** [*Constant*]

**double-float-negative-epsilon** [*Constant*]

**dpb** *newbyte bytespec integer* [*Function*]

| | |
|---|---|
| **dribble** `&optional` *pathname* | [*Function*] |
| **ecase** *keyform* {({({*key*}*) \| *key*} {*form*}*)}* | [*Macro*] |
| **ed** `&optional` *x* `&key :windows &allow-other-keys` | [*Function*] |
| **eighth** *list* | [*Function*] |
| **elt** *sequence index* | [*Function*] |
| **encode-universal-time** *second minute hour date month year* `&optional` *time-zone* | [*Function*] |
| **endp** *list* | [*Function*] |
| **enough-namestring** *pathname* `&optional` *defaults* | [*Function*] |
| **eq** *x y* | [*Function*] |
| **eql** *x y* | [*Function*] |
| **equal** *x y* | [*Function*] |
| **equalp** *x y* | [*Function*] |
| **error** *format-string* `&rest` *args* | [*Function*] |
| ***error-output*** | [*Variable*] |
| **etypecase** *keyform* {(*type* {*form*}*)}* | [*Macro*] |
| **eval** *form* | [*Function*] |
| **eval-when** ({*situation*}*) {*form*}* | [*Special Form*] |
| ***evalhook*** | [*Variable*] |
| **evalhook** *form evalhookfn applyhookfn* `&optional` *env* | [*Function*] |
| **evenp** *integer* | [*Function*] |
| **every** *predicate sequence* `&rest` *more-sequences* | [*Function*] |
| **exp** *number* | [*Function*] |
| **export** *symbols* `&optional` *package* | [*Function*] |
| **expt** *base-number power-number* | [*Function*] |
| **fboundp** *symbol* | [*Function*] |
| **fceiling** *number* `&optional` *divisor* | [*Function*] |
| ***features*** | [*Variable*] |
| **ffloor** *number* `&optional` *divisor* | [*Function*] |
| **fifth** *list* | [*Function*] |

| | |
|---|---|
| **file-author** *file* | [*Function*] |
| **file-length** *file-stream* | [*Function*] |
| **file-namestring** *pathname* | [*Function*] |
| **file-position** *file-stream* `&optional` *position* | [*Function*] |
| **file-write-date** *file* | [*Function*] |
| **fill** *sequence item* `&key :start :end` | [*Function*] |
| **fill-pointer** *vector* | [*Function*] |
| **find** *item sequence* `&key :from-end :test :test-not :start :end :key` | [*Function*] |
| **find-all-symbols** *string-or-symbol* | [*Function*] |
| **find-if** *test sequence* `&key :from-end :start :end :key` | [*Function*] |
| **find-if-not** *test sequence* `&key :from-end :start :end :key` | [*Function*] |
| **find-package** *name* | [*Function*] |
| **find-symbol** *string* `&optional` *package* | [*Function*] |
| **finish-output** `&optional` *output-stream* | [*Function*] |
| **first** *list* | [*Function*] |
| **fixnump** *object* | [*Function*] |
| **flet** ({(*name lambda-list* {*declaration* \| *documentation*}* {*form*}*)}*) {*form*}* | [*Special Form*] |
| **float** *number* `&optional` *float* | [*Function*] |
| **float-digits** *float* | [*Function*] |
| **float-precision** *float* | [*Function*] |
| **float-radix** *float* | [*Function*] |
| **float-sign** *float1* `&optional` *float2* | [*Function*] |
| **floatp** *object* | [*Function*] |
| **floor** *number* `&optional` *divisor* | [*Function*] |
| **fmakunbound** *symbol* | [*Function*] |
| **force-output** `&optional` *output-stream* | [*Function*] |
| **format** *destination format-control-string* `&rest` *arguments* | [*Function*] |
| **fourth** *list* | [*Function*] |
| **fresh-line** `&optional` *output-stream* | [*Function*] |

| | |
|---|---|
| **fround** *number* `&optional` *divisor* | [*Function*] |
| **ftruncate** *number* `&optional` *divisor* | [*Function*] |
| **funcall** *function* `&rest` *args* | [*Function*] |
| **function** *function* | [*Special Form*] |
| **functionp** *object* | [*Function*] |
| **gcd** `&rest` *integers* | [*Function*] |
| **gensym** `&optional` *x* | [*Function*] |
| **gentemp** `&optional` *prefix package* | [*Function*] |
| **get** *symbol indicator* `&optional` *default* | [*Function*] |
| **get-decoded-time** | [*Function*] |
| **get-dispatch-macro-character** *disp-char sub-char* `&optional` *readtable* | [*Function*] |
| **get-internal-real-time** | [*Function*] |
| **get-internal-run-time** | [*Function*] |
| **get-macro-character** *char* `&optional` *readtable* | [*Function*] |
| **get-output-stream-string** *string-output-stream* | [*Function*] |
| **get-properties** *place indicator-list* | [*Function*] |
| **get-setf-method** *form* | [*Function*] |
| **get-setf-method-multiple-value** *form* | [*Function*] |
| **get-universal-time** | [*Function*] |
| **getf** *place indicator* `&optional` *default* | [*Function*] |
| **gethash** *key hash-table* `&optional` *default* | [*Function*] |
| **go** *tag* | [*Special Form*] |
| **graphic-char-p** *char* | [*Function*] |
| **grindef** `&rest` *function-name* | [*Macro*] |
| **hash-table-count** *hash-table* | [*Function*] |
| **hash-table-p** *object* | [*Function*] |
| **host-namestring** *pathname* | [*Function*] |
| **identity** *object* | [*Function*] |
| **if** *test then* [*else*] | [*Special Form*] |

| | |
|---|---|
| ***ignore-extra-right-parens*** | [*Variable*] |
| **imagpart** *number* | [*Function*] |
| **import** *symbols* `&optional` *package* | [*Function*] |
| **in-package** *package-name* `&key :nicknames :use` | [*Function*] |
| **incf** *place* [*delta*] | [*Macro*] |
| **input-stream-p** *stream* | [*Function*] |
| **inspect** *object* | [*Function*] |
| **int-char** *integer* | [*Function*] |
| **integer-decode-float** *float* | [*Function*] |
| **integer-length** *integer* | [*Function*] |
| **integerp** *object* | [*Function*] |
| **intern** *string* `&optional` *package* | [*Function*] |
| **internal-time-units-per-second** | [*Constant*] |
| **intersection** *list1 list2* `&key :test :test-not :key` | [*Function*] |
| **isqrt** *integer* | [*Function*] |
| **keywordp** *object* | [*Function*] |
| **labels** ({(*name lambda-list* {*declaration* \| *documentation*}* {*form*}*)}*) {*form*}* | [*Special Form*] |
| **lambda-list-keywords** | [*Constant*] |
| **lambda-parameters-limit** | [*Constant*] |
| **last** *list* | [*Function*] |
| **lcm** *integer* `&rest` *more-integers* | [*Function*] |
| **ldb** *bytespec integer* | [*Function*] |
| **ldb-test** *bytespec integer* | [*Function*] |
| **ldiff** *list sublist* | [*Function*] |
| **least-negative-double-float** | [*Constant*] |
| **least-negative-long-float** | [*Constant*] |
| **least-negative-short-float** | [*Constant*] |
| **least-negative-single-float** | [*Constant*] |
| **least-positive-double-float** | [*Constant*] |

| | |
|---|---|
| **least-positive-long-float** | [*Constant*] |
| **least-positive-short-float** | [*Constant*] |
| **least-positive-single-float** | [*Constant*] |
| **length** *sequence* | [*Function*] |
| **let** ({*var* \| (*var value*)}*) {*declaration*}* {*form*}* | [*Special Form*] |
| **let*** ({*var* \| (*var value*)}*) {*declaration*}* {*form*}* | [*Special Form*] |
| **lisp-implementation-type** | [*Function*] |
| **lisp-implementation-version** | [*Function*] |
| **list** `&rest` *objects* | [*Function*] |
| **list*** *object* `&rest` *more-objects* | [*Function*] |
| **list-all-packages** | [*Function*] |
| **list-length** *list* | [*Function*] |
| **list-nreverse** *list* | [*Function*] |
| **list-reverse** *list* | [*Function*] |
| **listen** `&optional` *input-stream* | [*Function*] |
| **listp** *object* | [*Function*] |
| **load** *filename* `&key :verbose :print :if-does-not-exist` | [*Function*] |
| ***load-verbose*** | [*Variable*] |
| **locally** {*declaration*}* {*form*}* | [*Macro*] |
| **log** *number* `&optional` *base* | [*Function*] |
| **logand** `&rest` *integers* | [*Function*] |
| **logandc1** *integer1 integer2* | [*Function*] |
| **logandc2** *integer1 integer2* | [*Function*] |
| **logbitp** *index integer* | [*Function*] |
| **logcount** *integer* | [*Function*] |
| **logeqv** `&rest` *integers* | [*Function*] |
| **logior** `&rest` *integers* | [*Function*] |
| **lognand** *integer1 integer2* | [*Function*] |
| **lognor** *integer1 integer2* | [*Function*] |
| **lognot** *integer* | [*Function*] |

| | |
|---|---|
| **logorc1** *integer1 integer2* | [*Function*] |
| **logorc2** *integer1 integer2* | [*Function*] |
| **logtest** *integer1 integer2* | [*Function*] |
| **logxor** `&rest` *integers* | [*Function*] |
| **long-float-epsilon** | [*Constant*] |
| **long-float-negative-epsilon** | [*Constant*] |
| **long-site-name** | [*Function*] |
| **loop** {*form*}* | [*Macro*] |
| **lower-case-p** *char* | [*Function*] |
| **machine-instance** | [*Function*] |
| **machine-type** | [*Function*] |
| **machine-version** | [*Function*] |
| **macro-function** *symbol* | [*Function*] |
| **macroexpand** *form* `&optional` *env* | [*Function*] |
| **macroexpand-1** *form* `&optional` *env* | [*Function*] |
| ***macroexpand-hook*** | [*Variable*] |
| **macrolet** ({(*name lambda-list* {*declaration* \| *documentation*}* {*form*}*)}*) {*form*}* | [*Special Form*] |
| **make-array** *dimensions* `&key :element-type :initial-element :initial-contents :adjustable :fill-pointer :displaced-to :displaced-index-offset` | [*Function*] |
| **make-broadcast-stream** `&rest` *streams* | [*Function*] |
| **make-char** *char* `&optional` (*bits* 0) (*font* 0) | [*Function*] |
| **make-concatenated-stream** `&rest` *streams* | [*Function*] |
| **make-dispatch-macro-character** *char* `&optional` *non-terminating-p readtable* | [*Function*] |
| **make-echo-stream** *input-stream output-stream* | [*Function*] |
| **make-hash-table** `&key :test :size :rehash-size :rehash-threshold` | [*Function*] |
| **make-list** *size* `&key :initial-element` | [*Function*] |

| | |
|---|---|
| **make-package** *package-name* `&key :nicknames :use` | [*Function*] |
| **make-pathname** `&key :host :device :directory :name :type :version :defaults` | [*Function*] |
| **make-random-state** `&optional` *state* | [*Function*] |
| **make-sequence** *type size* `&key :initial-element` | [*Function*] |
| **make-string** *size* `&key :initial-element` | [*Function*] |
| **make-string-input-stream** *string* `&optional` *start end* | [*Function*] |
| **make-string-output-stream** `&optional` *string* | [*Function*] |
| **make-symbol** *print-name* | [*Function*] |
| **make-synonym-stream** *symbol* | [*Function*] |
| **make-two-way-stream** *input-stream output-stream* | [*Function*] |
| **makunbound** *symbol* | [*Function*] |
| **map** *result-type function sequence* `&rest` *more-sequences* | [*Function*] |
| **mapc** *function list* `&rest` *more-lists* | [*Function*] |
| **mapcan** *function list* `&rest` *more-lists* | [*Function*] |
| **mapcar** *function list* `&rest` *more-lists* | [*Function*] |
| **mapcon** *function list* `&rest` *more-lists* | [*Function*] |
| **maphash** *function hash-table* | [*Function*] |
| **mapl** *function list* `&rest` *more-lists* | [*Function*] |
| **maplist** *function list* `&rest` *more-lists* | [*Function*] |
| **mask-field** *bytespec integer* | [*Function*] |
| **max** *number* `&rest` *more-numbers* | [*Function*] |
| **member** *item list* `&key :test :test-not :key` | [*Function*] |
| **member-if** *predicate list* `&key :key` | [*Function*] |
| **member-if-not** *predicate list* `&key :key` | [*Function*] |
| **memq** *object list* | [*Function*] |
| **merge** *result-type sequence1 sequence2 predicate* `&key :key` | [*Function*] |
| **merge-pathnames** *pathname* `&optional` *defaults default-version* | [*Function*] |
| **min** *number* `&rest` *more-numbers* | [*Function*] |
| **minusp** *number* | [*Function*] |

| | |
|---|---|
| **mismatch** *sequence1 sequence2* `&key :from-end :test :test-not :key :start1 :start2 :end1 :end2` | [*Function*] |
| **mod** *number divisor* | [*Function*] |
| ***modules*** | [*Variable*] |
| **most-negative-double-float** | [*Constant*] |
| **most-negative-fixnum** | [*Constant*] |
| **most-negative-long-float** | [*Constant*] |
| **most-negative-short-float** | [*Constant*] |
| **most-negative-single-float** | [*Constant*] |
| **most-positive-double-float** | [*Constant*] |
| **most-positive-fixnum** | [*Constant*] |
| **most-positive-long-float** | [*Constant*] |
| **most-positive-short-float** | [*Constant*] |
| **most-positive-single-float** | [*Constant*] |
| **multiple-value-bind** ({*var*}*) *values-form* {*declaration*}* {*form*}* | [*Macro*] |
| **multiple-value-call** *function* {*form*}* | [*Special Form*] |
| **multiple-value-list** *form* | [*Macro*] |
| **multiple-value-prog1** *form* {*form*}* | [*Special Form*] |
| **multiple-value-setq** *vars form* | [*Macro*] |
| **multiple-values-limit** | [*Constant*] |
| **name-char** *name* | [*Function*] |
| **namestring** *pathname* | [*Function*] |
| **nbutlast** *list* `&optional` *n* | [*Function*] |
| **nconc** `&rest` *lists* | [*Function*] |
| **nil** | [*Constant*] |
| **nintersection** *list1 list2* `&key :test :test-not :key` | [*Function*] |
| **ninth** *list* | [*Function*] |
| **not** *x* | [*Function*] |
| **notany** *predicate sequence* `&rest` *more-sequences* | [*Function*] |

