sun microsystems

Sun™ Common Lisp Reference Guide

Part Number 800-1518-10
Revision: A February 2, 1987

Credits and Trademarks

Sun Workstation is a registered trademark of Sun Microsystems, Inc.

SunStation, Sun Microsystems, SunCore, SunWindows, SunView, DVMA, and the combination of Sun with a numeric suffix are trademarks of Sun Microsystems, Inc.

UNIX, UNIX/32V, UNIX System III, and UNIX System V are trademarks of AT&T Information Systems, Inc.

Intel and Multibus are registered trademarks of Intel Corporation.

DEC, PDP, VT, and VAX are registered trademarks of Digital Equipment Corporation.

About This Book

This book presents a complete technical description of Sun Common Lisp. It is designed as a reference tool. Programmers who use this book should have some knowledge of general Lisp programming concepts. This book is not intended to be a tutorial on Common Lisp. Rather, it is a comprehensive description and specification of the Common Lisp language and extensions to Common Lisp by Lucid, Inc.

Organization of This Book

The *Sun Common Lisp Reference Manual* has twenty-four chapters and two appendixes.

- Chapter 1. "Introduction" contains a brief overview of Common Lisp. It also describes the notational conventions used throughout this book.
- Chapter 2. "Data Types" introduces the data types provided by Common Lisp.
- Chapter 3. "Type Specifiers" describes the use of type specifiers in designating types.
- Chapter 4. "Program Structure" describes the organization of Common Lisp programs in terms of forms and functions.
- Chapter 5. "Control Structure" describes the constructs available for controlling the flow of program execution and evaluation.
- Chapter 6. "Macros" describes the use of macros and the macro text replacement facility.
- Chapter 7. "The Evaluator" discusses the evaluation of Common Lisp programs.
- Chapter 8. "Declarations" describes the use of declarations in tailoring a program to the needs of the user and the system.
- Chapter 9. "Predicates" describes the use of predicate functions and logical operations.
- Chapter 10. "Symbols" describes the use of symbol data objects.
- Chapter 11. "Packages" describes the use of packages in organizing the program name space.
- Chapter 12. "Numbers" describes the numerical data types and operations on numbers.
- Chapter 13. "Characters" describes the character data type and operations on characters.
- Chapter 14. "Sequences" describes the sequence data type and operations on sequences.
- Chapter 15. "Lists" describes the list data type and operations on lists.
- Chapter 16. "Arrays" describes the array data types and operations on arrays and vectors.
- Chapter 17. "Strings" describes the string data type and operations on strings.
- Chapter 18. "Hash Tables" describes the hash table data type and operations on hash tables.
- Chapter 19. "Structures" describes the creation of user-defined data types and the operations upon them.
- Chapter 20. "Streams" describes the use of streams in program input and output operations.

- Chapter 21. "Input/Output" describes the reading and printing operations of Common Lisp, including formatting options.
- Chapter 22. "File System Interface" describes the facilities for accessing files and communicating with the file system.
- Chapter 23. "Errors" describes error-signaling operations.
- Chapter 24. "Environmental Features" briefly describes facilities for compilation, debugging, documentation, and other functions that interface with the environment. For a complete technical description of Sun Common Lisp, the user is referred to the *Sun Common Lisp User's Guide.*
- Appendix A. "Alphabetical Listing of Common Lisp Functions" is a list of all Common Lisp functions, macros, constants, variables, and special forms, including all extensions to Common Lisp described in this manual.
- Appendix B. "Extensions to Common Lisp" lists the extensions to Common Lisp described in this manual.

Related Publications

The following books contain related information that the user may find helpful.

Sun Common Lisp User's Guide is a guide to using the special features and functions of Sun Common Lisp.

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

Chapter 11. Packages 11–1

Chapter 12. Numbers 12–1

Chapter 14. Sequences 14–1

Chapter 16. Arrays 16–1

Chapter 17. Strings 17–1

Chapter 18. Hash Tables 18–1

Chapter 19. Structures 19–1

Chapter 20. Streams 20–1

Chapter 21. Input/Output 21–1

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 the user types an expression at the terminal, Lisp evaluates it and displays the result automatically. Most 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 for the user by providing a dynamic heap of storage that is automatically allocated as needed and then reclaimed, or garbage collected, when no longer needed.

The process of compiling and debugging programs is discussed at length in the *Sun Common Lisp User's Guide*.

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 the user has 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. Users can write programs in either uppercase or lowercase, or a combination of the two, whichever is preferred.

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 the user is 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. This syntax and the Common Lisp data types are introduced in the following chapter.

Chapter 2. Data Types

Chapter 2. Data Types

About Data Types

A data type is a set of objects that satisfy certain criteria or possess certain properties. Unlike the data types of many programming languages, the data types of Common Lisp are properties of objects rather than of variables. Types are associated with the objects to which variables are bound, not with the variables themselves.

Common Lisp data types form a type hierarchy. An object may belong to more than one such set, and hence to more than one data type. For example, a string of characters is also a vector and therefore an array; since the vector data type is a subtype of sequence, a character string is also a sequence. The type **t** is a supertype of all other types and a proper subtype of none; it contains all objects. The type **nil** is a subtype of all other types and a proper supertype of none. It represents the empty type. There are no objects of type **nil**. The type **t** should not be confused with **t**, the Lisp object; similarly, the type **nil** should not be confused with the object **nil**. The ***common*** data type is a supertype that contains all of the objects required by the Common Lisp language.

The functions **typep** and **type-of** may be used to determine the type of a particular object. The predicate **typep** indicates whether an object belongs to a particular type. The function **type-of** returns one of the types to which the object belongs. Common Lisp provides numerous data type predicates to test objects for membership in particular types.

The most common and useful Common Lisp data types are introduced below. Figure 2–1 shows the hierarchical relationship of these types. Associated with each data type is a set of operations for creating and manipulating objects of that type. The user is referred to individual chapters of this manual for a more detailed discussion.

Numbers

Integers, ratios, floating-point numbers, and complex numbers are provided as separate data types. Integers and ratios together constitute a subtype of numbers called ***rational numbers***. Numbers and numerical operations are discussed in Chapter 12.

Integers

The ***integer*** data type consists of fixnums and bignums. The ***fixnum*** data type is designed to allow integers in the range from **most-negative-fixnum** to **most-positive-fixnum** to be represented efficiently, using a fixed number of bits. The fixnum data type is the default for the representation of integers. The ***bignum*** data type is provided to allow for the representation of integers of arbitrary magnitude. The distinction between fixnums and bignums is generally not visible to the user. In Sun Common Lisp, the more appropriate representation is used automatically.

Ratios

Ratios give an exact representation of the mathematical quotient of two integers. Ratios can be used to avoid the loss of precision that can result from using floating-point numbers.

Rational numbers are represented in ***canonical form***. If the ratio is not an integer, the canonical representation is a pair of integers, the numerator and denominator, that represent the rational as a fraction in reduced form. The denominator is always positive. If the denominator evenly divides the numerator, the rational number is converted to the resulting integer.

Floating-Point Numbers

Floating-point numbers constitute the type ***float***. Four floating-point number formats are provided: ***short-float***, ***single-float***, ***double-float***, and ***long-float***. These formats differ in the precision they provide and in the range of exponents they allow. Sun Common Lisp represents all four types of floating-point numbers in the single-float format.

When an operation involves both a rational and a floating-point argument, the rational number is first converted to floating-point format, and then the operation is performed. This conversion process is called ***floating-point contagion***.

Complex Numbers

Complex numbers are represented as composite objects consisting of a real part and an imaginary part. The two parts of a complex number must be of the same noncomplex type; if they are not, they are automatically converted to the same type, in accordance with the principle of floating-point contagion. Complex numbers are represented in ***canonical form***. If a complex number whose components are of type integer or ratio has an imaginary part whose value is zero, the canonical representation is an integer or ratio whose value is the same as that of the real part.

Characters

Characters in Common Lisp are data objects that represent printed symbols, such as letters, or operations for formatting text. Each character has three attributes: code, bits, and font.

Common Lisp defines a standard character set as a subtype of characters called ***standard characters***. The standard character set consists of 95 printing characters and the newline character. The font and bits attributes of all standard characters are zero.

String characters are a subtype of characters that can be contained in strings. Strings are vectors of characters. A string character is any character whose bits and font attributes are zero. The standard character data type is thus a subtype of the string character data type, and all of the standard characters can be stored in strings.

The character data type is discussed in Chapter 13.

Symbols

Symbols are data objects with five components: a print name, a value cell, a function cell, a property list, and a package cell.

Symbols are named data objects. The ***print name*** of a symbol is a string that is used to identify and locate the symbol. Symbols are organized into name spaces called ***packages***. Symbol names are unique within a package.