**notevery** *predicate sequence* `&rest` *more-sequences* [*Function*]

**nreconc** *list1 list2* [*Function*]

**nreverse** *sequence* [*Function*]

**nset-difference** *list1 list2* `&key :test :test-not :key` [*Function*]

**nset-exclusive-or** *list1 list2* `&key :test :test-not :key` [*Function*]

**nstring-capitalize** *string* `&key :start :end` [*Function*]

**nstring-downcase** *string* `&key :start :end` [*Function*]

**nstring-upcase** *string* `&key :start :end` [*Function*]

**nsublis** *a-list tree* `&key :test :test-not :key` [*Function*]

**nsubst** *new old tree* `&key :test :test-not :key` [*Function*]

**nsubst-if** *new test tree* `&key :key` [*Function*]

**nsubst-if-not** *new test tree* `&key :key` [*Function*]

**nsubstitute** *newitem olditem sequence* `&key :from-end :test :test-not :start :end :count :key` [*Function*]

**nsubstitute-if** *newitem test sequence* `&key :from-end :start :end :count :key` [*Function*]

**nsubstitute-if-not** *newitem test sequence* `&key :from-end :start :end :count :key` [*Function*]

**nth** *n list* [*Function*]

**nthcdr** *n list* [*Function*]

**null** *object* [*Function*]

**numberp** *object* [*Function*]

**numerator** *rational* [*Function*]

**nunion** *list1 list2* `&key :test :test-not :key` [*Function*]

**oddp** *integer* [*Function*]

**open** *filename* `&key :direction :element-type :if-exists :if-does-not-exist` [*Function*]

**or** {*form*}* [*Macro*]

**output-stream-p** *stream* [*Function*]

***package*** [*Variable*]

**package-name** *package* [*Function*]

**package-nicknames** *package* [*Function*]

**package-shadowing-symbols** *package* [*Function*]

**package-use-list** *package* [*Function*]

**package-used-by-list** *package* [*Function*]

**packagep** *object* [*Function*]

**pairlis** *keys data* `&optional` *a-list* [*Function*]

**parse-integer** *string* `&key :start :end :radix :junk-allowed` [*Function*]

**parse-namestring** *thing* `&optional` *host defaults* `&key :start :end :junk-allowed` [*Function*]

**pathname** *pathname* [*Function*]

**pathname-device** *pathname* [*Function*]

**pathname-directory** *pathname* [*Function*]

**pathname-host** *pathname* [*Function*]

**pathname-name** *pathname* [*Function*]

**pathname-type** *pathname* [*Function*]

**pathname-version** *pathname* [*Function*]

**pathnamep** *object* [*Function*]

**peek-char** `&optional` *peek-type input-stream eof-error-p eof-value recursive-p* [*Function*]

**phase** *number* [*Function*]

**pi** [*Constant*]

**plusp** *number* [*Function*]

**pop** *place* [*Macro*]

**position** *item sequence* `&key :from-end :test :test-not :start :end :key` [*Function*]

**position-if** *test sequence* `&key :from-end :start :end :key` [*Function*]

**position-if-not** *test sequence* `&key :from-end :start :end :key` [*Function*]

***pp-line-length*** [*Variable*]

**pprint** *object* `&optional` *output-stream* [*Function*]

**prin1** *object* `&optional` *output-stream* [*Function*]

| | |
|---|---|
| prin1-to-string object | [Function] |
| princ object &optional output-stream | [Function] |
| princ-to-string object | [Function] |
| print object &optional output-stream | [Function] |
| *print-array* | [Variable] |
| *print-base* | [Variable] |
| *print-case* | [Variable] |
| *print-circle* | [Variable] |
| *print-escape* | [Variable] |
| *print-gensym* | [Variable] |
| *print-length* | [Variable] |
| *print-level* | [Variable] |
| *print-pretty* | [Variable] |
| *print-radix* | [Variable] |
| *print-structure* | [Variable] |
| probe-file file | [Function] |
| proclaim decl-spec | [Function] |
| prog ({var \| (var [init]))}*) {declaration}* {tag \| statement}* | [Macro] |
| prog* ({var \| (var [init]))}*) {declaration}* {tag \| statement}* | [Macro] |
| prog1 first {form}* | [Macro] |
| prog2 first second {form}* | [Macro] |
| progn {form}* | [Special Form] |
| progv symbols values {form}* | [Special Form] |
| *prompt* | [Variable] |
| provide module-name | [Function] |
| psetf {place newvalue}* | [Macro] |
| psetq {var form}* | [Macro] |
| push item place | [Macro] |
| pushnew item place &key :test :test-not :key | [Macro] |
| *query-io* | [Variable] |

| | |
|---|---|
| **quit** &optional *status* | [*Function*] |
| **quote** *object* | [*Special Form*] |
| **random** *number* &optional *state* | [*Function*] |
| *random-state* | [*Variable*] |
| **random-state-p** *object* | [*Function*] |
| **rassoc** *item a-list* &key :test :test-not :key | [*Function*] |
| **rassoc-if** *predicate a-list* | [*Function*] |
| **rassoc-if-not** *predicate a-list* | [*Function*] |
| **rational** *number* | [*Function*] |
| **rationalize** *number* | [*Function*] |
| **rationalp** *object* | [*Function*] |
| **read** &optional *input-stream eof-error-p eof-value recursive-p* | [*Function*] |
| *read-base* | [*Variable*] |
| **read-byte** *binary-input-stream* &optional *eof-error-p eof-value* | [*Function*] |
| **read-char** &optional *input-stream eof-error-p eof-value recursive-p* | [*Function*] |
| **read-char-no-hang** &optional *input-stream eof-error-p eof-value recursive-p* | [*Function*] |
| *read-default-float-format* | [*Variable*] |
| **read-delimited-list** *char* &optional *input-stream recursive-p* | [*Function*] |
| **read-from-string** *string* &optional *eof-error-p eof-value* &key :start :end :preserve-whitespace | [*Function*] |
| **read-line** &optional *input-stream eof-error-p eof-value recursive-p* | [*Function*] |
| **read-preserving-whitespace** &optional *input-stream eof-error-p eof-value recursive-p* | [*Function*] |
| *read-suppress* | [*Variable*] |
| *readtable* | [*Variable*] |
| **readtablep** *object* | [*Function*] |
| **realpart** *number* | [*Function*] |
| *redefinition-action* | [*Variable*] |
| **reduce** *function sequence* &key :from-end :start :end :initial-value | [*Function*] |

**rem** *number divisor* [*Function*]

**remf** *place indicator* [*Macro*]

**remhash** *key hash-table* [*Function*]

**remove** *item sequence* &key :from-end :test :test-not :start :end :count :key [*Function*]

**remove-duplicates** *sequence* &key :from-end :test :test-not :start :end :key [*Function*]

**remove-if** *test sequence* &key :from-end :start :end :count :key [*Function*]

**remove-if-not** *test sequence* &key :from-end :start :end :count :key [*Function*]

**remprop** *symbol indicator* [*Function*]

**rename-file** *file new-name* [*Function*]

**rename-package** *package new-name* &optional *new-nicknames* [*Function*]

**replace** *sequence1 sequence2* &key :start1 :end1 :start2 :end2 [*Function*]

**require** *module-name* &optional *pathname* [*Function*]

**rest** *list* [*Function*]

**return** [*result*] [*Macro*]

**return-from** *name* [*result*] [*Special Form*]

**revappend** *list1 list2* [*Function*]

**reverse** *sequence* [*Function*]

**room** &optional *x* [*Function*]

**rotatef** {*place*}* [*Macro*]

**round** *number* &optional *divisor* [*Function*]

**rplaca** *cons object* [*Function*]

**rplacd** *cons object* [*Function*]

**sbit** *simple-bit-array* &rest *subscripts* [*Function*]

**scale-float** *float integer* [*Function*]

**schar** *simple-string index* [*Function*]

**search** *sequence1 sequence2* &key :from-end :test :test-not :key :start1 :start2 :end1 :end2 [*Function*]

| | |
|---|---|
| **second** *list* | [*Function*] |
| **set** *symbol value* | [*Function*] |
| **set-char-bit** *char name logical-value* | [*Function*] |
| **set-difference** *list1 list2* `&key :test :test-not :key` | [*Function*] |
| **set-dispatch-macro-character** *disp-char sub-char function* `&optional` *readtable* | [*Function*] |
| **set-exclusive-or** *list1 list2* `&key :test :test-not :key` | [*Function*] |
| **set-macro-character** *char function* `&optional` *non-terminating-p readtable* | [*Function*] |
| **set-syntax-from-char** *to-char from-char* `&optional` *to-readtable from-readtable* | [*Function*] |
| **setf** {*place newvalue*}* | [*Macro*] |
| **setq** {*var form*}* | [*Special Form*] |
| **seventh** *list* | [*Function*] |
| **shadow** *symbols* `&optional` *package* | [*Function*] |
| **shadowing-import** *symbols* `&optional` *package* | [*Function*] |
| **shiftf** {*place*}$^+$ *newvalue* | [*Macro*] |
| **short-float-epsilon** | [*Constant*] |
| **short-float-negative-epsilon** | [*Constant*] |
| **short-site-name** | [*Function*] |
| **signum** *number* | [*Function*] |
| **simple-bit-vector-p** *object* | [*Function*] |
| **simple-string-p** *object* | [*Function*] |
| **simple-vector-p** *object* | [*Function*] |
| **sin** *radians* | [*Function*] |
| **single-float-epsilon** | [*Constant*] |
| **single-float-negative-epsilon** | [*Constant*] |
| **sinh** *number* | [*Function*] |
| **sixth** *list* | [*Function*] |
| **sleep** *seconds* | [*Function*] |
| **software-type** | [*Function*] |

| | |
|---|---|
| **software-version** | [*Function*] |
| **some** *predicate sequence* &rest *more-sequences* | [*Function*] |
| **sort** *sequence predicate* &key :key | [*Function*] |
| **source-code** *function* | [*Function*] |
| **special-form-p** *symbol* | [*Function*] |
| **sqrt** *number* | [*Function*] |
| **stable-sort** *sequence predicate* &key :key | [*Function*] |
| **standard-char-p** *char* | [*Function*] |
| *standard-input* | [*Variable*] |
| *standard-output* | [*Variable*] |
| **step** *form* $\mid$ $\{$*function-name*$\}^+$ | [*Macro*] |
| **stream-element-type** *stream* | [*Function*] |
| **streamp** *object* | [*Function*] |
| **string** *x* | [*Function*] |
| **string-capitalize** *string* &key :start :end | [*Function*] |
| **string-char-p** *char* | [*Function*] |
| **string-downcase** *string* &key :start :end | [*Function*] |
| **string-equal** *string1 string2* &key :start1 :end1 :start2 :end2 | [*Function*] |
| **string-greaterp** *string1 string2* &key :start1 :end1 :start2 :end2 | [*Function*] |
| **string-left-trim** *character-bag string* | [*Function*] |
| **string-lessp** *string1 string2* &key :start1 :end1 :start2 :end2 | [*Function*] |
| **string-not-equal** *string1 string2* &key :start1 :end1 :start2 :end2 | [*Function*] |
| **string-not-greaterp** *string1 string2* &key :start1 :end1 :start2 :end2 | [*Function*] |
| **string-not-lessp** *string1 string2* &key :start1 :end1 :start2 :end2 | [*Function*] |
| **string-right-trim** *character-bag string* | [*Function*] |
| **string-trim** *character-bag string* | [*Function*] |
| **string-upcase** *string* &key :start :end | [*Function*] |
| **string/=** *string1 string2* &key :start1 :end1 :start2 :end2 | [*Function*] |

| | |
|---|---|
| **string<** *string1 string2* &key :start1 :end1 :start2 :end2 | [*Function*] |
| **string<=** *string1 string2* &key :start1 :end1 :start2 :end2 | [*Function*] |
| **string=** *string1 string2* &key :start1 :end1 :start2 :end2 | [*Function*] |
| **string>** *string1 string2* &key :start1 :end1 :start2 :end2 | [*Function*] |
| **string>=** *string1 string2* &key :start1 :end1 :start2 :end2 | [*Function*] |
| **stringp** *object* | [*Function*] |
| **sublis** *a-list tree* &key :test :test-not :key | [*Function*] |
| **subseq** *sequence start* &optional *end* | [*Function*] |
| **subsetp** *list1 list2* &key :test :test-not :key | [*Function*] |
| **subst** *new old tree* &key :test :test-not :key | [*Function*] |
| **subst-if** *new test tree* &key :key | [*Function*] |
| **subst-if-not** *new test tree* &key :key | [*Function*] |
| **substitute** *newitem olditem sequence* &key :from-end :test :test-not :start :end :count :key | [*Function*] |
| **substitute-if** *newitem test sequence* &key :from-end :start :end :count :key | [*Function*] |
| **substitute-if-not** *newitem test sequence* &key :from-end :start :end :count :key | [*Function*] |
| **subtypep** *type1 type2* | [*Function*] |
| **svref** *simple-vector index* | [*Function*] |
| **sxhash** *object* | [*Function*] |
| **symbol-function** *symbol* | [*Function*] |
| **symbol-name** *symbol* | [*Function*] |
| **symbol-package** *symbol* | [*Function*] |
| **symbol-plist** *symbol* | [*Function*] |
| **symbol-value** *symbol* | [*Function*] |
| **symbolp** *object* | [*Function*] |
| **t** | [*Constant*] |
| **tagbody** {*tag* \| *statement*}* | [*Special Form*] |