The ***value cell*** is the cell that holds the current value of the dynamic variable named by the symbol. A value may be associated with this cell by assignment functions or by constructs that establish new variable bindings.

The ***function cell*** contains the global function definition associated with the symbol. A function object may be associated with the function cell through the various function definition constructs.

A ***property list*** allows an extensible set of named components to be associated with a symbol. A component may be any Lisp object. Each successive two elements of the property list constitute an entry. The first element of an entry is the ***indicator***, or property name, and the second element is the property value. When a symbol is created, its property list is empty.

The ***package cell*** refers to a package object. A ***package*** is a catalogue containing an index of print names. It is used to locate a symbol.

An important use of symbols is to name other objects, that is, to serve as variables. Symbols are discussed in Chapter 10.

Packages

A ***package*** is a Common Lisp object that specifies a correspondence between print name strings and symbols. The package facility may be used to create a hierarchical program name space and to increase program modularity. Packages enable the user to avoid name conflicts that may arise when separate modules become part of the same system. Packages are discussed in Chapter 11.

Sequences

Sequences are ordered sets of elements and include both lists and vectors (one-dimensional arrays). Operations on sequences are provided as general operations that are relevant for both of these types. The sequence data type is discussed in Chapter 14.

Lists

Lists are sequences of linked elements, called ***conses*** (dotted pairs). The list data type consists of the data types ***cons*** and ***null***. The empty list, **nil**, is the only list object of the type null. The type null should not be confused with the predicate **null**. The list data type includes both true lists and dotted lists.

A ***cons*** is an object containing two components, a ***car*** and a ***cdr***, which can be any Lisp objects. Conses in a list are linked by their cdr components. The car components become the elements of the list. An ordinary, or true, list is terminated by **nil**, the empty list. A ***dotted list*** is terminated by some non-**nil** data object.

An ***association list*** is a list whose elements are conses. Each cons is regarded as a pair of associated objects. The car is called the ***key*** and the cdr the ***datum***. An association list can be treated as a mapping from keys to data.

The list data type is discussed in Chapter 15.

Arrays

Arrays are structured objects whose components can be directly accessed by means of index values. An array can have many dimensions. It is indexed by a sequence of integers called subscripts. Arrays can share their contents with other arrays and have their size altered dynamically. Arrays may be general or specialized. A ***general array*** can have elements that are members of any Common Lisp data type. A ***specialized array*** is an array whose elements must all be members of a particular data type.

A ***vector*** is a one-dimensional array. Since the vector data type is a subtype of the sequence data type, a vector is also a sequence. A ***general vector*** can have elements that are members of any Common Lisp data type. A ***specialized vector*** is a vector whose elements must all be members of a particular data type. ***Strings*** and ***bit vectors*** are important types of specialized vectors. Strings are vectors whose elements are of the string character data type. Bit vectors are one-dimensional arrays whose elements are of the bit data type. A vector can have a fill pointer. A ***fill pointer*** is an index that is used to incrementally fill in the elements of the vector and thus vary the length of the active portion.

A ***simple array*** is an array that does not share cells with another array, has no fill pointer, and whose size cannot be dynamically adjusted. A ***simple vector*** is a vector that is not displaced to another array, has no fill pointer, and whose size cannot be dynamically adjusted.

Arrays are discussed in Chapter 16.

Strings

Strings are specialized vectors of characters. The string type is identical to the type `(vector string-char)`. Like all vectors, strings may have fill pointers. Strings are discussed in Chapter 17.

Hash Tables

Hash tables are Common Lisp objects that provide mappings between other objects. Each hash table entry is a pair of associated objects, a key and a value. Hash table functions use keys to look up their associated values. Common Lisp provides hash table functions to add entries, delete entries, and look up the values associated with given keys. Chapter 18 discusses the use of hash tables and hashing functions.

Structures

Common Lisp allows the user to create record ***structures*** with a fixed number of named components. These structures are, in effect, user-defined data types. When these data types are defined, constructs to manipulate them are normally automatically defined by the system as well. These constructs include type predicates and access, constructor, and copier functions. Structures are created with the **defstruct** macro.

The definition of structures and the creation and manipulation of structure instances are discussed in Chapter 19.

Readtables

A ***readtable*** is a data object that is used to guide the action of the Lisp reader. It contains information about the syntax of Lisp characters that is used in parsing. Readtables are discussed in Chapter 21.

Streams

Streams are Common Lisp objects from which data can be read and to which data can be sent. Normally, the system reads characters from a character input stream, parses these characters as Lisp forms, evaluates each form as it is read, and prints representations of the results of the evaluation to a character output stream. The operations that can be performed on a stream depend on what type of stream it is. A stream may be input-only, output-only, or bidirectional. It may be a character stream or a binary stream.

There are several stream-value variables that are used by default by many Common Lisp system functions. These are known as ***standard streams***.

The use of streams is closely connected to the file system. Streams may also be created through the file system constructs for opening files.

Streams are discussed in Chapter 20. Chapter 21 discusses the use of streams in the context of the input/output system. The interaction between streams and the file system is discussed in Chapter 22.

Pathnames

Pathnames are objects that are used to represent file names in a way that is general enough to accommodate a diverse range of file system implementations. Pathnames have six components: host, device, directory, file name, type, and version. Pathnames and the file system interface are discussed in Chapter 22.

Random States

Random state objects are used to represent the internal state of the random number generator. They are manipulated by the random number generation facility. Random states are discussed in Chapter 12.

Functions

Functions are executable objects that may be applied to arguments to produce values. Functions in Common Lisp may be named or unnamed. Functions are discussed in Chapter 4.

Relationships Among Common Lisp Data Types

Figure 2-1 shows the relationships among the Common Lisp data types. An arrow from one data type to another indicates that the data type on the left of the arrow is a subtype of the data type on the right. Operations for testing the relationship between two types are discussed in Chapter 3.

Hierarchy of Data Types

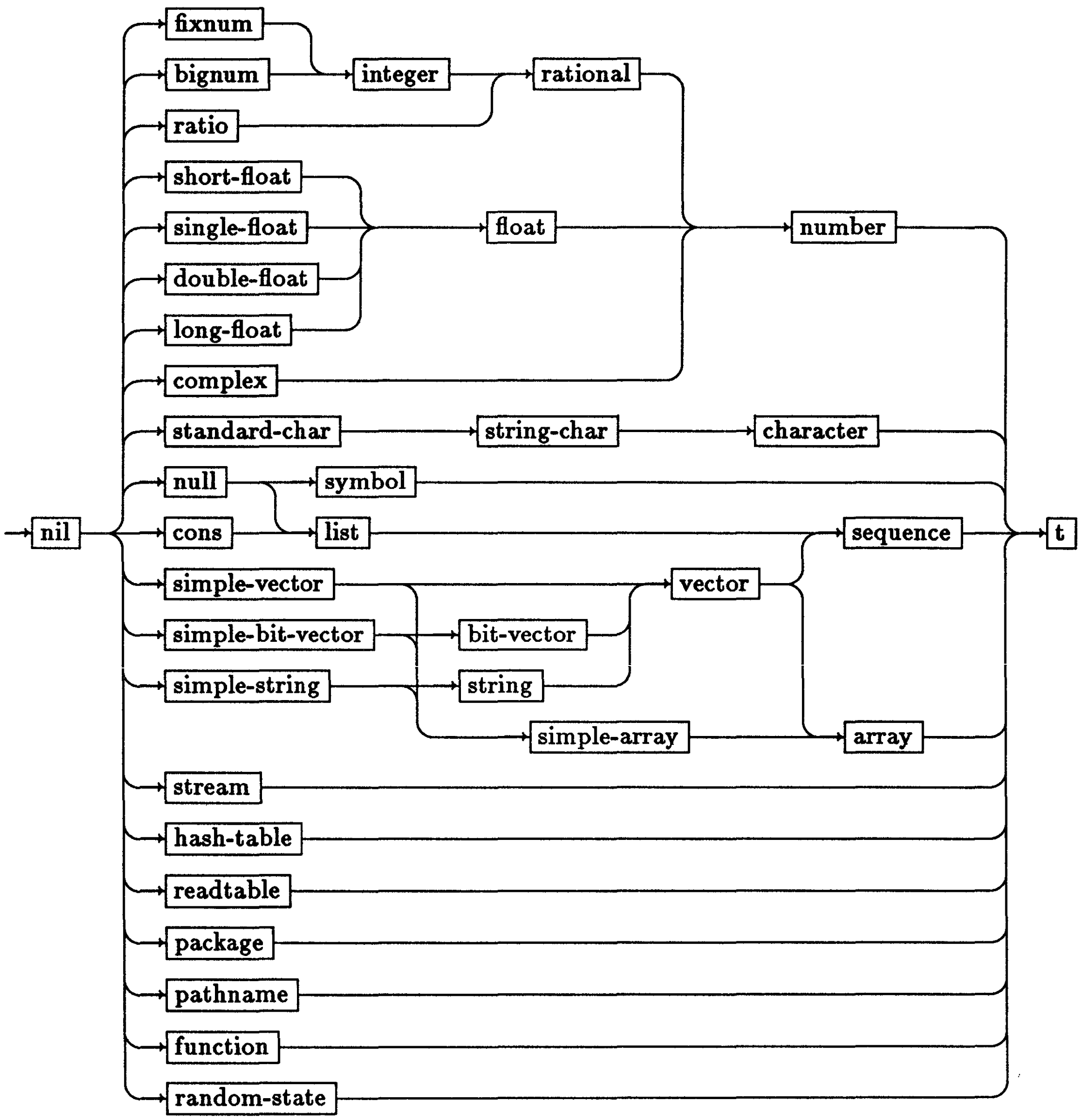

Figure 2–1. Relationships among the Common Lisp data types

Printed Representations of Data Types

In Common Lisp, each data type has its own printed (displayed) representation. This section provides a brief and partial overview of the most common formats that occur in the examples in the following chapters. For a detailed discussion of the printed representation of data types, the user is referred to Chapter 21.

Integers

An integer is printed as a sequence of digits in a particular base, or radix. For the decimal base, the radix indicator is a decimal point following the number. For other bases, the radix indicator is one of the following forms preceding the number: `#o` (octal), `#x` (hexadecimal), `#b` (binary), or `#nr` (other base *n*; the base *n* is printed in decimal).

Ratios

Ratios are always printed in lowest reduced form, with the numerator printed, then a slash (/), and then the denominator. In a negative ratio, the numerator is preceded by a minus sign.

Floating-Point Numbers

Floating-point numbers are printed as one or more digits on each side of a decimal point, sometimes followed by an exponent marker. If the number is negative, it is preceded by a minus sign.

Complex Numbers

A complex number is printed as `#C(`*r i*`)`, where *r* is the printed representation of the number's real part and *i* is the printed representation of the number's imaginary part.

Characters

A character is printed as `#\` followed by the character, if it is a printing character, or by the name of the character, if it is not.

Symbols

A symbol is printed as its print name along with any character quoting or name qualification necessary to identify the symbol uniquely. This may include backslashes (\), vertical bars (|), a colon (:), a package name and one or two colons (:), or a leading #: (for uninterned symbols). If the print name could be interpreted as a potential number, then backslashes or vertical bars are included to prevent such interpretation.

If the symbol is in the keyword package, it is printed with a leading colon. If the symbol is not accessible in the current package, it is printed with a leading package name and one or two colons. A leading `#:` is printed if the symbol is uninterned (has no home package).

Lists

A true list is printed as follows: first a left parenthesis, then the elements of the list in order, and finally a right parenthesis. The list elements are separated by white space (space, tab, carriage-return, or newline characters).

A dotted list is printed as follows: first a left parenthesis, then the car of the list, a dot, the cdr of the list, and finally a right parenthesis. The dot is separated from the car and the cdr of the list by white space.

Conses are printed with list notation rather than dot notation whenever possible.

Arrays

An array is printed with the `#nA(...)` syntax. In this case, the output starts with `#nA`, where *n* is the number of dimensions of the array, and then the contents of the array are printed in row-major order with parentheses indicating the structure of the array. The length of the top-level list printed is the size of the first dimension, and the lengths of the subsequent deeper levels are the sizes of the second dimension, the third dimension, and so on.

If the array has elements that are either bits or string characters, then the deepest level printed may take the form of a bit vector or string.

Vectors

A vector is printed as `#(` and `)` enclosing the elements of the vector, which are separated by white space. For a vector with a fill pointer, only those elements before the fill pointer are printed.

Bit Vectors

A printed bit vector consists of `#*` followed by the bits in the bit vector. For a bit vector with a fill pointer, only those bits before the fill pointer are printed.

Strings

A string is preceded and followed by a double quote (`"`), and any double-quote or single escape character in the string is preceded by a backslash (\). A string with a fill pointer is printed only up to the fill pointer.

Structures

A structure is printed as `#S` immediately followed by a list in the form `(`*name slot1 value1 slot2 value2* ... `)`, where *name* is the name of the structure, *slot1* is the name of one of the structure's slots, and *value1* is the corresponding value.

Pathnames

A printed pathname consists of `#P` followed immediately by the pathname enclosed in double quotes.

Random States

An object of type random state is printed like a structure, with the `#S` syntax.

Other Data Types

An object that is a hash table, a readtable, a package, a stream, or a function object is printed with the `#<...>` syntax. This form describes the data type and may give some indication of the particular instance (such as a memory address where it appears).

Chapter 3. Type Specifiers

Chapter 3. Type Specifiers

About Type Specifiers

Common Lisp objects called ***type specifiers*** are used to designate types. Type specifiers can be atomic type specifiers or lists. Type specifier lists designate specialized types in terms of simpler types. The user may define new atomic type specifiers in terms of existing types and type specifier lists.

New type specifier identifiers are defined by means of the **deftype** special form and the **defstruct** macro. The **deftype** special form can be used to define a new type specifier name in terms of existing type specifiers. Creating a new structure with the **defstruct** macro automatically creates a new type specifier identifier that designates instances of the structure type.

Type specifiers are used in declarations and as arguments to many functions that construct new objects.

The predicate **typep** uses type specifiers for type discrimination. Only objects that are actually members of the given type satisfy the predicate.

Atomic System-Defined Type Specifiers

array	**integer**	**short-float**
atom	**keyword**	**signed-byte**
bignum	**list**	**simple-array**
bit	**long-float**	**simple-bit-vector**
bit-vector	**mod**	**simple-string**
character	**nil**	**simple-vector**
common	**null**	**single-float**
compiled-function	**number**	**standard-char**
complex	**package**	**stream**
cons	**pathname**	**string**
double-float	**random-state**	**string-char**
fixnum	**ratio**	**symbol**
float	**rational**	**t**
function	**readtable**	**unsigned-byte**
hash-table	**sequence**	**vector**

Figure 3–1. Table of Atomic Type Specifiers

Syntax for Type Specifiers

```
typespec::= atomic-type-specifier
          | (satisfies predicate-name)
          | (member {object}*)
          | (not typespec)
          | (and {typespec}*)
          | (or {typespec}*)
          | (array [{typespec | * } [dimensions]])
          | (simple-array [{typespec | * } [dimensions]])
          | (vector [{typespec | * } [{size | *}]])
          | (simple-vector [size])
          | (string [size | *])
          | (simple-string [size | *])
          | (bit-vector [size | *])
          | (simple-bit-vector [size | *])
          | (integer [integer-limit [integer-limit]])
          | (fixnum [fixnum-limit [fixnum-limit]])
          | (mod [integer | *])
          | (signed-byte [size | *])
          | (unsigned-byte [size | *])
          | (rational [rational-limit [rational-limit]])
          | (float [float-limit [float-limit]])
          | (short-float [short-float-limit [short-float-limit]])
          | (single-float [single-float-limit [single-float-limit]])
          | (double-float [double-float-limit [double-float-limit]])
          | (long-float [long-float-limit [long-float-limit]])
          | (complex [typespec | *])
          | (function [arg-typespec-list [value-typespec]])

arg-typespec-list::= ({typespec}* [&optional typespec] [&rest typespec]
                      [&key  {typespec}*])

value-typespec::= typespec | (values . arg-typespec-list)

dimensions::= integer | * | ({integer | * }*)

size::= integer
integer-limit::= integer | * | (integer)
fixnum-limit::= fixnum | * | (fixnum)
```

rational-limit::= *rational* | * | (*rational*)
float-limit::= *float* | * | (*float*)
short-float-limit::= *short-float* | * | (*short-float*)
single-float-limit::= *single-float* | * | (*single-float*)
double-float-limit::= *double-float* | * | (*double-float*)
long-float-limit::= *long-float* | * | (*long-float*)

Type Specifier Lists

A type specifier may be defined to denote the set of all objects that satisfy a particular predicate by use of the construct `(satisfies` *predicate-name*`)`, where the symbol *predicate-name* has a global function definition as a predicate of one argument.

A type specifier may be defined to denote the set of all objects that are members of a certain set by use of the `(member` {*object*}*`)` construct. The objects in this set are precisely those given in the list.

Other type specifier lists define combinations or specializations of existing type specifiers.

Specializations of atomic type specifiers indicate that only a specific subset of the objects that satisfy the atomic type specifier is designated. Use of such type specifiers may enable the system to represent or access objects more efficiently.