| | |
|---|---|
| **tailp** *sublist list* | [*Function*] |
| **tan** *radians* | [*Function*] |
| **tanh** *number* | [*Function*] |
| **tenth** *list* | [*Function*] |
| ***terminal-io*** | [*Variable*] |
| **terpri** `&optional` *output-stream* | [*Function*] |
| **the** *value-type form* | [*Special Form*] |
| **third** *list* | [*Function*] |
| **throw** *tag result* | [*Special Form*] |
| **time** *form* | [*Macro*] |
| **trace** {*trace-spec*}* | [*Macro*] |
| ***trace-output*** | [*Variable*] |
| **tree-equal** *object1 object2* `&key :test :test-not` | [*Function*] |
| **truename** *pathname* | [*Function*] |
| **truncate** *number* `&optional` *divisor* | [*Function*] |
| **type-of** *object* | [*Function*] |
| **typecase** *keyform* {(*type* {*form*}*)}* | [*Macro*] |
| **typep** *object type-specifier* | [*Function*] |
| **unexport** *symbols* `&optional` *package* | [*Function*] |
| **unintern** *symbol* `&optional` *package* | [*Function*] |
| **union** *list1 list2* `&key :test :test-not :key` | [*Function*] |
| **unless** *test* {*form*}* | [*Macro*] |
| **unread-char** *character* `&optional` *input-stream* | [*Function*] |
| **untrace** {*function-name*}* | [*Macro*] |
| **unuse-package** *packages-to-unuse* `&optional` *package* | [*Function*] |
| **unwind-protect** *protected-form* {*cleanup-form*}* | [*Special Form*] |
| **upper-case-p** *char* | [*Function*] |
| **use-package** *packages-to-use* `&optional` *package* | [*Function*] |
| **user-homedir-pathname** `&optional` *host* | [*Function*] |
| **values** `&rest` *args* | [*Function*] |

| | |
|---|---|
| **values-list** *list* | [*Function*] |
| **vector** `&rest` *objects* | [*Function*] |
| **vector-pop** *vector* | [*Function*] |
| **vector-push** *new-element vector* | [*Function*] |
| **vector-push-extend** *new-element vector* `&optional` *extension* | [*Function*] |
| **vectorp** *object* | [*Function*] |
| **warn** *format-string* `&rest` *args* | [*Function*] |
| **when** *test* {*form*}* | [*Macro*] |
| **with-input-from-string** (*var string* {*keyword value*}*) {*declaration*}* {*form*}* | [*Macro*] |
| **with-open-file** (*stream filename* {*options*}*) {*declaration*}* {*form*}* | [*Macro*] |
| **with-open-stream** (*var stream*) {*declaration*}* {*form*}* | [*Macro*] |
| **with-output-to-string** (*var* [*string*]) {*declaration*}* {*form*}* | [*Macro*] |
| **write** *object* `&key :stream :escape :radix :base :circle :pretty :level :length :case :gensym :array :structure` | [*Function*] |
| **write-byte** *integer binary-output-stream* | [*Function*] |
| **write-char** *character* `&optional` *output-stream* | [*Function*] |
| **write-line** *string* `&optional` *output-stream* `&key :start :end` | [*Function*] |
| **write-string** *string* `&optional` *output-stream* `&key :start :end` | [*Function*] |
| **write-to-string** *object* `&key :escape :radix :base :circle :pretty :level :length :case :gensym :array :structure` | [*Function*] |
| **y-or-n-p** `&optional` *format-control-string* `&rest` *arguments* | [*Function*] |
| **yes-or-no-p** `&optional` *format-control-string* `&rest` *arguments* | [*Function*] |
| **zerop** *number* | [*Function*] |

# Appendix B. Extensions to Common Lisp

This appendix is a listing of the extensions to Common Lisp described in this guide. Note that these extensions are not part of the Common Lisp specification. They are listed by chapter.

## Starting Up

| | |
|---|---|
| **abort** `&optional` *status* | [*Function*] |
| **disksave** *target-file* `&key :restart-function :full-gc :gc :reserved-free-segments :dynamic-free-segments :verbose` | [*Function*] |
| **quit** `&optional` *status* | [*Function*] |

## Debugging Lisp Programs

| | |
|---|---|
| ***debug-print-level*** | [*Variable*] |
| ***debug-print-length*** | [*Variable*] |

## Tracing Functions

| | |
|---|---|
| ***max-trace-indentation*** | [*Variable*] |
| ***trace-arglist*** | [*Variable*] |
| ***trace-bar-p*** | [*Variable*] |
| ***trace-columns-per-level*** | [*Variable*] |
| ***trace-level*** | [*Variable*] |
| ***trace-new-definitions*** | [*Variable*] |
| ***trace-values*** | [*Variable*] |
| ***traced-function-list*** | [*Variable*] |

## Stepping Through an Evaluation

| | |
|---|---|
| ***max-step-indentation*** | [*Variable*] |
| ***step-columns-per-level*** | [*Variable*] |
| ***step-level*** | [*Variable*] |

## The Foreign Function Interface

| | |
|---|---|
| **define-c-callable** *name arglist {form}** | [*Macro*] |
| **define-c-function** *function-name arglist* `&key :result-type` | [*Macro*] |
| **define-foreign-symbol** *symbol-name* | [*Macro*] |
| **define-fortran-function** *function-name arglist* `&key :result-type` | [*Macro*] |
| **extract-stream-handles** *common-lisp-stream* | [*Function*] |
| **foreign-address-of** *function-name* | [*Function*] |
| ***foreign-temporary-directory*** | [*Variable*] |
| **load-foreign-files** *files* `&optional` *libraries* | [*Function*] |
| **load-foreign-libraries** *symbols* `&optional` *libraries* | [*Function*] |
| **make-lisp-stream** `&key :input-handle :output-handle :element-type :stream-type :name` | [*Function*] |
| **register-lisp-function** *function-name* | [*Function*] |
| **syscall** *system-call-number* `&rest` *arguments* | [*Function*] |

## Running UNIX Programs from Lisp

**run-unix-program** *name* `&key` `:input :output` [*Function*]
`:error-output :wait`
`:arguments`
`:if-input-does-not-exist`
`:if-output-exists`
`:if-error-output-exists`

## Compiling Lisp Programs

**clear-undef** [*Function*]

**compiler-options** `&key` `:messages :warnings` [*Function*]
`:fast-entry :tail-merge`
`:notinline :target`

## Storage Management in Lisp

**change-memory-management** `&key` `:growth-limit` [*Function*]
`:growth-rate`
`:expand :expand-p`
`:reclamation-ratio`
`:expand-stack`
`:help`

**gc** [*Function*]

**gc-off** `&optional` ***no-reconsideration*** [*Function*]

**gc-on** [*Function*]

***gc-silence*** [*Variable*]

**get-stack-remaining** [*Function*]

## The Flavor System

| | |
|---|---|
| *all-flavor-names* | [*Variable*] |
| **cleanup-all-flavors** | [*Function*] |
| **compile-flavor-methods** {*flavor*}* | [*Macro*] |
| **continue-whopper** {*arg*}* | [*Macro*] |
| **continue-whopper-all** | [*Macro*] |
| **defflavor** *flavor-name* ({*var* \| (*var default-form*)}*) ({*flavor*}*) {*option*}* | [*Macro*] |
| **defmethod** (*flavor* [*method-type*] *message*) *lambda-list* {*declaration* \| *documentation*}* {*form*}* | [*Macro*] |
| **defwhopper** (*flavor message*) *lambda-list* {*declaration* \| *documentation*}* {*form*}* | [*Macro*] |
| **defwrapper** (*flavor message*) (*lambda-list . body-var*) {*declaration* \| *documentation*}* {*form*}* | [*Macro*] |
| **flavor-allowed-init-keywords** *flavor* | [*Function*] |
| **flavor-allows-init-keyword-p** *flavor keyword* | [*Function*] |
| **instancep** *object* | [*Function*] |
| **lexpr-continue-whopper** {*arg*}* *arg-list* | [*Macro*] |
| **make-instance** *flavor* {*keyword value*}* | [*Function*] |
| **recompile-flavor** *flavor* `&optional` *messages do-dependents* | [*Function*] |
| **self** | [*Variable*] |
| **send** *instance message* `&rest` *args* | [*Function*] |
| **set-in-instance** *instance symbol value* | [*Function*] |
| **symeval-in-instance** *instance symbol* `&optional` *no-error-p unbound* | [*Function*] |
| **undefmethod** (*flavor* [*method-type*] *message*) | [*Macro*] |
| **without-cleaning-flavors** {*form*}* | [*Macro*] |

## Window System Tools

| | |
|---|---|
| **activate-viewport** *viewport* `&optional` *descendants* | [*Function*] |
| **active-region-bitmap** *active-region* | [*Function*] |
| **active-region-method** *active-region event-name* | [*Function*] |
| **active-region-p** *object* | [*Function*] |
| **attach-active-region** *bitmap active-region* | [*Function*] |
| **bitblt** *source-bitmap source-x source-y destination-bitmap destination-x destination-y width height operation* `&key :clipping-region` | [*Function*] |
| **bitblt-position** *source-bitmap source-position destination-bitmap destination-position width height operation* `&key :clipping-region` | [*Function*] |
| **bitblt-region** *source-bitmap source-region destination-bitmap destination-region operation* | [*Function*] |
| **bitmap-active-regions** *bitmap* | [*Function*] |
| **bitmap-extent** *bitmap* `&optional` *result-extent* | [*Function*] |
| **bitmap-height** *bitmap* | [*Function*] |
| **bitmap-output-stream-p** *object* | [*Function*] |
| **bitmap-p** *object* | [*Function*] |
| **bitmap-value** *bitmap x y* | [*Function*] |
| **bitmap-width** *bitmap* | [*Function*] |
| **charblt** *bitmap position font char* `&key :operation` | [*Function*] |
| **clear-bitmap** *bitmap* `&optional` *region* | [*Function*] |
| **clear-bitmap-active-regions** *bitmap* | [*Function*] |
| **copy-bitmap** *bitmap* | [*Function*] |
| **copy-font** *font new-name* | [*Function*] |
| **current-mouse-cursor** | [*Function*] |
| **deactivate-viewport** *viewport* | [*Function*] |
| ***default-font*** | [*Variable*] |
| **default-font-baseline** | [*Constant*] |

**default-font-code-limit** [*Constant*]

**default-font-height** [*Constant*]

**delete-font** *font* [*Function*]

**detach-active-region** *active-region* [*Function*]

**draw-circle** *bitmap center radius* [*Function*]
`&key :width :operation`

**draw-line** *bitmap start end* [*Function*]
`&key :width :operation`

**draw-polyline** *bitmap positions* [*Function*]
`&key :width :operation`

**draw-polypoint** *bitmap positions* [*Function*]
`&key :width :operation`

**expose-viewport** *viewport* [*Function*]

**extent-height** *extent* [*Function*]

**extent-width** *extent* [*Function*]

**extentp** *object* [*Function*]

**find-font** *name* [*Function*]

**font-baseline** *font* [*Function*]

**font-bitmap** *font* [*Function*]

**font-clear-char** *font char* [*Function*]

**font-code-limit** *font* [*Function*]

**font-fixed-width** *font* [*Function*]

**font-height** *font* [*Function*]

**font-name** *font* [*Function*]

**font-set-char** *font char offset* [*Function*]
`&optional` *width bitmap position*

**fontp** *object* [*Function*]

**hide-viewport** *viewport* [*Function*]

**initialize-windows** `&key :height :width` [*Function*]
```
                        :screen-x :screen-y
                        :label :icon-image
                        :icon-x :icon-y
                        :icon-label :icon-font
```

**keyboard-input** [*Function*]

**leave-window-system** *[Function]*

**listen-any** `&optional` ***mouse-input-stream*** *[Function]*

**load-bitmap** ***file-name*** *[Function]*

**load-font** ***file-name*** *[Function]*

**make-active-region** ***region*** `&key` *[Function]*
```
:bitmap
:mouse-left-down
:mouse-left-up
:mouse-middle-down
:mouse-middle-up
:mouse-right-down
:mouse-right-up
:mouse-moved
:mouse-still
:mouse-enter-region
:mouse-exit-region
```

**make-bitmap** `&key :extent :width :height` *[Function]*

**make-bitmap-output-stream** `&key` *[Function]*
```
:bitmap
:extent :width :height
:operation
:initial-font
```

**make-extent** `&optional` ***width height*** *[Function]*

**make-font** ***name*** `&key` *[Function]*
```
:bitmap :bitmap-width :code-limit
:fixed-width :height :baseline
```

**make-mouse-cursor** ***bitmap*** `&key` *[Function]*
```
:x-offset :y-offset
:operation
```

**make-mouse-input-stream** `&key` *[Function]*
```
:queue-mouse-events-p
:viewport
```

**make-pop-up-menu** ***choice-list*** `&optional` ***default-value*** *[Function]*

**make-position** `&optional` ***x y*** *[Function]*

**make-region** `&key` *[Function]*
```
:origin :x :y
:extent :width :height
:corner :corner-x :corner-y
```

**make-viewport** `&key` *[Function]*
```
:bitmap :width :height
:bitmap-region
:parent :fixed
:screen-position
:screen-x :screen-y
:activate
```

**make-window** `&key :position :x :y` *[Function]*
`:extent :width :height`
`:viewport-x :viewport-y`
`:inner-border-width :outer-border-width`
`:viewport-width :viewport-height`
`:initial-font :operation`
`:title :title-font`
`:parent :scroll :activate`
`:calculate-vertical-scroll-ratio`
`:calculate-horizontal-scroll-ratio`
`:vertical-scroll :horizontal-scroll`