Many of these lists allow arguments to be unspecified. An unspecified argument is denoted by `*`. Unspecified arguments occurring at the end of a type specifier list may be omitted entirely. If all arguments are omitted, the type specifier name itself may be used (instead of a list).

Logical Combinations of Type Specifiers

The logical operators **and**, **or**, and **not** may be used to define type specifiers as logical combinations of other type specifiers.

The type specifier `(not` *typespec*`)` denotes the set of all objects that are not of the specified type.

The type specifier `(and` {*typespec*}*`)` denotes the set of all objects that are members of all of the specified types.

The type specifier `(or` {*typespec*}*`)` denotes the set of all objects that are members of at least one of the specified types.

Type Specifiers for Array Subtypes

There are several ways of specifying subtypes of arrays.

The type specifier (`array` *typespec dimensions*) denotes the set of arrays that have the given dimensions and whose elements are of the specified type. The *dimensions* argument may be either an integer or a list. If the *dimensions* argument is a nonnegative integer, it indicates the number of dimensions of the array. If it is a list, the number of elements implicitly indicates the number of dimensions of the array; the elements of the list indicate the length of each dimension. Any of the arguments may be unspecified.

The type specifier (`simple-array` *typespec dimensions*) is identical to (`array` *typespec dimensions*) except that it designates a set of simple arrays. A simple array is an array that is not displaced to another array, that has no fill pointer, and whose size cannot be dynamically adjusted.

The type specifier (`vector` *element-type size*) designates the set of one-dimensional arrays whose elements are of the specified type and whose lengths are of the given size. The *size* argument is a nonnegative integer or is unspecified.

The type specifier (`simple-vector` *size*) is identical to (`vector t` *size*) except that it designates a set of simple vectors. A simple vector is a vector that is not displaced to another array, that has no fill pointer, and whose size cannot be dynamically adjusted.

Vectors whose elements are restricted to string characters or bits are termed strings and bit vectors respectively.

The type specifier (`string` *size*) is an abbreviation for (`array string-char` (*size*)). Likewise, (`simple-string` *size*) is an abbreviation for (`simple-array string-char` (*size*)).

The type specifier (`bit-vector` *size*) is an abbreviation for (`array bit` (*size*)), and (`simple-bit-vector` *size*) is an abbreviation for (`simple-array bit` (*size*)).

Type Specifiers for Numerical Subranges

Numerical subrange types may also be denoted by the type specifiers.

The type specifier (`integer` *integer-limit integer-limit*) denotes the set of integers in the given range. Either argument may be specified as an integer, a list of an integer, or `*`. An integer argument specifies an inclusive limit; a list argument specifies an exclusive limit; and `*` means that there is no limit on the value. The type (`integer 0 1`) is equivalent to the type **bit**.

The type specifier (`mod` *integer*) denotes the set of nonnegative integers whose values are less than *integer*. It is equivalent to (`integer 0` (*integer*)).

The type specifier `(signed-byte` *size*`)` denotes the set of integers that can be represented in two's complement format in a byte of *size* bits or less. The type `(signed-byte *)` is equivalent to `integer`.

The type specifier `(unsigned-byte` *size*`)` denotes the set of nonnegative integers that can be represented in a byte of *size* bits or less. The type specifier `(unsigned-byte *)` is equivalent to `(integer 0 *)`.

The type specifier `(fixnum` *fixnum-limit fixnum-limit*`)` is like `(integer` *integer-limit integer-limit*`)` except it denotes fixnums in the given range. The arguments must be fixnums, lists of fixnums, or unspecified.

The type specifier `(rational` *rational-limit rational-limit*`)` denotes the set of rational numbers in the given range. Either argument may be specified as a rational, a list of a rational, or *. A rational argument denotes an inclusive limit; a list argument denotes an exclusive limit; and * means that there is no limit on the value.

The type specifiers `(float` *float-limit float-limit*`)`, `(short-float` *short-float-limit short-float-limit*`)`, `(single-float` *single-float-limit single-float-limit*`)`, `(double-float` *double-float-limit double-float-limit*`)`, and `(long-float` *long-float-limit long-float-limit*`)` denote subranges of floating-point numbers of the given types. Either argument may be specified as a floating-point number, a list of a floating-point number, or *. A floating-point number argument denotes an inclusive limit; a list argument denotes an exclusive limit; and * means that there is no limit on the value. The arguments must be of the appropriate floating-point format. In Sun Common Lisp, all floating-point numbers are represented in single-float format.

The type specifier `(complex` *typespec*`)` denotes the set of complex numbers whose real and imaginary parts are of the given type.

Type Specifiers for Functions

The type specifier `(function` *arg-typespec-list value-typespec*`)` is used in declaring functions. It denotes the set of functions that accept arguments of the given types and produce results that belong to the specified value type. The `function` type specifier is not acceptable to **typep**.

Categories of Operations

This section groups operations on type specifiers according to functionality.

Defining and Manipulating Types

coerce	**deftype**

These functions define type specifiers and manipulate the types of objects.

Discriminating Among Types

subtypep	**type-of**
typep	**commonp**

These functions discriminate among types.

coerce

Purpose: The function **coerce** is used to convert an object from one data type to another; the resulting object is returned. If such a coercion is not possible, an error is signaled. If it is already of the required *result-type*, the original object is returned.

The coercions listed below are the only ones that are possible.

Syntax: **coerce** *object result-type* [*Function*]

Remarks: The following conversions are performed by **coerce**.

A sequence type may be converted to any other sequence type, provided that the resulting sequence is of a type that is compatible with the types of the elements of the original sequence. Elements of the new sequence will be **eql** to corresponding elements of the original sequence.

Certain objects may be converted to characters: strings of length 1, symbols whose print names are of length 1, and nonnegative integers *n* for which `(int-char n)` is defined. Coercing a string of length 1 results in the character contained in that string. Coercing a symbol whose print name is of length 1 results in the character contained in that print name string. Coercing a nonnegative integer for which `(int-char n)` is defined results in the character defined by `(int-char n)`.

Any number may be converted to a complex number.

Any noncomplex number may be converted to a floating-point number.

Any object may be converted to type **t**.

Examples:

```
> (setq *print-array* t)
T
> (coerce '(a b c) 'vector)
#(A B C)
> (coerce 'a 'character)
#\A
> (coerce 4.56 'complex)
#C(4.56 0.0)
> (coerce (cons 1 2) t)
(1 . 2)
```

See Also: **rational**

rationalize

char-code

char-int

commonp

Purpose: The predicate **commonp** is true if its argument is a member of any standard Common Lisp data type; otherwise it is false.

Syntax: **commonp** *object* [*Function*]

Examples:

```
> (commonp *query-io*)
T
> (commonp nil)
T
> (commonp (expt 2 130))
T
```

deftype

Purpose: The **deftype** macro is used to define a name for a new type specifier.

The **deftype** macro is like the **defmacro** macro in that the *form* arguments of its body constitute an expansion function for the type specifier definition.

The *name* of the new type specifier is returned as the value of the **deftype** form.

Syntax: **deftype** *name lambda-list* {*declaration* | *documentation*}* {*form*}* [*Macro*]

Remarks: The lambda list may contain **&optional** and **&rest** keywords.

If no *initform* is specified for an **&optional** *lambda-list* argument, the default value * is used.

If the type name is used as an atomic type specifier, it is treated as a list with no arguments.

A documentation string may be attached to the name of the type by use of the optional *documentation* argument; the documentation type for this string is **type**.

Examples:

```
> (deftype modd2 (&optional (limit 2)) '(integer 0 ,limit))
MODD2
> (typep 0 '(modd2))
T
> (typep 3 '(modd2))
NIL
> (typep 3 '(modd2 5))
T
```

See Also: **defmacro**

subtypep

Purpose: The function **subtypep** compares two type specifiers. It returns two values. If *type1* is definitely a subtype of *type2*, then true and true are returned. If *type1* is definitely not a subtype of *type2*, then false and true are returned. In all other cases, false and false are the values returned.

Syntax: **subtypep** *type1 type2* [*Function*]

Remarks: Type arguments of **subtypep** must be type specifiers that are acceptable to **typep**.

The type *type1* may be a proper subtype of *type2*.

Examples:

```
> (subtypep 'compiled-function 'function)
T
T
> (subtypep 'integer 'string)
NIL
INTEGER
> (subtypep '(satisfies foo) nil)
NIL
NIL
```

type-of

Purpose: The function **type-of** returns a type of which its *object* argument is a member.

Syntax: **type-of** *object* [*Function*]

Remarks: If the object is an instance of a structure created by the use of the **defstruct** construct, **type-of** returns the type name for the structure. In all other instances, **type-of** is probably useful only for debugging purposes. The action of **type-of** can be implementation dependent.

Examples:

```
> (type-of 'a)
SYMBOL
> (type-of "abc")
SIMPLE-STRING
> (type-of '(1 . 2))
CONS
> (type-of #c(0 1))
COMPLEX
> (defstruct foo x y z)
FOO
> (type-of (make-foo))
FOO
```

See Also:

typep

typecase

defstruct

typep

Purpose: The predicate **typep** tests an object for membership in a particular data type.

Syntax: **typep** *object type-specifier* [*Function*]

Remarks: The *type-specifier* argument may be any type specifier except **function**, **values**, or a list whose first element is either of these.

Examples:

```
> (typep 12 'integer)
T
> (typep nil t)
T
> (typep nil nil)
NIL
> (typep 1 '(mod 2))
T
```

Chapter 4. Program Structure

Chapter 4. Program Structure

About Program Structure

Common Lisp programs are built from forms. A ***form*** is any data object that can be evaluated to produce values and, possibly, side effects. In particular, certain forms call functions to perform computations upon other forms. Some of these forms also define functions. Not all data objects can be evaluated; hence not all data objects are valid forms.

A ***function*** is a data object that performs computations upon forms. When a function is called, the function's arguments are bound to values, and the forms contained within the function body are evaluated in the context of these bindings. Normally functions also return one or more values.

Forms

There are five basic categories of forms: self-evaluating forms, variables, special forms, macro calls, and function calls.

Self-evaluating Forms

Self-evaluating forms are forms that evaluate to themselves. The value of a self-evaluating form is that object itself. The following are self-evaluating forms: numbers, characters, strings, bit vectors, keywords, **t**, and **nil**. The predicate **constantp** is true of any self-evaluating form.

Variables

Variables provide symbolic references to the objects of a Lisp form. Variables can be either lexical or special, depending on the program context.

A variable is an association of an identifier with a location. The location is the cell or cells where the value associated with the variable is stored. The association between the variable name and the location is termed a ***binding***. Depending on the type of binding that is current for the given identifier, this location may be a register, a stack location, or some other memory location. In particular, for certain types of variables, it may be the value cell of a symbol.

Bindings may be either ***lexical*** or ***dynamic***. Correspondingly, a given variable is either a lexical or a dynamic variable, depending on the program context. Dynamic variables are also called ***special variables***.

The ***scope*** of a binding is that portion of a program in which the binding is in effect. The scope of a variable thus determines when and where the variable may be referenced.

A ***lexical variable*** is a variable whose scope is lexical or textual. That is, the variable may be accessed only by expressions that lie textually within the same construct in which the variable was established. Lexical variables are created by lambda expressions, **let** forms, function definitions, and a number of other basic forms. The control structures that create lexical variables are discussed in the chapter "Control Structure."

A ***special variable*** consists of the binding of an identifier to the value cell of a symbol. This binding may temporarily alter the value of the symbol. Variables created by **let** and similar constructs may be declared special. The scope of a special variable is dynamic. This means that until the construct that establishes the variable binding terminates, references to the variable name access the special variable, even though such references may not be textually within the scope of the establishing construct. The declaration of special variables is discussed in the chapter "Declarations."

When new variable bindings are created, existing variable bindings may be shadowed. ***Shadowing*** occurs when a name or identifier that is meaningful at a given point is re-used there for a different item. In this case, the newly created item shadows the older item, causing references to the common name to refer to the new item.

The context of bindings that are visible at a given point in a program is termed the ***environment***. The ***lexical environment*** consists of those lexical bindings that are visible at a particular point in the program, as determined by the structure of the program text. The lexical environment of a top-level form is termed the ***null lexical environment***. This environment has no lexical bindings. The ***dynamic environment*** consists of those dynamic bindings that are visible at a particular point during program execution, as determined by the dynamic execution of the program. The dynamic environment is also referred to as the ***global environment***.

Special Forms

A ***special form*** is a list form whose first element is one of a limited set of symbols. No new special forms may be defined by the user.

Special forms are processed in a special manner by the evaluator and the compiler. Special evaluation rules are invoked for these forms.

Like functions and macros, special forms may return one or more values or cause nonlocal exits.

The following table lists all of the Common Lisp symbols that have definitions as special forms.

block	**if**	**progv**
catch	**labels**	**quote**
compiler-let	**let**	**return-from**
declare	**let***	**setq**
eval-when	**macrolet**	**tagbody**
flet	**multiple-value-call**	**the**
function	**multiple-value-prog1**	**throw**
go	**progn**	**unwind-protect**

Figure 4–1. Common Lisp Special Forms

Macros

A ***macro call*** is a list form whose first element is the name of a macro. A macro call returns a Lisp expression to be evaluated in place of the macro call. Macros thus provide a text replacement facility. They enable the user to write forms that do not obey the usual rules for evaluation. Macros are discussed in the chapter "Macros."

Function Calls

A ***function call*** is a list form whose first element is either the name of a function or an anonymous function definition (lambda expression). The remaining elements of the list form are considered to be the arguments to the function. The arguments are evaluated as forms in the order in which they occur, and the function is invoked upon them. This process is called ***applying*** the function to the arguments.

The actual function arguments are all evaluated before the function is invoked, and the formal function parameters are bound to the resulting values. (If any function argument results in more than one value, only the first of these values is used.) If the resulting values are pointers to objects and the function modifies its arguments, the original data objects may be modified as a side effect of the function call.

The function invocation may result in one or more values, or it may cause a nonlocal exit. The result of the function call form is the result returned by the function.

Functions

A function may be specified in a function call form in one of two ways: by the function name or by a lambda expression.

Named Functions

A named function is a function object to which a name has been given either by use of the **defun** macro or by the **flet** or **labels** special form. The use of a name to name a function is completely independent of any association it may have as a variable identifier.

Lambda Expressions

A lambda expression defines an anonymous function.

A lambda expression acts just like a function, but it is not associated with a function name.

The syntax for lambda expressions is the following:

```
(lambda lambda-list {declaration | documentation}* {form}*)

lambda-list::= ({var}*
                [&optional {var | (var [initform [supplied-p-parameter] ])}*]
                [&rest var]
                [&key {var | ({var | (keyword var)} [initform [supplied-p-parameter] ])}*
                      [&allow-other-keys] ]
                [&aux {var | (var [initform] )}*] )
```

Lambda Lists

Lambda lists are used in the specification of named functions and lambda expressions. The lambda list specifies the parameters of the function. When the function is applied to arguments, the parameters specified in the lambda list are bound to actual argument values, and the forms in the body of the lambda expression or function are executed in the context of these bindings.

- All required parameters must be specified first. All parameters preceding the first lambda list keyword are required parameters. They are bound to actual argument values in the order in which they occur. There must be at least as many actual argument forms as there are required parameters. If no lambda list keywords are specified, there must be exactly as many actual arguments as parameters.

- Any optional parameters must be specified next. They are preceded by the lambda list keyword **&optional**. If optional parameters are specified, they are bound in order to the corresponding remaining values in the argument list. If there are no remaining arguments at any point in the processing of optional parameters, then any remaining optional parameter is bound to the value that results from the evaluation of its associated *initform*, if the latter is given, or to **nil**, if not. A *supplied-p-parameter* variable may be used in conjunction with an *initform*. Its purpose is to indicate whether an actual argument value was supplied. It is bound to *true* if an actual argument was supplied; otherwise (if the *initform* was evaluated) it is bound to **nil**.

- One rest parameter may be specified next. It is preceded by the **&rest** lambda list keyword. If a rest parameter has been specified, it is bound to a list consisting of all the actual arguments that have not yet been processed. If no arguments remain, the rest parameter is bound to **nil**.

- The use of the lambda list keyword **&key** and keyword parameter specifiers allows keyword arguments to be used in function calls. If any keyword parameters are to appear in the function call, they must be preceded by **&key** in the lambda list. These keyword parameters may be followed by the lambda list keyword **&allow-other-keys**.

  A keyword parameter may be specified in one of three ways. These forms differ in whether the name for the keyword to be used in the actual argument list is specified explicitly or implicitly and whether an initial value is to be used if such a keyword argument is not given.

  If a variable, *var*, specifies the keyword parameter, the keyword argument to be used in the argument list consists of a keyword (in the keyword package) with the same name as *var*. If such a keyword does not appear in the argument list, *var* is bound to **nil**.

  If the form (*var* [*initform* [*supplied-p-parameter*]]) specifies the keyword parameter, the keyword argument to be used is specified in the same way as in the simpler case discussed above. This construct, however, allows the variable to be bound to an initial value if the keyword is not specified in the argument list. The *supplied-p-parameter* may be used to test whether such an argument value was given.

  The form ((*keyword var*) [*initform* [*supplied-p-parameter*]]) allows the explicit specification of the argument list keyword that is associated with *var*. It also allows the variable to be bound to an initial value if the keyword is not specified in the argument list.