**maximum-cursor-height** *[Constant]*

**maximum-cursor-width** *[Constant]*

**menu-mouse-buttons** *[Function]*

**mouse-buttons** *[Function]*

***mouse-buttons*** *[Variable]*

**mouse-cursor-bitmap** *mouse-cursor-object* *[Function]*

**mouse-cursor-operation** *mouse-cursor-object* *[Function]*

**mouse-cursor-p** *object* *[Function]*

**mouse-cursor-x-offset** *mouse-cursor-object* *[Function]*

**mouse-cursor-y-offset** *mouse-cursor-object* *[Function]*

**mouse-event-buttons** *mouse-event-object* *[Function]*

**mouse-event-event-type** *mouse-event-object* *[Function]*

**mouse-event-p** *object* *[Function]*

**mouse-event-x** *mouse-event-object* *[Function]*

**mouse-event-y** *mouse-event-object* *[Function]*

**mouse-input** *[Function]*

**mouse-input-stream-interrupt-char** *[Function]*
*mouse-input-stream char*

**mouse-input-stream-p** *object* *[Function]*

**mouse-input-stream-queue-mouse-events-p** *[Function]*
*mouse-input-stream*

**mouse-input-stream-viewport** *mouse-input-stream* *[Function]*

**mouse-x** *[Function]*

| | |
|---|---|
| ***mouse-x*** | [*Variable*] |
| **mouse-y** | [*Function*] |
| ***mouse-y*** | [*Variable*] |
| **move-mouse** *x y* | [*Function*] |
| **move-viewport** *viewport x y* | [*Function*] |
| **peek-any** &optional *peek-type mouse-input-stream eof-error-p eof-value recursive-p* | [*Function*] |
| **pop-up-menu-choose** *pop-up-menu-object* | [*Function*] |
| **pop-up-menu-p** *object* | [*Function*] |
| **position-x** *position* | [*Function*] |
| **position-y** *position* | [*Function*] |
| **positionp** *object* | [*Function*] |
| **read-any** &optional *mouse-input-stream eof-error-p eof-value recursive-p* | [*Function*] |
| **read-any-no-hang** &optional *mouse-input-stream eof-error-p eof-value recursive-p* | [*Function*] |
| **region-contains-point-p** *region x y* | [*Function*] |
| **region-contains-position-p** *region position* | [*Function*] |
| **region-corner** *region* &optional *result-position* | [*Function*] |
| **region-corner-x** *region* | [*Function*] |
| **region-corner-y** *region* | [*Function*] |
| **region-height** *region* | [*Function*] |
| **region-intersection** *region region* &rest *regions* | [*Function*] |
| **region-origin** *region* &optional *result-position* | [*Function*] |
| **region-origin-x** *region* | [*Function*] |
| **region-origin-y** *region* | [*Function*] |
| **region-size** *region* &optional *result-extent* | [*Function*] |
| **region-union** *region region* &rest *regions* | [*Function*] |
| **region-width** *region* | [*Function*] |
| **region/=** *region region* &rest *regions* | [*Function*] |
| **region<** *region region* &rest *regions* | [*Function*] |

**region<=** *region region* `&rest` *regions* [*Function*]

**region=** *region region* `&rest` *regions* [*Function*]

**region>** *region region* `&rest` *regions* [*Function*]

**region>=** *region region* `&rest` *regions* [*Function*]

**regionp** *object* [*Function*]

**rename-font** *font new-name* [*Function*]

**reshape-viewport** *viewport* `&key :region :x :y :width :height :corner-x :corner-y` [*Function*]

**root-viewport** [*Function*]

**store-bitmap** *bitmap file-name* [*Function*]

**store-font** *font file-name* [*Function*]

**stream-current-font** *bitmap-output-stream* [*Function*]

**stream-draw-circle** *bitmap-output-stream radius* `&key :width :operation` [*Function*]

**stream-draw-line** *bitmap-output-stream end* `&key :width :operation` [*Function*]

**stream-draw-polyline** *bitmap-output-stream positions* `&key :width :operation` [*Function*]

**stream-linefeed-distance** *bitmap-output-stream* [*Function*]

**stream-operation** *bitmap-output-stream* [*Function*]

**stream-position** *bitmap-output-stream* `&optional` *result-position* [*Function*]

**stream-x-position** *bitmap-output-stream* [*Function*]

**stream-y-position** *bitmap-output-stream* [*Function*]

**string-width** *string font* [*Function*]

**stringblt** *bitmap position font string* `&key :operation` [*Function*]

**unread-any** *char-or-mouse-event* `&optional` *mouse-input-stream* [*Function*]

**viewport-at-point** *x y* [*Function*]

**viewport-at-position** *position* [*Function*]

**viewport-bitmap** *viewport* [*Function*]

**viewport-bitmap-offset** *viewport* `&optional` *result-position* [*Function*]

| | |
|---|---|
| **viewport-bitmap-region** *viewport* `&optional` *result-region* | [*Function*] |
| **viewport-bitmap-x-offset** *viewport* | [*Function*] |
| **viewport-bitmap-y-offset** *viewport* | [*Function*] |
| **viewport-children** *viewport* | [*Function*] |
| **viewport-parent** *viewport* | [*Function*] |
| **viewport-screen-region** *viewport* `&optional` *result-region* | [*Function*] |
| **viewportp** *object* | [*Function*] |
| **window-frame** *window* | [*Function*] |
| **window-horizontal-scroll-ratio** *window* | [*Function*] |
| **window-inner-border-width** *window* | [*Function*] |
| **window-outer-border-width** *window* | [*Function*] |
| **window-title** *window* | [*Function*] |
| **window-title-font** *window* | [*Function*] |
| **window-vertical-scroll-ratio** *window* | [*Function*] |
| **windowp** *object* | [*Function*] |
| **windows-available-p** | [*Function*] |
| **with-asynchronous-method-invocation-allowed** {*form*}* | [*Macro*] |
| **with-fast-drawing-environment** {*form*}* | [*Macro*] |
| **with-mouse-methods-preempted** *bitmap* {*form*}* | [*Macro*] |

# Appendix C. Implementing Editor Commands

This appendix explains the data structures used by the Editor and describes the Lisp functions and macros that can be used to implement new Editor commands.

The names of the functions and macros explained in this appendix are external symbols of the package **editor**.

## Editor Data Types

The Editor uses various special data types. These include lines, marks, regions, buffers, windows, string-tables, and rings.

- A ***line*** is a portion of text containing no newline characters but delimited by a newline character or by the end of the text. Each line is separated from the next line by an implicit newline character.
- A ***mark*** is a pointer to a particular position within a line. Each mark points to a location between two characters (or between a character and the beginning or end of the text). The ***character position*** of a mark is equal to the number of characters that precede it in the line. Thus, the character position of a mark preceding the first character in the line is zero and that of a mark preceding a newline character is equal to the number of characters in the line.

  Each mark is one of three kinds: ***temporary***, ***left-inserting permanent***, or ***right-inserting permanent***. The two types of permanent marks differ only in their effects on inserted text, which goes to the left of a left-inserting mark and to the right of a right-inserting mark. A program may run faster by using a temporary mark instead of a permanent mark, but using a temporary mark after the text it points to has been altered will give undefined results. A permanent mark, however, continues to point to a valid position even after the text has been changed.
- A ***region*** is the area of text delimited by two marks, a starting mark and an ending mark. Changing the position of either of these marks changes the bounds of the region. Operations on regions in which the starting mark follows the ending mark are undefined.
- A ***buffer*** is a data structure that includes a region of text, a buffer name, a current position (the ***point***) within the region, an optional file, key bindings, modes, windows displaying the text, and some variables.

- A ***window*** is an object that allows you to view some portion of a buffer. Although each window displays only one buffer, a given buffer can be displayed in more than one window.
- A ***string-table*** is a table that associates case-insensitive names (strings) with objects. The names of variables, commands, modes, and buffers are stored in string-tables. Thus, all such names are case insensitive.
- A ***ring*** is a circular object that holds a fixed number of objects. When a new object is pushed into a ring, each object already in the ring is moved one position further from the beginning, and any object in the last position of the ring is removed. A ring is circular in that the objects it contains are contiguous even if the ring is not filled to capacity. A ring can be rotated in either direction without disturbing the contiguous ordering of its elements.

***in-the-editor*** [*Variable*]
This variable has the value t if you are currently in the Editor (that is, at some level within the function ed). Otherwise the value of ***in-the-editor*** is **nil**.

## Hooks

Certain Editor actions have ***hooks*** associated with them. An action's hook is a list of functions that are called before, or sometimes after, the action is taken. Calling the functions in a hook's list is referred to as ***invoking*** the hook. Some objects involved in the Editor action may be passed to the functions as arguments.

**add-hook** *place hook-fun* [*Macro*]
**remove-hook** *place hook-fun* [*Macro*]
These macros either add or remove a function from the hook list associated with the *place* argument, which should be either the symbol for an Editor variable or a generalized variable.

**invoke-hook** *name* **&rest** *args* [*Function*]
Calls the functions in the list that has the name given by the Editor variable *name*. The named variable must exist. Each of these functions is called with the arguments *args*.

## Commands

The ***command interpreter*** is the function that normally controls the Editor. It reads in keystrokes and invokes the corresponding commands.

Each command is implemented by some Lisp function. You can call a command by typing a key that is bound to that command, by executing Lisp code that calls the command's Lisp function, or by giving the command's name to the **Extended Command** prompt. A command's name is generally one or more capitalized words separated by spaces—**Scroll Window Down**, for example. The Lisp function names for most Editor commands are the same as the command names, but all spaces are replaced by hyphens and "-command" is appended—**scroll-window-down-command**, for example.

Every command has ***documentation*** that provides online help. The documentation for a given command can be either a string or a function. In a documentation string, the first line concisely describes the command, and the rest of the string provides particular details. A documentation function takes one argument, either **:short** or **:full**, indicating whether the function should return a short documentation string or fully document the command.

A ***key binding*** associates a single keystroke or a sequence of keystrokes with a given command. The keystroke or sequence of keystrokes is called a ***key***, and typing that key causes the command interpreter to invoke the command to which the key is bound.

A key binding can be global, or it can be local to a particular mode or buffer. The command interpreter looks first for a key binding local to the current buffer, then for a binding local to one of the current buffer's modes, and finally for a global key binding. If no applicable key binding exists, the command interpreter indicates an error by sounding a beep or by flashing the screen.

A ***transparent mode*** is one whose local key bindings do not shadow less local bindings. Such key bindings are themselves called "transparent." Normally the command interpreter stops searching when it finds the first applicable key binding. If that binding is transparent, however, the command interpreter invokes the corresponding command and continues to scan for other applicable key bindings, as if the transparent one had not existed. A key can have many transparent bindings and thus can cause many commands to be invoked.

When you call a command from Lisp code, you must include the prefix argument as the first argument to the Lisp command; **nil** is acceptable as a first argument and indicates the absence of a prefix argument. Arguments for which the command normally prompts are passed as optional arguments after the prefix argument. The command should prompt for any optional arguments that you do not supply.

***command-names*** [*Variable*]

This is the string-table of all commands.

**defcommand** {*command-name* | (*command-name function-name*)} *lambda-list command-doc function-doc* {*form*}* [*Macro*]

Creates the command *command-name*, which is called by the Lisp function *function-name*. If the *function-name* argument is omitted, the function name is the same as the command name, but all spaces are replaced by hyphens and "-command" is appended. The command documentation is specified as *command-doc*, and the function documentation is specified as *function-doc*.

**make-command** *name documentation function* [*Function*]
Creates the command *name* with the associated Lisp function *function* and with the command documentation supplied in the argument *documentation*.

**command-documentation** *command* [*Function*]
**command-function** *command* [*Function*]
**command-name** *command* [*Function*]
These functions return or set (with **setf**) either the documentation, the function, or the name of the specified command.

***invoke-hook*** [*Variable*]
Holds a function that the command interpreter calls in order to invoke a command. The arguments passed to this function are the desired command and its prefix argument. The normal value of ***invoke-hook*** is a function that calls the **command-function** of the command with the given prefix argument.

**bind-key** *name key* `&optional` *scope where* [*Function*]
Binds the *key* argument to the command *name*. The binding can be limited to the environment specified by the arguments *scope* and *where*. The argument *scope* can have one of the following values:

- **:global** makes the binding global; this is the default.
- **:mode** makes the binding local to the mode *where*.
- **:buffer** makes the binding local to the buffer *where*.

**command-bindings** *command* [*Function*]
Returns a list of the key bindings for the argument *command*. Each key binding returned is a list of three items: the key vector, the scope of the binding, and any buffer or mode to which the binding is local. If the binding is global, the last item in the list is **nil**.

**link-key** *key1 scope1 where1 key2* `&optional` *scope2 where2* [*Function*]
Creates a binding for the argument *key1* in the environment defined by the arguments *scope1* and *where1*. The binding created is identical to the binding for *key2* in the environment defined by the arguments *scope2* (which defaults to :global) and *where2*. See **bind-key** for the meanings of the environment-defining arguments. Cross-mode or cross-buffer bindings may have unexpected results.

**delete-key-binding** *key* `&optional` *scope where* [*Function*]
Deletes the binding for the *key* argument in the environment defined by the arguments *scope* (default **:global**) and *where*. See **bind-key** for the meanings of the environment-defining arguments.

**get-command** *key* `&optional` *scope where* [*Function*]

Returns the command to which the argument *key* is bound in the environment defined by the arguments *scope* and *where*. If the *scope* argument is **:current** or omitted, multiple values are returned. These values are the current command bound in the current buffer and a list of the commands of any current transparent bindings for the argument *key*. Otherwise only one command value is returned, with the arguments *scope* and *where* interpreted as in **bind-key**.

**map-bindings** *function scope* `&optional` *where* [*Function*]

Maps the argument *function* over the key bindings in the environment defined by the arguments *scope* and *where*, which are interpreted as in **bind-key**. For each such key binding, the given function is called with two arguments: the key that is bound and the command to which it is bound.

**last-command-type** [*Function*]

Returns or sets (with **setf**) the ***last command type***. If a completed command has not set the command type, the value is set to **nil** unless the command was invoked because of a transparent key binding. Command types are usually keywords.

**prefix-argument** *argument* [*Function*]

Returns or sets (with **setf**) the ***command prefix argument***. If a completed command has not set the prefix argument, the value is set to **nil** unless the command was invoked because of a transparent key binding. The prefix argument should be either an integer or **nil**. The Lisp function for a command is passed the command's prefix argument as the first Lisp argument.