  There must be an even number of actual keyword arguments. Keyword arguments are considered to occur in pairs. The first argument in the pair is a keyword; the second is the value to which the corresponding keyword parameter is to be bound. 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. If a given keyword argument is specified more than once, however, only the first keyword-value pair is used in the binding of the keyword parameter. If a rest parameter has been specified, the arguments used in processing keyword parameters are the same as those used in processing the rest parameter.

- The **&allow-other-keys** lambda list keyword is used to specify that the argument list may contain a keyword that does not correspond to a lambda list keyword parameter. Otherwise it is an error if such an argument pair occurs unless the argument list contains a keyword-value pair whose key is **:allow-other-keys** and whose value is non-**nil**. The **&rest** keyword parameter may be used to access values specified by means of the **&allow-other-keys** and **:allow-other-keys** constructs.

  It is an error if there are remaining arguments and neither a rest parameter nor a keyword parameter has been specified.

- Finally, the **&aux** lambda list keyword may be used to specify auxiliary variables. These serve as local variables within the lambda expression or function. Auxiliary variables are not bound to argument list values. An auxiliary variable may be bound within the lambda expression itself or by specifying a corresponding *initform* in the lambda list.

Since the lambda list elements are processed in the order in which they occur, any *initform* may reference a parameter variable (including a *supplied-p-parameter* variable) that is bound earlier in the processing of the lambda list.

After the lambda list parameters are bound to actual argument values, the forms contained in the body of the lambda expression or function are evaluated in sequence in the context of these bindings. The result returned by the lambda expression or function is the result of the last form evaluated. If no forms are evaluated, **nil** is returned.

The variable bindings in effect before the function invocation are restored when the function exits.

Lexical Closures

A closure is a function along with a binding context. When a function or lambda expression is created, it is created within a context of lexical bindings. Creating a lexical closure means retaining this lexical environment of bindings through the lifetime of the function (closure) object. The function is thus able to reference these same bindings in different invocation contexts. With closures it is thus possible to create objects that retain separate contexts that can be manipulated.

The following example shows a function that returns a lexical closure in which the variable *x* is bound to 20. When the closure function is itself invoked, this binding is referenced.

```
> (defun foo ()
    (let ((x nil) (fn nil))
      (setq fn #'(lambda (y) (setq x (cons x y))))
      (setq x 20)
      fn))
FOO
> (funcall (foo) 1)
(20 . 1)
```

Functions that are intended to generate a series of new values for consumption by other functions are called ***generators***. The following example shows a generator that is written as a lexical closure. It generates the positive integers. Each time it is called, it produces a new integer in the series. The internal state of the generator is maintained in the lexical closure.

```
> (setq closure (let ((x 0)) #'(lambda () (incf x) x)))
#<Interpreted-Function (LAMBDA NIL (INCF X) X) 40FC97>
> (funcall closure)
1
> (funcall closure)
2
> (funcall closure)
3
```

Categories of Operations

This section groups operations related to program structure according to functionality.

Data Type Predicates

functionp	**compiled-function-p**

These predicates determine whether an object is a function object.

Declaring Global Variables and Named Constants

defconstant	**defvar**
defparameter	

These constructs proclaim special variables and constants.

Function Definition

defun	**lambda-list-keywords**
define-function	**lambda-parameters-limit**
redefinition-action	

These constructs are used in defining functions.

Function Calls

apply	**call-arguments-limit**
funcall	

These constructs are used in applying functions to arguments.

Accessing Variable and Function Bindings

symbol-value	**makunbound**
symbol-function	**fmakunbound**
boundp	**function**
fboundp	**special-form-p**
	constantp

These operations access variable and function bindings.

Controlling Evaluation

quote	**eval-when**

These functions affect the evaluation process.

Identity Operator

identity

This function returns its argument unchanged.

apply

Purpose: The function **apply** applies its *function* argument to a list of arguments.

The *function* argument must be a function object. It may be a compiled code object, a lambda expression, or a symbol that has a global definition as a function (not a macro or special form).

Syntax: **apply** *function arg* &rest *more-args* [*Function*]

Remarks: The last argument specified must be a list. It is appended to a list of all the other arguments except *function*.

If the function uses keyword arguments, the keywords must also be given in the argument list.

The macro **setf** may be used with **apply** if the *function* argument is a function that is acceptable to **setf**.

Examples:

```
> (apply #'+ 1 2 3 '(4 5 6))
21
> (apply #'(lambda (x y z) (+ x (- y z))) '(1 2 3))
0
```

See Also: **funcall**

function

boundp

Purpose: The predicate **boundp** is true if the dynamic variable associated with its *symbol* argument has a value; otherwise it is false.

Syntax: **boundp** *symbol* [*Function*]

Examples:

```
> (setq sym 1)
1
> (boundp 'sym)
T
> (makunbound 'sym)
SYM
> (boundp 'sym)
NIL
> (let ((sym 2)) (boundp 'sym))
NIL
```

See Also:

set

setq

symbol-value

makunbound

call-arguments-limit

Purpose: The constant **call-arguments-limit** defines the upper exclusive bound on the number of arguments that may be passed to any Common Lisp function.

The value of **call-arguments-limit** in Sun Common Lisp is 2^9.

Syntax: **call-arguments-limit** [*Constant*]

Examples:

```
> call-arguments-limit
512
```

See Also: **lambda-parameters-limit**

multiple-values-limit

compiled-function-p

Purpose: The predicate **compiled-function-p** is true if its argument is a compiled code object; otherwise it is false.

Syntax: **compiled-function-p** *object* [*Function*]

Examples:

```
> (compiled-function-p (symbol-function 'append))
T
> (compiled-function-p #'(lambda (x) x))
NIL
```

constantp

Purpose: The predicate **constantp** is true if its argument is a constant; otherwise it is false. A constant is an object that always evaluates to the same value.

The following objects are constants: numbers, characters, strings, keywords, **t**, **nil**, bit vectors, symbols declared by means of **defconstant**, and lists whose first element is **quote**.

Syntax: **constantp** *object* [*Function*]

Examples:

```
> (constantp 1)
T
> (constantp ''foo)
T
> (defconstant this-is-a-constant 'never-changing)
THIS-IS-A-CONSTANT
> (constantp 'this-is-a-constant)
T
> (constantp "foo")
T
```

See Also: **defconstant**

defconstant

Purpose: The **defconstant** macro is used to proclaim a special variable. The variable is initialized to the result of evaluating the *initial-value* argument. Once such a variable has been defined using **defconstant**, its value is constant and may not be changed by assignment or binding.

The **defconstant** macro returns *name* as its result.

Syntax: **defconstant** *name initial-value* [*documentation*] [*Macro*]

Remarks: The *name* argument is a symbol; it is not evaluated.

No special binding of the variable may already exist when **defconstant** is called.

Note that a constant defined by **defconstant** may be changed with **defconstant**, but functions compiled using the old value may be incorrect.

A documentation string may be attached to the name of the global variable by the optional *documentation* argument; the documentation type for this string is **variable**.

Examples:

```
> (defconstant this-is-a-constant 'never-changing "for a test")
THIS-IS-A-CONSTANT
> this-is-a-constant
NEVER-CHANGING
> (documentation 'this-is-a-constant 'variable)
"for a test"
> (constantp 'this-is-a-constant)
T
```

See Also:

defvar

defparameter

proclaim

documentation

define-function

Purpose: The function **define-function** is used by the macro **defun** to do the actual defining of a new function. It replaces the function cell of the named symbol with the specified function object.

If the function is currently traced, it remains traced, but with the new definition.

Syntax: **define-function** *name function* [*Function*]

Remarks: The *name* argument is a symbol.

The function **define-function** is an extension to Common Lisp.

Examples:

```
> (defun foo () 101)
FOO
> (foo)
101
> (define-function 'foo #'+)
FOO
> (foo)
0
> (foo 1 2 3)
6
> (define-function 'foo #'(lambda () 202))
FOO
> (foo)
202
```

See Also: **defun**

symbol-function

redefinition-action

defparameter

Purpose: The **defparameter** macro is used to proclaim a special variable. The variable is initialized to the result of evaluating the *initial-value* argument.

The **defparameter** macro returns *name* as its result.

Syntax: **defparameter** *name* *initial-value* [*documentation*] [*Macro*]

Remarks: The *name* argument is a symbol; it is not evaluated.

A documentation string may be attached to the name of the global variable by the optional *documentation* argument; the documentation type for this string is **variable**.

Examples:

```
> (defparameter *p* 1)
*P*
> *p*
1
> (constantp '*p*)
NIL
> (setq *p* 2)
2
> (defparameter *p* 3)
*P*
> *p*
3
```

See Also: **defvar**

defconstant

proclaim

documentation

defun

Purpose: The **defun** macro is used to define a new function.