**recursive-edit** [*Function*]

**Enter Recursive Edit Hook** [*Variable*]

The function **recursive-edit** calls the command interpreter recursively. The command interpreter reads keystrokes and invokes commands until either the function **exit-recursive-edit** or the function **abort-recursive-edit** is called. The hook **Enter Recursive Edit Hook** is invoked before the command interpreter is called.

**exit-recursive-edit** `&optional` *values-list* [*Function*]

**Exit Recursive Edit Hook** [*Variable*]

The function **exit-recursive-edit** terminates the command interpreter and returns as multiple values all the objects in the list *values-list*, which defaults to **nil**. This function ends the innermost recursive edit, if one exists. If there is no recursive edit in progress, the top level of the command interpreter, which is invoked by a call to the function ed, is terminated. The indicated multiple values are returned by **recursive-edit** or by **ed**. The hook **Exit Recursive Edit Hook** is invoked after the recursive edit terminates.

**abort-recursive-edit** **&rest** *args* [*Function*]

**Abort Recursive Edit Hook** [*Variable*]

The function **abort-recursive-edit** terminates the command interpreter with the error indicated by the arguments, which are processed just like the arguments to **format**. The resulting formatted string is returned by the innermost **recursive-edit** in progress, if any, or by **ed**. Before the termination, the hook **Abort Recursive Edit Hook** is invoked with the same arguments given to **abort-recursive-edit**.

## Lines

**line-string** *line* [*Function*]

Returns as a simple string or sets (with **setf**) the characters in the given line. A line must not be set to a string containing newline characters.

**line-previous** *line* [*Function*]

**line-next** *line* [*Function*]

These functions return either the line before or the line after the given line or **nil** if there is no such preceding or succeeding line.

**line-buffer** *line* [*Function*]

Returns the buffer that contains the given line. If the line is not part of any buffer, **nil** is returned.

**line-length** *line* [*Function*]

Returns the number of characters in the specified line, not counting the implicit newline character at the end.

**line-character** *line index* [*Function*]

Returns the character that is preceded in the given line by the number of characters specified by the *index* argument. If the index equals the number of characters in the line, then a newline character is returned. The index value must not be negative or greater than the number of characters in the line.

**line-plist** *line* [*Function*]

Returns the property list for the specified line. A line's property list is set to **nil** whenever the text of the line is changed. The property list can be accessed with **setf**, **getf**, and **remf** and can be used to cache information about the line.

## Marks

**mark-line** ***mark*** [*Function*]

Returns the line into which the specified mark points.

**mark-charpos** ***mark*** [*Function*]

Returns the character position to which the specified mark points.

**mark-kind** ***mark*** [*Function*]

Returns or sets (with **setf**) the kind of the given mark—**:right-inserting**, **:left-inserting**, or **:temporary**.

**previous-character** ***mark*** [*Function*]

**next-character** ***mark*** [*Function*]

These functions return or set (with **setf**) either the character just before or the character just after the specified mark; **nil** means there is no such character. Setting the character causes the character formerly in that position to be replaced by the new character.

**mark** ***line charpos*** `&optional` ***kind*** [*Function*]

Creates a mark that points into the specified line at the character position ***charpos***, where zero is the position just before the first character on the line. The argument ***kind*** specifies the kind of mark created—**:right-inserting**, **:left-inserting**, or **:temporary**. The default is **:temporary**.

**copy-mark** ***mark*** `&optional` ***kind*** [*Function*]

Creates a mark pointing at the same position as the given mark. The argument ***kind*** specifies the kind of the new mark; the default is the kind of the original mark.

**delete-mark** ***mark*** [*Function*]

Deletes the specified mark. Permanent marks should be deleted when no longer in use.

**with-mark** ({(*mark pos* [*kind*])}*) {*form*}* [*Macro*]

Evaluates the given forms; each variable ***mark*** is bound to a mark that points to the position given by the corresponding mark ***pos***. The kind of each mark is given by the associated ***kind*** argument; the default is the kind of the corresponding mark ***pos***. The value of the last form is returned.

**move-to-position** ***mark charpos*** `&optional` ***line*** [*Function*]

Points the specified mark at the character position ***charpos*** on the given line. If no ***line*** argument is given, the default is the line to which the mark currently points.

**move-to-mark** ***mark new-position*** [*Function*]

Points the argument ***mark*** at the position pointed to by the mark ***new-position*** and returns the modified argument ***mark***.

**line-start** ***mark*** `&optional` ***line*** [*Function*]

**line-end** ***mark*** `&optional` ***line*** [*Function*]

These functions point the specified mark at either the beginning or the end of the given line. If no ***line*** argument is given, the default is the line to which the mark currently points.

**buffer-start** ***mark*** `&optional` ***buffer*** [*Function*]

**buffer-end** ***mark*** `&optional` ***buffer*** [*Function*]

These functions point the specified mark at either the beginning or the end of the given buffer. If no ***buffer*** argument is given, the default is the buffer into which the mark currently points; the mark must point into some buffer.

**mark-before** ***mark*** [*Function*]

**mark-after** ***mark*** [*Function*]

These functions point the specified mark either one character position before or one character position after its current position. If there is no such character before or after the mark, the mark is unchanged and **nil** is returned.

**character-offset** ***mark n*** [*Function*]

Points the specified mark ***n*** characters after its current position or -***n*** characters before its current position if ***n*** is negative. If the specified number of characters do not occur before or after the mark, the mark is unchanged and **nil** is returned.

**line-offset** ***mark n*** `&optional` ***charpos*** [*Function*]

Points the specified mark ***n*** lines past its current position or -***n*** lines before its current position if ***n*** is negative. If there are not the specified number of lines before or after the mark, the mark is unchanged and **nil** is returned. When the mark is changed, the resulting character position within the new line is the minimum of the argument ***charpos*** and the length of the new line, where ***charpos*** defaults to the original character position of the mark.

## Regions

**region** ***start end*** [*Function*]

Creates a region from the two marks ***start*** and ***end***. These marks must point into the same piece of text (such as a buffer), and the mark ***start*** must not follow the mark ***end***.

**make-empty-region** [*Function*]

Creates a region whose starting and ending marks point to the beginning of an empty line. The starting mark is right inserting and the ending mark is left inserting.

**copy-region** ***region*** [*Function*]

Creates a region that contains a copy of the text in the specified region.

**region-to-string** ***region*** [*Function*]
**string-to-region** ***string*** [*Function*]
These functions coerce either the specified region to a string or the specified string to a region. Lines within a string are separated by newline characters.

**line-to-region** ***line*** [*Function*]
Creates a region that contains the characters in the specified line. The starting mark is right inserting and the ending mark is left inserting.

**region-start** ***region*** [*Function*]
**region-end** ***region*** [*Function*]
These functions return either the starting or the ending mark of the specified region.

**region-bounds** ***region*** [*Function*]
Returns as multiple values the starting mark and the ending mark of the specified region.

**set-region-bounds** ***region start end*** [*Function*]
Makes the marks ***start*** and ***end*** the starting and ending marks of the specified region. The starting and ending marks must be in the same piece of text (for instance, the same buffer), and the starting mark must not follow the ending mark.

**count-lines** ***region*** [*Function*]
Returns the number of lines (or partial lines) in the specified region. A newline character is considered to be part of the line that it ends.

**count-characters** ***region*** [*Function*]
Returns the number of characters in the specified region, including the implicit newline character at the end of each line.

## Buffers

**current-buffer** [*Function*]
**Set Buffer Hook** [*Variable*]
**After Set Buffer Hook** [*Variable*]
The function **current-buffer** returns or sets (with **setf**) the currently selected buffer, which is generally the buffer displayed in the current window. Before the current buffer is changed, the hook **Set Buffer Hook** is invoked with the new buffer as its argument, and after the current buffer has been changed, the hook **After Set Buffer Hook** is invoked with the old buffer as its argument.

**current-point** [*Function*]
Returns the mark that is the current point within the current buffer.

***buffer-list*** *[Variable]*
This is a list of all the buffers the Editor may presently access.

***buffer-names*** *[Variable]*
This is the string-table of all buffers.

**make-buffer** *name &optional modes* *[Function]*
**Make Buffer Hook** *[Variable]*
The function **make-buffer** creates the buffer *name*. If a buffer with that name already exists, no buffer is created and **nil** is returned. The argument *modes* is a list of modes to be active in the buffer, starting with the major mode. If the *modes* argument is omitted, the modes listed in the Editor variable **Default Modes** are made active in the new buffer. The new buffer is added to the list ***buffer-list*** and to the string-table ***buffer-names***. Whenever a buffer is created, the hook **Make Buffer Hook** is invoked with the new buffer as its argument.

**buffer-name** *buffer* *[Function]*
**Buffer Name Hook** *[Variable]*
The function **buffer-name** returns or sets (with **setf**) the name of the specified buffer. A buffer name is a string. If a buffer exists with the same name as the name being set, no name is changed and **nil** is returned. Whenever a buffer name is changed, the hook **Buffer Name Hook** is invoked with the buffer and the new name as arguments.

**buffer-region** *buffer* *[Function]*
Returns or sets (with **setf**) the region of the specified buffer.

**buffer-pathname** *buffer* *[Function]*
**Buffer Pathname Hook** *[Variable]*
The function **buffer-pathname** returns or sets (with **setf**) the pathname of the file associated with the specified buffer. A value of **nil** means no file is associated with the buffer. Whenever a pathname is changed, the hook **Buffer Pathname Hook** is invoked with the buffer and the new pathname as arguments.

**buffer-point** *buffer* *[Function]*
Returns the mark that is the current point within the specified buffer.

**buffer-writable** *buffer* *[Function]*
Returns or sets (with **setf**) a flag indicating whether the specified buffer's text can be changed. The flag is **t** if the text can be changed; otherwise it is **nil**.

**buffer-modified** *buffer* *[Function]*
Returns or sets (with **setf**) a flag indicating whether the specified buffer's text has been changed. The flag is **t** if the text has been changed; otherwise it is **nil**.

**buffer-variables** *buffer* *[Function]*
Returns the string-table of all the Editor variables local to the specified buffer.

**buffer-windows** *buffer* [*Function*]

Returns a list of all the windows in which the specified buffer might currently appear. This list may include windows that are not currently visible.

**delete-buffer** *buffer* [*Function*]

**Delete Buffer Hook** [*Variable*]

The function **delete-buffer** deletes the specified buffer by removing it from the list ***buffer-list*** and from the string-table ***buffer-names***. The hook **Delete Buffer Hook** is invoked with the buffer as its argument before the deletion occurs.

## Predicates

**linep** *object* [*Function*]

**markp** *object* [*Function*]

**editor-region-p** *object* [*Function*]

**bufferp** *object* [*Function*]

**editor-window-p** *object* [*Function*]

**string-table-p** *object* [*Function*]

**ringp** *object* [*Function*]

**commandp** *object* [*Function*]

These functions are predicates that return a non-**nil** value if the argument *object* is of the given type; otherwise they return **nil**.

**start-line-p** *mark* [*Function*]

Returns **t** if the specified mark is just before the first character in a line; otherwise it returns **nil**.

**end-line-p** *mark* [*Function*]

Returns **t** if the specified mark is just after the last character in a line (that is, just before a newline character); otherwise it returns **nil**.

**empty-line-p** *mark* [*Function*]

Returns **t** if the line pointed into by the specified mark has no characters in it (not counting the newline character); otherwise it returns **nil**.

**blank-line-p** *line* [*Function*]

Returns **t** if all characters in the specified line have a **:whitespace** character attribute of 1; otherwise it returns **nil**.

**blank-before-p** *mark* [*Function*]

**blank-after-p** *mark* [*Function*]

These functions return **t** if all characters between the specified mark and the beginning or end of the line containing it have a **:whitespace** character attribute of 1; otherwise they return **nil**.

**same-line-p** *mark1 mark2* [*Function*]

Returns **t** if the two marks *mark1* and *mark2* point into the same line; otherwise it returns **nil**.

**mark<** *mark1 mark2* [*Function*]

**mark<=** *mark1 mark2* [*Function*]

**mark>=** *mark1 mark2* [*Function*]

**mark>** *mark1 mark2* [*Function*]

These functions return **t** if the two specified marks follow the given relative ordering within a piece of text (such as a buffer); otherwise they return **nil**. Both marks must be within the same piece of text.

**mark=** *mark1 mark2* [*Function*]

**mark/=** *mark1 mark2* [*Function*]

These functions return **t** if the two specified marks do (**mark=**) or do not (**mark/=**) point to the same location in the same piece of text (for instance, the same buffer); otherwise they return **nil**.

**line<** *line1 line2* [*Function*]

**line<=** *line1 line2* [*Function*]

**line>=** *line1 line2* [*Function*]

**line>** *line1 line2* [*Function*]

These functions return **t** if the two specified lines follow the given relative ordering in a piece of text (such as a buffer); otherwise they return **nil**. The two lines must be in the same piece of text.

**lines-related** *line1 line2* [*Function*]

Returns **t** if the two lines *line1* and *line2* are in the same piece of text (for instance, the same buffer); otherwise it returns **nil**.

**first-line-p** *mark* [*Function*]

**last-line-p** *mark* [*Function*]

These functions return **t** if there is no text before (**first-line-p**) or no text after (**last-line-p**) the line into which the specified mark points; otherwise they return **nil**.

## String-Tables

**make-string-table** [*Function*]
Creates and returns an empty string-table.

**delete-string** ***string table*** [*Function*]
Deletes the specified string from the string-table ***table***.

**getstring** ***string table*** [*Function*]
Accesses the entry for the specified string in the string-table ***table*** and returns multiple values or sets a single value with **setf**. The first value returned is the value associated with the given string or **nil** if the string is not in the table. The second value returned is **t** if the string is in the table and **nil** if it is not. When **setf** is used with **getstring**, the given string is added to the table if necessary, and the value associated with the string is set to the value specified by **setf**.

**complete-string** ***string tables*** [*Function*]
Returns multiple values indicating the best match of the specified string with the names in the given string-tables. The ***tables*** argument is a list of the string-tables to be searched. The first value returned is the longest leading substring that is common to all the names of which the specified string is a leading substring. If there is only one such name, the value for that name's table entry is returned as the second value and **t** is returned as the third value; otherwise the second and third values returned are **nil**. If no name in the tables starts with the specified string, all three values returned are **nil**.