The *name* argument of **defun** must be a symbol; it is not evaluated. The function **defun** causes a global function definition to be attached to the symbol *name* as the contents of the symbol's function cell. This function definition is given by the expression that follows:

(lambda *lambda-list* {*declaration* | *documentation*}* {*form*}*)

The *name* of the new function is returned as the value of the **defun** form.

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

The function body is enclosed in a **block** construct. This block bears the same name as the function itself. Thus the **return-from** construct may be used to cause an exit from the function as well as the block.

Syntax: **defun** *name lambda-list* {*declaration* | *documentation*}* {*form*}* [*Macro*]

Remarks: The definition of functions and the syntax of lambda lists are discussed in the section "Functions."

The function is defined in the lexical environment in which the **defun** form is executed. Normally, the **defun** macro occurs as a top-level form. If it is a top-level form, the null lexical environment is used.

A documentation string may be attached to the name of the function by use of the optional *documentation* argument; the documentation type for this string is **function**.

The **defun** macro may be used to redefine a function or to replace a macro definition with a function definition. The Common Lisp special forms may not be redefined.

Examples:

```
> (defun ex (a b &optional c (d 66) &rest keys &key test (start 0))
    (list a b c d keys test start))
EX
> (ex 1 2)
(1 2 NIL 66 NIL NIL 0)
> (ex 1 2 3 4 :test 'equal :start 50)
(1 2 3 4 (:TEST EQUAL :START 50) EQUAL 50)
> (ex :test 1 :start 2)
(:TEST 1 :START 2 NIL NIL 0)
```

defun

See Also: **flet**

labels

block

return-from

documentation

defvar

Purpose: The **defvar** macro is used to proclaim a special variable.

If an *initial-value* argument is specified, the variable is initialized to the result of evaluating *initial-value*. If the variable already has a value, this value is not changed and *initial-value* is not evaluated.

The **defvar** macro returns *name* as its result.

Syntax: **defvar** *name* [*initial-value* [*documentation*]] [*Macro*]

Remarks: The *name* argument is a symbol; it is not evaluated.

A documentation string may be attached to the name of the global variable by use of the optional *documentation* argument; the documentation type for this string is **variable**.

Examples:

```
> (defvar *v* 'global)
*V*
> *v*
GLOBAL
> (let ((*v* 'local))
    (symbol-value '*v*))
LOCAL
> (setq should-stay-nil nil)
NIL
> (defvar *v* (setq should-stay-nil t))
*V*
> *v*
GLOBAL
> should-stay-nil
NIL
```

See Also: **defparameter**

proclaim

documentation

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. Each *situation* argument must be either **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 *form* arguments are executed in order. 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*]

Examples:

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

fboundp

Purpose: The predicate **fboundp** is true if its *symbol* argument has an associated global function definition; otherwise it is false.

Syntax: **fboundp** *symbol* [*Function*]

Remarks: The function definition may be that of a function, a macro, or a special form.

Examples:

```
> (defun foo (x) x)
FOO
> (fboundp 'foo)
T
> (fmakunbound 'foo)
FOO
> (fboundp 'foo)
NIL
> (flet ((foo #'(lambda (x) x)))
    (fboundp 'foo))
NIL
```

See Also: **symbol-function**

fmakunbound

fmakunbound

Purpose: The function **fmakunbound** causes its *symbol* argument to have no associated global function definition. It returns *symbol* as its result.

Syntax: **fmakunbound** *symbol* [*Function*]

Examples:

```
> (defun foo (x) x)
FOO
> (fboundp 'foo)
T
> (fmakunbound 'foo)
FOO
> (fboundp 'foo)
NIL
> (flet ((foo (x) (1+ x)))
    (fmakunbound 'foo)
    (foo 1))
2
```

See Also: **fboundp**

funcall

Purpose: The function **funcall** applies its *function* argument to the specified arguments.

The *function* argument must be a function object. It may be a compiled code object, a lambda expression, or a symbol that has a global definition as a function (not a macro or special form).

Syntax: **funcall** *function* &rest *args* [*Function*]

Examples:

```
> (funcall #'+ 1 2 3)
6
> (funcall 'car '(1 2 3))
1
> (funcall 'position 1 '(1 2 3 2 1) :start 1)
4
> (funcall #'(lambda () 101))
101
```

See Also: **apply**

function

function

Purpose: The special form **function** returns the function object associated with its argument.

If the *function* argument is a symbol, this object is the function definition that is associated with the symbol's function cell. If *function* is a lambda expression, **function** returns a lexical closure for that lambda expression.

Syntax: **function** *function* [*Special Form*]

Remarks: The notation `#'`*function* may be used as an abbreviation for `(function` *function*`)`.

The *function* argument is not evaluated.

Examples:

```
> (defun foo () 'top-level)
FOO
> (funcall (function foo))
TOP-LEVEL
> (flet ((foo () 'shadow))
    (funcall (function foo)))
SHADOW
> (eq (function foo) #'foo)
T
> (eq (function foo) (symbol-function 'foo))
T
> (flet ((foo () 'shadow))
    (eq (function foo) (symbol-function 'foo)))
NIL
```

functionp

Purpose: The predicate **functionp** is true if its argument is of a form that is appropriate for applying to arguments, as with the **funcall** or **apply** function; otherwise it is false.

Syntax: **functionp** *object* [*Function*]

Remarks: The predicate **functionp** is true of symbols, any list whose first element is **lambda**, values returned by **function**, and values returned by **compile** when its first argument is **nil**.

Examples:

```
> (functionp 'sss)
T
> (functionp (symbol-function 'append))
T
> (functionp :test)
T
> (functionp nil)
T
> (functionp 12)
NIL
```

identity

Purpose: The function **identity** returns its argument unchanged. It is intended for use with functions that require a function as an argument.

Syntax: **identity** *object* [*Function*]

Examples:

```
> (identity 101)
101
> (let ((f #'identity))
    (funcall f 101))
101
> (mapcan #'identity '((1 2 3) (4 5 6)))
(1 2 3 4 5 6)
```

lambda-list-keywords

Purpose: The constant **lambda-list-keywords** defines the lambda list keywords that are available for use in lambda expressions, function definitions, and macro definitions. Its value is a list. This list contains the symbols **&optional**, **&rest**, **&key**, **&aux**, **&allow-other-keys**, **&body**, **&whole**, and **&environment**.

Syntax: **lambda-list-keywords** *[Constant]*

Remarks: The lambda list keywords **&body**, **&whole**, and **&environment** may be used only in macro definitions.

The use of lambda list keywords in function definitions is discussed in the section "Functions." The use of lambda list keywords in macro definitions is discussed in the chapter "Macros."

Examples:

```
> lambda-list-keywords
(&OPTIONAL &REST &KEY &AUX &ALLOW-OTHER-KEYS &BODY &WHOLE &ENVIRONMENT)
```

lambda-parameters-limit

Purpose: The constant **lambda-parameters-limit** defines the upper exclusive bound on the number of distinct parameter names in a lambda list.

The value of **lambda-parameters-limit** in Sun Common Lisp is 2^9.

Syntax: **lambda-parameters-limit** [*Constant*]

Examples:

```
> lambda-parameters-limit
512
```

See Also: **call-arguments-limit**

makunbound

Purpose: The function **makunbound** causes the dynamic variable associated with its *symbol* argument to be unbound (have no value). It returns *symbol* as its result.

Syntax: **makunbound** *symbol* [*Function*]

Examples:

```
> (setq foo 1)
1
> (boundp 'foo)
T
> (makunbound 'foo)
FOO
> (boundp 'foo)
NIL
```

See Also: **boundp**

quote

Purpose: The special form **quote** returns its *object* argument. The object is not evaluated. The special form **quote** is used when it is desirable not to evaluate an object or form, but rather to manipulate it as a constant.

Syntax: **quote** *object* [*Special Form*]

Remarks: The single-quote character may be used as an abbreviation for **quote**. The construct `'`*object* is equivalent to `(quote` *object*`)`.

Examples:

```
> (setq a 1)
1
> (quote (setq a 1))
(SETQ A 1)
> a
1
> 'a
A
> ''a
(QUOTE A)
```

redefinition-action

Purpose: The global variable ***redefinition-action*** is only used in the functions **define-function** and **define-macro**. It is used to specify what action will be taken when a redefinition occurs.

If ***redefinition-action*** is set to **:warn**, the user is warned when a function or macro is redefined. If the variable is set to **:query**, the user is asked whether he wishes to proceed with the redefinition. If ***redefinition-action*** is set to any other value, no warning is given.

The default value of ***redefinition-action*** is **:warn**.

Syntax: ***redefinition-action*** *[Variable]*

Remarks: The variable ***redefinition-action*** is an extension to Common Lisp.