**find-ambiguous** ***string table*** [*Function*]
**find-containing** ***string table*** [*Function*]
These functions return an alphabetically ordered list of the strings in the specified table that contain the substring ***string***. For the function **find-ambiguous**, the list includes only those strings whose leading substrings match the ***string*** argument.

**do-strings** (***string-var value-var table***) {***declaration***}* [*Macro*]
{***tag*** | ***statement***}*

Executes the given ***statement*** body once for each entry in the given string-table. Execution is in alphabetical order by the table's strings. During a given iteration, the variable ***string-var*** is bound to the string of an entry, and the variable ***value-var*** is bound to the value of that entry.

## Rings

**make-ring** *length* &optional *delete-function* [*Function*]

Creates a ring that can hold the number of Lisp objects indicated by the argument *length*, which must be a positive number. The *delete-function* argument specifies a function that is called with each object just before the object is removed from the end of the ring.

**ring-length** *ring* [*Function*]

Returns multiple values consisting of the number of objects currently in the specified ring and the total number of objects the ring can hold.

**ring-ref** *ring index* [*Function*]

Returns or sets (with **setf**) the indexed object in the specified ring. An index of zero represents the last object pushed into the ring (the object at the beginning of the ring).

**ring-push** *object ring* [*Function*]

Pushes the specified object into the ring indicated. If the ring already contains as many objects as it can hold, the object at the end of the ring is removed (and passed to the ring's *delete-function*; see **make-ring**).

**ring-pop** *ring* [*Function*]

Removes and returns the object that is at the beginning of the specified ring, which must not be empty. All other objects in the ring are moved one position closer to the beginning of the ring.

**rotate-ring** *ring offset* [*Function*]

Rotates the specified ring by the number of positions indicated by the *offset* argument. A positive number moves each object that much closer to the beginning of the ring, with elements that pass the beginning of the ring moved to the end. A negative number causes rotation in the other direction.

## Changing Text

**insert-character** *mark character* [*Function*]
**insert-string** *mark string* [*Function*]
**insert-region** *mark region* [*Function*]

These functions insert the specified character, string, or region at the mark indicated.

**ninsert-region** *mark region* [*Function*]

Destructively inserts the specified region (which should not be referenced elsewhere) at the mark indicated. The region should not already be a part of a buffer.

**delete-characters** ***mark n*** [*Function*]

Deletes the *n* characters following the specified mark or the -*n* characters preceding the mark if *n* is negative; returns **t**. If there are less than *n* characters after the mark (or less than -*n* characters before the mark), no characters are deleted and **nil** is returned.

**delete-region** ***region*** [*Function*]

Deletes the specified region without copying the text to save it (see **delete-and-save-region** below).

**delete-and-save-region** ***region*** [*Function*]

Deletes the specified region and returns a region containing a copy of the text from the deleted region.

**filter-region** ***function region*** [*Function*]

Destructively alters the specified region by mapping the function indicated over simple strings containing the region's text lines, in order. The region is reconstructed from the function output. The function must not destructively modify the strings passed to it and must not return a string containing a newline character. Destructively modifying such a string after it has been returned will not change the reconstructed region.

## Searching and Replacement

**new-search-pattern** ***kind direction pattern*** [*Function*]
`&optional` ***result-search-pattern***

Creates a search-pattern object that can be passed to **find-pattern** or **replace-pattern**. The argument ***result-search-pattern***, if given, should be a search-pattern to be destructively modified to the new pattern. The argument ***direction*** specifies the search direction—**:forward** or **:backward**. The argument ***pattern*** specifies what to search for; its interpretation depends on the ***kind*** argument, which can have one of the following values:

- **:string-insensitive** makes a case-insensitive search-pattern from the search string ***pattern***.
- **:string-sensitive** makes a case-sensitive search-pattern from the search string ***pattern***.
- **:character** makes a case-sensitive search-pattern to find the character ***pattern***.
- **:not-character** makes a case-sensitive search-pattern to find any character except the character ***pattern***.
- **:test** makes a search-pattern to find a character for which the function ***pattern*** returns a non-**nil** value. The function should depend only on its one character argument, and it should have no side effects.

- **:test-not** makes a search-pattern to find a character for which the function *pattern* returns **nil**.
- **:any** makes a search-pattern to find any character in the string *pattern*.
- **:not-any** makes a search-pattern to find any character that is not in the string *pattern*.

**find-pattern** *mark search-pattern* [*Function*]
Starting at the specified mark, searches for the first match of the pattern given by the *search-pattern* argument. A successful search moves the mark to the position preceding the matching text and returns the number of characters matched. An unsuccessful search leaves the mark unchanged and returns **nil**.

**replace-pattern** *mark search-pattern replacement* `&optional` *n* [*Function*]
Starting at the specified mark, replaces the first *n* matches of the given search-pattern with the string *replacement*. If *n* is omitted or **nil**, all matches found are replaced. This function returns a mark pointing to the position preceding the place where the last replacement occurred.

## Modes

The modes selected in a buffer affect the operation of the Editor. For instance, a given key may invoke different commands in different modes. Each mode has a case-insensitive name, such as **Lisp**, that generally indicates what sort of editing the mode is intended to facilitate.

A given mode is either a ***major mode*** or a ***minor mode***. There is always just one major mode selected in each buffer, but any number of minor modes can be selected in a buffer.

Each mode has a hook, which is a list of functions that are called whenever the mode is entered or exited for a given buffer. Each function in the hook list is passed two arguments: the buffer whose mode is being changed and **t** if the mode is being entered or **nil** if the mode is being exited. The mode hook name is made from the mode name by changing spaces in the name to hyphens and then appending `-mode-hook`.

**Default Modes** [*Variable*]
This is a list of the names of the modes initially selected in any buffer created without an explicit list of modes. The initial value of **Default Modes** is `("Fundamental")`.

***mode-names*** [*Variable*]
This is the string-table of all modes.

**defmode** *name* &key :setup-function :cleanup-function :major-p :precedence :transparent-p [*Function*]

Creates a mode named by the string *name* and adds it to the string-table ***mode-names***. The mode will be a minor mode unless **:major-p** is supplied and non-**nil**, in which case it will be a major mode. The functions **:setup-function** and **:cleanup-function** are called after the mode is entered and before it is exited respectively. The number **:precedence**, which applies only to minor modes, determines the mode's precedence for the values of key bindings, Editor variables, and character attributes. The order in which these values are chosen is explained in the section "Scopes and Shadowing." If **:transparent-p** is given and non-**nil**, the mode and all key bindings local to it are transparent. Commands for transparent bindings are invoked without terminating the command interpreter's search for the first nontransparent binding of a key. The function **defmode** also creates a hook for the new mode by appending `-mode-hook` to the mode name and by changing spaces in the name to hyphens.

**buffer-major-mode** *buffer* [*Function*]
**Buffer Major Mode Hook** [*Variable*]

The function **buffer-major-mode** returns or sets (with **setf**) the name of the major mode for the specified buffer. After a buffer's major mode is changed, the hook **Buffer Major Mode Hook** is invoked with the buffer and the new mode as arguments.

**buffer-minor-mode** *buffer name* [*Function*]
**Buffer Minor Mode Hook** [*Variable*]

The function **buffer-minor-mode** returns or sets (with **setf**) a flag indicating whether the minor mode *name* is selected in the specified buffer. The flag is **t** if the mode is selected or **nil** if it is not selected. After a minor mode is changed, the hook **Buffer Minor Mode Hook** is invoked with the buffer, the mode name, and the new flag value as arguments.

**mode-variables** *name* [*Function*]

Returns the string-table of all Editor variables local to the mode specified by the *name* argument.

**mode-major-p** *name* [*Function*]

Returns **t** if the mode with the specified name is a major mode or **nil** if it is a minor mode. A mode with the given name must exist or an error is signaled.

## Scopes and Shadowing

The scopes of key bindings, Editor variables, and character attributes may be global, local to a mode, or local to a buffer. A buffer-local value applies only if its buffer is the current buffer. A mode-local value applies only if its mode is selected in the current buffer. A global value applies everywhere. If more than one value applies in the current environment, however, any applicable local values shadow less local ones, with the following selection priority (highest first):

- An applicable buffer-local value.
- Applicable mode-local values for minor modes, in order by precedence of the modes, highest precedence first. For minor modes of equal precedence, any applicable value may be selected.
- An applicable mode-local value for the major mode.
- An applicable global value.

## Editor Variables

Editor variables provide useful facilities for the Editor that regular Lisp variables cannot provide. The principal facilities of Editor variables are scoping appropriate to the Editor, with mode- and buffer-local variables; hooks, which are lists of functions to be called whenever a variable's value is set; and documentation associated with each variable name so that variables and their meanings can be found from within the Editor.

An Editor variable's hook is different from an Editor action's hook (such as **Set Buffer Hook**). An Editor variable's hook is stored in its hook cell, but an Editor action's hook is stored in the cell that contains the value of the Editor variable associated with the action. A variable's hook is invoked when the variable is explicitly set, and an action's hook is invoked when the corresponding Editor action takes place.

The name of an Editor variable has two forms: a case-insensitive string and a symbol. The string name normally consists of one or more words separated by spaces. The corresponding symbol name is made by changing each space in the string name into a hyphen and then entering the resulting symbol into the package **editor** as an external symbol.

***global-variable-names*** [*Variable*]
This is the string-table of all global Editor variables. The value for an entry is the variable's symbol name. See also ***buffer-variables*** and ***mode-variables***.

**editor-defvar** *string-name documentation* &key :mode :buffer :hooks :value [*Function*]

Creates an Editor variable with the name *string-name*. The argument *documentation* is the variable's documentation string. The variable is made local to a given buffer by including the argument **:buffer**, or it is made local to a given mode by including the argument **:mode**. If neither of these keywords is used, the variable is global. The arguments **:hooks** and **:value** specify the variable's initial hook list and value, each of which defaults to **nil**. If the given variable already exists with the given scope, this function sets the variable's hook list and value from any corresponding arguments that are given.

**variable-value** *name* &optional *scope where* [*Function*]

**variable-documentation** *name* &optional *scope where* [*Function*]

**variable-hooks** *name* &optional *scope where* [*Function*]

**variable-name** *name* &optional *scope where* [*Function*]

These functions return either the value, the documentation, the hook list, or the string name of the Editor variable *name*. The optional *scope* argument can have one of the following values:

- **:current** selects the variable indicated by the current environment (checking for buffer-local and mode-local variables first, then global variables); this is the default scope.
- **:global** selects a global variable.
- **:mode** selects the variable local to the mode *where*.
- **:buffer** selects the variable local to the buffer *where*.

**string-to-variable** *string* [*Function*]

Returns the symbol name corresponding to the Editor variable with the string name *string*. The given string does not have to name an actual Editor variable.

**value** *name* [*Macro*]

**setv** *name new-value* [*Macro*]

The macro **value** returns or sets (with **setf**) the current value of the Editor variable *name*. The macro **setv** sets the value of the Editor variable *name* to the value of the argument *new-value*. In each case, the *name* argument is not evaluated.

**editor-let** ({(*var value*)}*) {*form*}* [*Macro*]

Evaluates the given forms with each of the given Editor variables bound to the corresponding value. The bindings exist only within the macro **editor-let**, which returns the value of the last supplied form. No hooks are invoked for these bindings.

**editor-boundp** *name* `&optional` *scope where* [*Function*]

Returns a non-nil value if there is an Editor variable with the given name in the appropriate environment, which is interpreted as in **variable-value**; otherwise it returns **nil**.

**delete-variable** *name* `&optional` *scope where* [*Function*]

**Delete Variable Hook** [*Variable*]

The function **delete-variable** deletes the variable *name*, which must exist. The deletion may be limited to a particular environment by specifying the *scope* and *where* arguments, which have the same meanings as for **variable-value**; **:current** is not allowed, however, and the default scope is global. Before the variable is deleted, the hook **Delete Variable Hook** is invoked with the same arguments.

## Character Attributes

The Editor maintains ***attributes*** for each character. Such attributes can be tested—to tell if a character is whitespace, for instance. Editor character attributes are normally global but can also be mode local to major modes. Attributes can be readily changed.

The name of a character attribute has two forms: a case-insensitive string and a symbol. The string name normally consists of one or more capitalized words separated by spaces. The corresponding symbol name is made by changing each space in the string name into a hyphen and then entering the resulting symbol into the keyword package. For instance, `"Whitespace"` and **:whitespace** are the string name and symbol name of one attribute.

Each character attribute has a hook, which is a list of functions that are called whenever an attribute value is changed. Each function is called with the attribute, the character, and the new attribute value as arguments.

***character-attribute-names*** [*Variable*]

This is the string-table of character attributes.

**defattribute** *string-name documentation*
`&optional` *type initial-value* [*Function*]

**Make Character Attribute Hook** [*Variable*]

The function **defattribute** creates a character attribute with the name *string-name*. The argument *documentation* is the attribute's documentation string. The values of this character attribute may be given a Lisp type, which can be any type permitted in **make-array**. Each character has the specified initial value for this attribute if the *initial-value* argument is given. After the attribute is defined, the hook **Make Character Attribute Hook** is invoked with the same arguments.

**character-attribute-name** *attribute* [*Function*]

**character-attribute-documentation** *attribute* [*Function*]

These functions return either the name or the documentation for the specified character attribute.

**character-attribute** *attribute character* [*Function*]

**Character Attribute Hook** [*Variable*]

The function **character-attribute** returns or sets (with **setf**) the value of the specified attribute for the character indicated. The attribute must already be defined. Before an attribute's value is set, the hook **Character Attribute Hook** is invoked with the same arguments. A *character* argument of **nil** represents the beginning or end of a buffer; this allows special attributes to be associated with a buffer boundary.

**character-attribute-p** *symbol* [*Function*]

Returns **t** if the specified symbol is a character attribute's name; otherwise it returns **nil**.

**shadow-attribute** *attribute character value mode* [*Function*]

**Shadow Attribute Hook** [*Variable*]

The function **shadow-attribute** sets a mode-local value, given by the *value* argument, for the specified attribute, character, and major mode. When this function is called, the hook **Shadow Attribute Hook** is invoked with the same arguments. While a value set by **shadow-attribute** is in effect, setting the given character attribute sets only the shadowing value, not the global value.

**unshadow-attribute** *attribute character mode* [*Function*]

**Unshadow Attribute Hook** [*Variable*]

The function **unshadow-attribute** removes the shadowing value for the given attribute, character, and mode. The hook **Unshadow Attribute Hook** is invoked with the same arguments.

**find-attribute** *mark attribute* `&optional` *test* [*Function*]

**reverse-find-attribute** *mark attribute* `&optional` *test* [*Function*]

These functions search forward and backward respectively from the specified mark for the first character found whose value for the *attribute* argument satisfies the given test function. The test function, which should take one argument and have no side effects, defaults to **not zerop**. If the search succeeds, the mark is modified to point to the position preceding (or succeeding, for **reverse-find-attribute**) the character found; otherwise the mark is unchanged and **nil** is returned.

**character-attribute-hooks** *attribute* [*Function*]

Returns, sets (with **setf**), or modifies (with **add-hook** or **remove-hook**) the hook list for the specified attribute.

## Logical Characters

Logical characters make it easy to redefine the possible keyboard responses indicating a particular choice. A ***logical character*** is a keyword that represents some one-character response from the keyboard, such as a **:help** inquiry. Each logical character represents a single response, but that response may be assigned to several characters. Furthermore, a given character may correspond to several logical characters; the character's interpretation at a given time depends on the program reading the response.

The following standard logical characters are predefined:

- **:yes** means the action in question should be taken.
- **:no** means the action in question should not be taken.
- **:do-all** means the action in question should be repeated as many times as possible.
- **:exit** means the command should end normally.
- **:help** means the command should provide some helpful information.
- **:confirm** confirms the input, if any; otherwise it means the default action should be taken.
- **:quote** means the following character should be treated simply as a character, not as a command.
- **:recursive-edit** means the command should begin a recursive edit from the current environment.

A new logical character should be defined for any response that represents a general action that might be invoked from different commands, for any response that needs to be easy to change, or for a character that is not **standard-char-p** and that thus may be implementation dependent.

***logical-character-names*** [*Variable*]
This is the string-table of all logical characters. The value of an entry is a logical character's keyword.

**define-logical-character** *string-name documentation* [*Function*]
Creates a logical character with the name *string-name*, the documentation string *documentation*, and the keyword made by replacing all spaces in the string name with hyphens.

**logical-character-name** ***keyword*** [*Function*]
**logical-character-documentation** ***keyword*** [*Function*]
These functions return either the string name or the documentation of the logical character ***keyword***.

**logical-char=** ***character keyword*** [*Function*]
Returns or sets (with **setf**) a flag that is non-**nil** if the specified character has the ***keyword*** argument as a logical character; if the ***keyword*** argument is not a logical character of the ***character*** argument, the flag returned or set is **nil**. If the ***character*** argument is a letter, its case is ignored; thus, uppercase and lowercase letters are equivalent. Bits and fonts, however, are significant. The argument ***keyword*** must be defined as a logical character.

**logical-character-characters** ***keyword*** [*Function*]
Returns a list of the characters that have the ***keyword*** argument as a logical character.

## The Display

A ***window*** is an object that allows you to view some portion of a buffer. Although each window displays only one buffer, a given buffer can be displayed in more than one window.

The current window is the one in which the cursor appears. The cursor follows the current point within that buffer. Whenever the point is moved to a location that is not on the screen, the window is recentered around the point's new location in the buffer.

A ***mode line*** is a text line that may be displayed at the bottom of a window to indicate the status of the window and its displayed buffer. A mode line is specified by a format control string and a function. The string generates the text of the mode line, and the function generates as multiple values any arguments to be used by the string for formatting. The window containing the mode line is passed as the argument to the function.

**current-window** [*Function*]
**Set Window Hook** [*Variable*]
The function **current-window** returns or sets (with **setf**) the current window (where the cursor appears). Before the current window is set, the hook **Set Window Hook** is invoked with the new window as its argument.

**window-modeline-string** ***window*** [*Function*]
**window-modeline-function** ***window*** [*Function*]
These functions return or set (with **setf**) either the mode-line string or the mode-line function for the specified window.

**update-window-modeline** *window* [*Function*]

Causes the mode line for the specified window to be regenerated and updated. To keep the mode-line display current, this function can be called from hooks on the variables that are displayed in the mode line.

**make-editor-window** *mark* `&optional` *modeline-string modeline-function* [*Function*]

**Make Editor Window Hook** [*Variable*]

**Default Modeline String** [*Variable*]

**Default Modeline Function** [*Variable*]

The function **make-editor-window** creates a new window to display the text starting at the specified mark, which must be in some buffer. The arguments *modeline-string* and *modeline-function* specify the window's mode line. The window has no mode line if *modeline-string* is **nil**. If no *modeline-string* argument is given, the value of **Default Modeline String** is used; if no *modeline-function* argument is given, the value of **Default Modeline Function** is used. The hook **Make Editor Window Hook** is invoked with the new window as its argument.

***window-list*** [*Variable*]

This is a list of all the windows created by the function **make-editor-window**.

**delete-window** *window* [*Function*]

**Delete Window Hook** [*Variable*]

The function **delete-window** deletes the specified window after first invoking the hook **Delete Window Hook** with the given window as its argument.

**window-buffer** *window* [*Function*]

**Window Buffer Hook** [*Variable*]

The function **window-buffer** returns or sets (with **setf**) the buffer being displayed in the specified window. Before the displayed buffer is changed, the hook **Window Buffer Hook** is invoked with the window and the new buffer as arguments.

**window-display-start** *window* [*Function*]

Returns or sets (with **setf**) a mark that points to the position preceding the first character displayed in the specified window. If this mark changes, the window may be recentered to keep the current buffer's point within the displayed area.

**window-display-end** *window* [*Function*]

Returns a mark that points to the position following the last character displayed in the specified window.

**window-point** *window* [*Function*]

Returns or sets (with **setf**) a mark that points to the buffer location of the cursor for the specified window. The cursor location cannot be set for the current window because the cursor is located automatically at the current buffer point. Setting the cursor position for some other window sets the location to which the buffer point will be moved when that window becomes the current window.

**grow-window** *window n* [*Function*]

Tries to make the specified window larger by *n* lines (or smaller by -*n* lines if *n* is negative). This operation, which is constrained by the size of the screen, may make an adjacent window smaller or larger.

**center-window** *window mark* &optional *fraction* [*Function*]

Repositions the specified window so that the mark indicated is offset from the top by the fraction of the window height given by the *fraction* argument. If the *fraction* argument is omitted, the default value of 0.5 is used.

**scroll-window** *window n* [*Function*]

Scrolls the specified window down *n* lines in the buffer, or up -*n* lines if *n* is negative. The cursor remains at the same text position unless that position leaves the screen, in which case the cursor is centered within the text that is on the screen.

**displayed-p** *mark window* [*Function*]

Returns **t** if the specified mark is adjacent to a character that is currently displayed in the given window; otherwise it returns **nil**.

**window-height** *window* [*Function*]
**window-width** *window* [*Function*]

These functions return or set (with **setf**) either the height or the width of the window area in which the buffer text is displayed, in character positions. Setting such a value may fail.

**next-window** *window* [*Function*]
**previous-window** *window* [*Function*]

These functions return the window that either immediately precedes or immediately follows the specifed window.

**mark-to-cursorpos** *mark window* [*Function*]

Returns as multiple values the Cartesian coordinates of the specified mark in the window indicated or **nil** if the mark is not displayed. The *x* and *y* values returned are measured from the origin (0,0) in the upper left corner of the window in character positions and line positions respectively.

**cursorpos-to-mark** *x y window* [*Function*]

Returns as a mark the cursor position (*x*,*y*) within the specifed window; returns **nil** if that cursor position does not correspond to any text in the given window. The *x* and *y* values given are measured from the origin (0,0) in the upper left corner of the window in character positions and line positions respectively.

**mark-column** *mark* [*Function*]

Returns the position on the horizontal ($x$) axis where the specified mark would be displayed on an infinitely wide screen, while taking into account special characters such as tabs.

**move-to-column** *mark column* `&optional` *line* [*Function*]

Moves the specified mark to the position within the given line that corresponds to the column position *column*; the *line* argument defaults to the mark's current line. If the indicated line does not reach to the given column, the mark is unchanged and **nil** is returned.

**show-mark** *mark window time* [*Function*]

Highlights the position within the specified window of the given mark for *time* seconds; then returns **t**. If the mark does not appear within the given window, **nil** is returned.

**redisplay** [*Function*]

Causes the screen to be updated to reflect any changes to the text. The command interpreter calls this function after each command has been completed.

**redisplay-all** [*Function*]

Causes the whole screen to be redisplayed.

***echo-area-window*** [*Variable*]

***echo-area-buffer*** [*Variable*]

The echo area is a small window that appears at the bottom of the screen; prompting, echoing of input, and brief information reports appear in the echo area. The variable ***echo-area-window*** holds the echo area's window object, and ***echo-area-buffer*** holds the buffer displayed there, which is normally in the mode **Echo Area**. The echo-area window has no mode line.

**clear-echo-area** [*Function*]

Clears all text from the echo-area window. This is normally done by the command interpreter after each command has finished unless the **message** function has been used or **buffer-modified** has been set to **nil** for the ***echo-area-buffer***.

**message** *format-control-string* `&rest` *args* [*Function*]

Displays a message in the echo area, starting on a fresh line. The message displayed is generated from the **format** function arguments given. This is the recommended way of displaying messages in the echo area.

***echo-area-stream*** [*Variable*]

This buffered Editor output stream inserts its text at the current point in ***echo-area-buffer***. Because the stream is buffered, you should use the function **force-output** to assure that the text has been inserted.

## Prompting the User

Functions for prompting execute a recursive edit in the buffer **Echo Area**. Useful parsing commands such as name completion and help are provided by commands bound in the mode **Echo Area**.

**prompt-for-buffer** `&key :prompt :help :must-exist :default :default-string` *[Function]*

Prompts for a buffer name, provides name completion, and returns the selected buffer. The keyword arguments for this function and for the other prompting functions have the following meanings:

- **:prompt** specifies the string to be displayed as the prompt.
- **:help** specifies either a string to be displayed or a function to be called if the help command is typed to the current prompt. If the value is a function, it should take no arguments and return a help string to be displayed or **nil** after carrying out some helpful action.
- **:must-exist** specifies whether the user must give a response of the expected type. If **:must-exist** is non-**nil**, the user is reprompted after any response that is not of the expected type. If **:must-exist** is **nil** and the response is not of the expected type, the response given is returned as a string.
- **:default** specifies the default value to be returned if the user gives the null response. If no default is given, a null response from the user causes reprompting.
- **:default-string** specifies a string representing the default if **:default** is also given. The string is displayed after the prompt. If **:default** is given but **:default-string** is not, some representation of **:default** is displayed.

**prompt-for-character** `&key :prompt :change-window` *[Function]*

Prompts for a single character without waiting for confirmation. The macro **command-case** may be more useful. If the keyword **:change-window** is non-nil or omitted, the current window is changed to the echo-area window while the character is being read.

**prompt-for-key** `&key :prompt :help :must-exist :default :default-string` *[Function]*

Prompts for a key, which is a vector of characters that can be bound to a command. If the keyword **:must-exist** is non-**nil**, the key given must be bound in the current environment, in which case the command bound to the key is returned as a second value.

**prompt-for-file** `&key :prompt :help :must-exist :default :default-string` *[Function]*

Prompts for a valid filename and returns a pathname. If the keyword **:must-exist** is non-**nil**, the named file must exist. If the named file does not exist, the filename entered is merged with **:default** as if by the function **merge-pathnames**. In all cases, a pathname is returned.

**prompt-for-integer** `&key :prompt :help :must-exist :default :default-string` *[Function]*

Prompts for an optionally signed decimal integer.

**prompt-for-keyword** *string-tables* `&key :prompt :help :must-exist :default :default-string` *[Function]*

Prompts for a keyword and provides completion from the names in the *string-tables* argument, which is a list of string-tables. If the keyword **:must-exist** is non-**nil**, the user must give an unambiguous leading substring of one string name in the string-tables list. In this case, the entire string name is returned, and the value of the corresponding string-table entry is returned as a second value. If **:must-exist** is **nil**, the response string is returned exactly as given, although completion can be requested with the Editor commands **Complete Parse** and **Complete Field**.

**prompt-for-expression** `&key :prompt :help :must-exist :default :default-string` *[Function]*

Prompts for a Lisp expression. If the keyword **:must-exist** is **nil**, the string given is returned, even if a Lisp read error occurs.

**prompt-for-string** `&key :prompt :help :default :default-string` *[Function]*
Prompts for and returns a string. No checking is done on the string.

**prompt-for-variable** `&key :prompt :help :must-exist :default :default-string` *[Function]*

Prompts for the name of an Editor variable. If the keyword **:must-exist** is non-**nil**, the user must give the name of a variable that is defined in the current environment; the variable's symbol name is returned as a second value.

**prompt-for-y-or-n** `&key :prompt :help :must-exist :default :default-string` *[Function]*

Prompts for a single-character response of `y` or `n` (ignoring case) and returns **t** (for `y`) or **nil** (for `n`). No confirmation is required. Giving a **:default** value allows and specifies the result of typing just the confirmation character. If the keyword **:must-exist** is **nil**, the first character typed is returned if it is not `y` or `n`.

**prompt-for-yes-or-no** `&key :prompt :help :must-exist :default :default-string` *[Function]*

Prompts for `Yes` or `No` typed in full (ignoring case), plus confirmation. Giving a **:default** value allows and specifies the result of typing just the confirmation character. The value returned is **t** (for `Yes`) or **nil** (for `No`). If the response is neither `Yes` nor `No` and if the keyword **:must-exist** is **nil**, the string of text typed before the confirmation character is returned.

**command-case** ({*key value*}*) {(({(*tag*}*) |*tag*} *help* {*form*}*)}* [*Macro*]
Prompts for a single-character response and evaluates the forms of the first option that has a *tag* argument matching the given character. A tag can be either a logical character (such as **:help**) or a standard character that satisfies the predicate **standard-char-p**. The character read is compared to logical characters with **logical-char=** or to standard characters with **char=**, but the letter case is ignored. The **reprompt** macro within the body of an option restarts the prompting and response reading.

The logical characters **:help** and **:abort** have default options that can be replaced by explicit options. The default **:help** option displays a help message and then reprompts. This help message consists of the string *value* of any **:help** *key* argument, followed by all the *help* strings in the given options. The default **:abort** option signals an Editor error.

If **t** is specified in place of a *tag* list, a default option is created that has no *help* string and that is evaluated only if no other option is matched, including the default **:help** and **:abort** options. By default, the default option includes a beep and a reprompt.

The keywords that can appear as *key* arguments in the macro **command-case** and the uses of each corresponding *value* argument follow:

- **:prompt** specifies a string *value* for the prompt.
- **:help** specifies a string *value* to be displayed first by the default **:help** option.
- **:bind** binds the variable *value* to the response character during the evaluation of the selected option.
- **:character** causes the character *value* to be treated as the first response character, instead of reading a response from the user.
- **:change-window** specifies a flag *value* that determines whether the current window is to be changed to the echo-area window during the character-reading process. The current window is changed to the echo-area window if the flag is non-**nil** or if no **:change-window** keyword is specified. In some cases it may be useful not to change the current window so that the user can base the response on the current point within the current buffer.

## Parsing

Parsing is controlled by a character attribute, an Editor variable, and a collection of global Lisp variables.

The character attribute **Parse Field Separator** is used to decide if a given character is a field separator for the Editor command **Complete Field**. A character with a value of 1 for this attribute is a field separator.

**Beep On Ambiguity** [*Variable*]
Attempting to complete an ambiguous parse when this variable is non-**nil** causes a beep.

***parse-verification-function*** [*Variable*]
This function is called by **Confirm Parse** to do most of the parsing work. The function is called with one argument, namely the value of ***parse-input-region*** at the time **Confirm Parse** was called. The function returns a list of values as the result of the recursive edit or **nil** if the parse failed. A return of zero values can be indicated by returning a **nil** first value and a non-**nil** second value.

***parse-string-tables*** [*Variable*]
This is the list of string-tables used in the current parse.

***parse-value-must-exist*** [*Variable*]
This contains the value of the **:must-exist** argument.

***parse-default*** [*Variable*]
This is the default value for the parse.

***parse-default-string*** [*Variable*]
This is the string representation of the variable ***parse-default***. If **Confirm Parse** is called when ***parse-input-region*** is empty, the value of the variable ***parse-default-string*** is given to the function ***parse-verification-function***.

***parse-prompt*** [*Variable*]
This is the current prompt string.

***parse-help*** [*Variable*]
This is the current help string or function.

***parse-starting-mark*** [*Variable*]
This is a mark in the echo-area buffer where the parse started.

***parse-input-region*** [*Variable*]
This is a region starting at ***parse-starting-mark*** and extending to the end of the echo-area buffer. The text within this region is parsed by **Confirm Parse**.

## Editor I/O

**beep** *[Function]*
Attracts the user's attention by sounding a beep or flashing the screen.

***last-character-typed*** *[Variable]*
This is the last character typed by the user.

***editor-input*** *[Variable]*
Holds an input stream that reads characters from the keyboard without echoing them.

**text-character** *character* *[Function]*
Turns the specified keyboard character into a character that can be placed into a buffer.

***input-transcript*** *[Variable]*
If non-nil, this should be a vector with a fill pointer, in which case all input read is recorded by pushing it onto this vector.

**make-editor-output-stream** *mark* `&optional` *buffered* *[Function]*
Creates an Editor output stream whose characters get inserted at the specified permanent mark. The argument *buffered* creates the following streams:

- **:none** makes an unbuffered stream; this is the default.
- **:line** makes a buffered stream that is forced out whenever a newline character appears (or whenever an explicit **force-output** is done).
- **:full** makes a buffered stream that is never forced out except when an explicit **force-output** is done.

**make-editor-region-stream** *region* *[Function]*
Returns an input stream from which the text in the specified region can be read.

**with-input-from-region** (*var region*) {*declaration*}* {*form*}* *[Macro]*
Evaluates the given forms with the variable *var* bound to an input stream that provides the text from the specified region.

**with-output-to-mark** (*var mark* [*buffered*]) {*declaration*}* {*form*}* *[Macro]*
Evaluates the given forms with the variable *var* bound to an output stream that inserts its text at the specified permanent mark. The argument *buffered* has the same meaning as in **make-editor-output-stream**.

**with-random-typeout** (*var n*) {*declaration*}* {*form*}* *[Macro]*
Evaluates the given forms with the variable *var* bound to an output stream that displays its text on the screen in some nice form. The argument *n* is the approximate number of lines the output will require.

## Handling Errors

**editor-error** `&rest` *args* [*Function*]

Signals a minor Editor error (such as an unsuccessful search), aborts the command in progress, and never returns. If there are no arguments, a **beep** occurs. Any arguments given are taken as arguments to the function **format** for generating an error message that is displayed.

**catch-editor-error** {*form*}* [*Macro*]

Evaluates the given forms and traps any Editor error that occurs within them. When an Editor error occurs, evaluation is stopped and **nil** is returned. If no Editor error occurs, the value returned is that of the last form.

## Reading and Writing Files

**buffer-write-date** *buffer* [*Function*]

Returns in Universal Time format the time and date when the specified buffer was last written out to a file.

**read-file** *pathname mark* [*Function*]

Inserts the text from the file specified by the *pathname* argument at the mark indicated.

**write-file** *pathname region* [*Function*]

Writes the text from the region indicated to the file specified by the *pathname* argument.

## Exiting the Editor

**exit-editor** `&optional` *value* [*Function*]

**Exit Hook** [*Variable*]

The function **exit-editor** exits the Editor and returns the optional value, which defaults to `t`. Before the Editor is exited, the hook **Exit Hook** is invoked.

D

# Getting Fast Code from the Sun Common Lisp Compiler

sun
microsystems

D

# Getting Fast Code from the Sun Common Lisp Compiler

This appendix explains how to get optimized code from the Sun Common Lisp compiler. Before reading this appendix, read the chapter on the compiler earlier in this book. This appendix assumes that you understand declarations. If necessary, see Chapter 9 of *COMMON LISP, The Language*, by Guy L. Steele Jr. for more information on declarations.

## D.1. Compiler Optimization Settings

The `OPTIMIZE` declaration specifier influences the kind of code that the compiler emits. The default values in Sun Common Lisp permit the compiler to generate fast code.

## D.2. Speedy Operations

Some operations are fast with or without `TYPE` declarations. These fast operations include:

- References to variables except "closed-over" lexical variables
- Entry and exit of scopes that introduce no special variables
- `if`, `cond`, `loop`, `do`, `go`, `return`, and others (local flow of control)
- `and`, `or`, `not`
- Most type testing (`typep`, `consp`, `symbolp`, `null`, and others)
- `svref`, `schar`
- `eq`
- `setq` except "closed-over" lexical variables

The Sun Common Lisp compiler generates fast code for these operations with the default optimization settings.

Somewhat slower operations include:

- Entry and exit of blocks that bind special variables
- Entry and exit of blocks whose names are "closed-over"
- Access to "closed-over" lexical variables
- Creation of closures (functions that "close over" one or more lexical variables)

## D.3. Primitive Operations on Speedy Types

In Common Lisp, some types of objects are simple enough that a compiler can generate good in-line code to operate on them. In particular, the compiler can generate particularly fast code for the following common types:

- `cons` (and `list`)
- `fixnum`
- `char` including `string-char`
- Simple general vectors (elements of type `T`) and simple strings
- Simple general arrays (elements of type `T`), especially of known dimensions
- Structures

In general, creating an object takes quite a bit longer than doing simple operations on it, perhaps an order of magnitude more time.

To get fast code from the compiler, let it know enough about the types of inputs or outputs so that it can dispense with overhead. You can eliminate four kinds of overhead by requests to the compiler. They are:

- Type dispatching on inputs
- Type dispatching on outputs (for fixnum operations)
- Checking number of arguments (somewhat less important)
- Procedure call (somewhat less important)

For these types, primitive operations can be exceptionally fast. Overhead tends to be a high proportion of the time spent doing simple operations on objects of these types. When you make declarations, the compiler assumes that the declarations are true and eliminates that overhead.

## D.4. Making Declarations

To make optimizing declarations, make sure to tell the compiler the types of things.

### Fixnums

Fixnums are "fast integers", currently integers in the range from about -500 million to +500 million. Declare inputs to simple arithmetic operations and numeric predicates as fixnums for fast code. For the basic arithmetic operations, also declare the output to be a fixnum.

### Vectors and Arrays

The "fastest arrays" are "simple arrays." Currently arrays of "general" type, type `T`, are also faster than all other alternatives. Use `svref` for accesses to simple vectors, or for 2-dimensional arrays use code such as this:

```
(proclaim '(type (simple-array t (512 512)) tab))
(setq tab (make-array '(512 512)))

(defun f (a i j)
  (declare (type (simple-array t (512 512)) a))
  (declare (fixnum i j))
  (aref a i j))
```

## D.5. Types of Overhead

There are three types of overhead described below:

- Type Dispatching
- Checking Number of Arguments
- Procedure Calls

### Type Dispatching

Type dispatching includes checking that the types of arguments are legitimate for the operation applied to them. It also includes determining how to do the operation to the particular things passed to the function. For example, fetching an element from a sequence is done differently for a list, than for a simple vector, and fetching from a non-simple vector is different again.

### Checking Number of Arguments

For ordinary function calls (not via `FUNCALL` or `APPLY`), the compiler can check the number of arguments passed to functions at compile-time. When the compiler generates in-line code, it does not generate code to check the number of arguments. Some short functions are called as subroutines. For these, declaring `:fast-entry T` makes the compiler generate calls on a version of the routine that does not check the number of arguments passed to it.

The form of the `:fast-entry T` request is:

```
(eval-when (compile)
  (compiler-options :fast-entry t))
```

### Procedure Call

When compiled code calls a procedure, there is overhead for the subroutine call and return. Usually, registers are also saved and restored, arguments are pushed onto the stack, etc. If a procedure call is not executed ("in-line code"), these overheads are avoided.

## D.6. How to Tell the Compiler

How do you see to it that the compiler knows the types of expressions (inputs or outputs)? It knows from declarations you make. The simplest form of declaration is the `the` declaration. `the` declares information about an individual expression. Other declarations can be understood as implicitly putting `the` declarations around a set of expressions.

Making a declaration about a variable implicitly declares every reference to the variable and every expression assigned to the variable by `setq`, `psetq`, and others.

Making a declaration about a function implicitly declares all the inputs and ouptuts of all calls of that function that the compiler sees.

## D.7. Specific Fast Operations, By Type

- `CONS` and `LIST`

  `CAR, CDR, RPLACA, RPLACD` are fast, in-line.

- `FIXNUM`

  The simple arithmetic operations on fixnums (mainly `+, -, <, >, =, >=, <=, /=`) are fast and in-line. For fixnum operations there is normally a check of the size of the result even when the inputs are fixnums. If the result is too large to be a fixnum, an extended-precision integer is created. If the output is declared to be a fixnum, no check of the output is made. It is simply assumed to be a fixnum.

- `CHAR` including `STRING-CHAR`

  Comparisons and conversion to and from integer are fast.

- Simple Arrays (including simple general vectors and simple strings)

  It is important for the compiler to know that the array is in fact a vector and the vector is simple. It is also important for it to know what types of elements the vector can contain. `SVREF` automatically assumes a simple `GENERAL` vector. `SCHAR` implicitly takes a simple string. `SBIT` implicitly takes a simple bit vector.

  There are other sorts of simple vectors as well. Uses of them should be declared. Access to arrays of type fixnum is currently not as fast as access to arrays of type `T` (general arrays).

  for vectors known to be simple declared to have elements of type `T`, and indices known to be fixnums, `AREF`, `(SETF (AREF . . . ))` are fast and in-line.

  Access to multidimensional simple arrays (2D at least) is also fast and in-line with appropriate declarations as shown above.

- Arrays

  The functions `LENGTH` and `ARRAY-DIMENSION` are fast and in-line. Most other accessors of array attributes are fast.

- Structures

  Structure accessors including slot setting functions are fast and in-line. They are implicitly declared to assume that the structure argument is a structure of the appropriate type. You do not need to make declarations to get fast code for these operations.

## D.8. Additional Topics

### Tail Recursion

For those of you who understand what a tail call is (either to self or to some other function), be it known that the compiler removes tail recursions, sometimes saving greatly on stack usage.

### `LABELS` and `FLET`

Calls to functions defined by `FLET` or `LABELS` are much faster than calls on other functions. In fact, the compiler can sometimes completely eliminate the function call.

# Index

Systems for Open Computing™

**Corporate Headquarters**
Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, CA 94043
415 960-1300
TLX 37-29639

**For U.S. Sales Office locations, call:**
800 821-4643
In CA: 800 821-4642

**European Headquarters**
Sun Microsystems Europe, Inc.
Bagshot Manor, Green Lane
Bagshot, Surrey GU19 5NL
England
0276 51440
TLX 859017

**Australia:** (02) 413 2666
**Canada:** 416 477-6745
**France:** (1) 40 94 80 00
**Germany:** (089) 95094-0
**Hong Kong:** 852 5-8651688
**Italy:** (39) 6056337
**Japan:** (03) 221-7021
**Korea:** 2-7802255
**Nordic Countries:** +46 (0)8 7647810
**PRC:** 1-8315568
**Singapore:** 224 3388
**Spain:** (1) 2532003
**Switzerland:** (1) 8289555
**The Netherlands:** 02155 24888
**Taiwan:** 2-7213257
**UK:** 0276 62111

**Europe, Middle East, and Africa, call European Headquarters:**
0276 51440

**Elsewhere in the world, call Corporate Headquarters:**
415 960-1300
Intercontinental Sales