Examples:

```
> *redefinition-action*
:WARN
> (defun foo ())
FOO
> (defmacro bar ())
BAR
> (defun foo ())
;;; Warning: Redefining FOO
FOO
> (defmacro bar ())
;;; Warning: Redefining BAR
BAR
> (define-function 'foo #'car)
;;; Warning: Redefining FOO
FOO
> (define-macro 'bar #'do)
;;; Warning: Redefining BAR
BAR
> (let ((*redefinition-action* :quiet))
    (defun foo ()))
FOO
> (let ((*redefinition-action* :quiet))
    (define-macro 'bar #'do))
BAR
> (setq *redefinition-action* :quiet)
:QUIET
> (defun foo ())
FOO
> (defmacro bar ())
BAR
```

```
> (define-function 'foo #'car)
FOO
> (define-macro 'bar #'do)
BAR
```

See Also: **define-function**

define-macro

special-form-p

Purpose: The predicate **special-form-p** is true if its *symbol* argument has an associated global function definition that is a special form; otherwise it is false.

Syntax: **special-form-p** *symbol* [*Function*]

Examples:

```
> (special-form-p 'if)
T
> (special-form-p 'car)
NIL
> (special-form-p 1)
NIL
```

symbol-function

Purpose: The function **symbol-function** returns the contents of the function cell named by its *symbol* argument. This function definition may be a function, a special form, or a macro. An error is signaled by **symbol-function** if the function definition does not exist.

Syntax: **symbol-function** *symbol* [*Function*]

Remarks: The existence of a function definition associated with a symbol may be tested with **fboundp**.

The macro **setf** may be used with **symbol-function** to replace the contents of the function cell.

Examples:

```
> (defun foo () "this function returns this string")
FOO
> (funcall (symbol-function 'foo))
"this function returns this string"
> (setf (symbol-function 'foo)
        #'(lambda () "this function is a replacement"))
#<Interpreted-Function (LAMBDA NIL "this function is a replacement") 3B85AF>
> (funcall (symbol-function 'foo))
"this function is a replacement"
```

See Also: **fboundp**

symbol-value

Purpose: The function **symbol-value** returns the contents of the value cell of the variable associated with its *symbol* argument. An error is signaled if this variable is unbound.

The function **symbol-value** may also be applied to named constants and keywords. Applying **symbol-value** to a keyword returns that keyword.

Syntax: **symbol-value** *symbol* [*Function*]

Remarks: The predicate **boundp** may be used to test the existence of a value associated with a symbol.

The macro **setf** may be used with **symbol-value** to replace the contents of the value cell.

Examples:

```
> (setq a 1)
1
> (symbol-value 'a)
1
> (let ((a 2)) (symbol-value 'a))
1                          ;only the global value is given by symbol-value
> (let ((a 2))
   (setf (symbol-value 'a) 3)
   a)
2
> a
3
> (symbol-value 'a)
3
```

See Also: **boundp**

set

setq

Chapter 5. Control Structure

Chapter 5. Control Structure

About Control Structure

Common Lisp provides many different constructs for controlling the flow of program execution and evaluation. This collection of functions, macros, and special forms encourages the design of clear, understandable programs.

The available programming constructs include assignment to lexical, dynamic, and generalized variables; various forms of iteration; conditionals; blocks; function calls; nonlocal transfers and exits; and function returns with multiple values.

Assignment Constructs

Common Lisp provides both simple and generalized assignment constructs.

Simple Assignment

The **set**, **setq**, and **psetq** constructs are used to alter the values of variables. The **set** function is used to alter the value of a dynamic variable. The **setq** and **psetq** forms may be used to assign values to both lexical and dynamic variables.

Generalized Variables

A simple variable is a binding of an identifier with a location. It is accessed by name. Common Lisp also provides a more general notion of variable. A ***generalized variable*** is a binding of an accessing formula with a location.

Like simple variables, generalized variables can be updated. The syntax for updating generalized variables requires, in place of the variable name, a specification of the accessing formula for the variable.

In the syntactic descriptions of operations on generalized variables, this accessing formula is referred to as a *place* form. It may be any one of the following:

- The name of a lexical or dynamic variable.
- A call to a selector function created by means of **defstruct**.
- A call to any of the functions listed in Figure 5–1.
- A **the** type declaration.
- Calls to access forms defined by **defsetf** or **define-setf-method**.
- Calls to **apply** that also have special meaning to **setf**.
- A macro call that expands into one of these forms.

symbol-value	**aref**	**car**	**caaadr**
symbol-function	**svref**	**cdr**	**caadar**
symbol-plist	**get**	**caar**	**caaddr**
macro-function	**elt**	**cadr**	**cadaar**
documentation	**getf**	**cdar**	**cadadr**
first	**gethash**	**cddr**	**caddar**
second	**fill-pointer**	**caaar**	**cadddr**
third	**char**	**caadr**	**cdaaar**
fourth	**schar**	**cadar**	**cdaadr**
fifth	**bit**	**caddr**	**cdadar**
sixth	**sbit**	**cdaar**	**cdaddr**
seventh	**subseq**	**cdadr**	**cddaar**
eighth	**char-bit**	**cddar**	**cddadr**
ninth	**ldb**	**cdddr**	**cdddar**
tenth	**mask-field**	**caaaar**	**cddddr**
nth	**rest**		

Figure 5–1. Table of Place Constructors

The macro **setf** takes a generalized place specifier and a value and stores the value in the specified location. It is intended to be used for all operations that need to update a piece of data. Using **setf** uniformly to update such data eliminates the need for numerous different functions to do updating on different types of data locations.

Blocks and Sequencing

The forms **progn**, **prog1**, and **prog2** provide the primitive sequencing constructs of Common Lisp. They cause a series of forms to be executed in the order in which they are listed as arguments.

The **block** special form acts in a similar way but allows a name to be associated with the series of forms. The execution of a block may be terminated by the use of the **return** and **return-from** constructs. The **defun** macro provides an implicit **block** around the body of the defined function. This block bears the same name as the funtion. The iteration forms **loop**, **do**, **do***, **dolist**, and **dotimes** also provide implicit blocks.

The **prog**, **prog***, **progv**, **let**, **let***, and **compiler-let** constructs establish new variable bindings and execute a series of forms using these bindings. These constructs differ in the types of bindings they provide and in how the bindings are made. In addition, the **prog** and **prog*** constructs provide implicit tagbodies and thus allow for control transfer operations.

The **flet**, **labels**, and **macrolet** constructs establish new function definition bindings and execute a series of forms using these bindings.

Iteration

Common Lisp provides several forms of iteration.

The **loop** construct provides a primitive indefinite iteration facility.

The **do**, **do***, **dolist**, and **dotimes** constructs provide structured means of definite iteration. These forms all create bindings for iteration variables and provide for the execution of a series of forms within the context of these bindings. Explicit termination conditions may be specified for the iteration. The **dolist** construct is tailored for iterating over the elements of a list. The **dotimes** construct allows for iteration over a sequence of integers.

Conditionals

Conditional control structures allow the execution of forms to be contingent on the results of evaluating other forms.

The **if**, **when**, and **unless** constructs allow for the execution of a form to be dependent on the results of another form. The **cond** construct is a generalization of **if**. It provides a multibranch **if** facility.

The various **case** and **typecase** forms provide for the selective execution of one group of forms out of a set of many such groups. The selection is made on the basis of the value or type of a key associated with the set.

Control Transfer

The most common form of control transfer is the function call. Functions and the function call mechanism are discussed in the chapter "Program Structure."

A simple "goto" facility is provided by the **go** and **tagbody** constructs. The **tagbody** special form allows for control transfer within a body of code by means of *tags*, or statement labels. The **go** form is used to cause control to transfer to the statement labeled by the tag. The forms **do**, **do***, **dolist**, **dotimes**, **prog**, and **prog*** all have implicit tagbodies. Tagbody tags have lexical scope. A **go** form may thus transfer control only to a tag in a lexically surrounding tagbody.

The **return** and **return-from** constructs provide for structured exits from blocks. They are used in conjunction with the **block** construct. Block names have lexical scope. A **return** or **return-from** form may transfer control only to the end of a lexically surrounding block.

The **catch** and **throw** facility provides a means of control transfer in which the destination is determined by the dynamic environment.

The **unwind-protect** construct guarantees that a series of cleanup forms will be executed before a nonlocal exit occurs.

Multiple Values

Normally, a Lisp function returns a single value, although the single value might be a list or a vector of many objects. In certain cases, however, it is natural for a function to compute and return more than one value. Common Lisp provides a straightforward way of doing this.

Unless explicit requests are made both to return multiple values and to receive them, a function call supplies only a single value. If the function returns multiple values, but the caller expects only a single value, the result is the first value, and the remaining multiple values are discarded. If the function returns no values, but the caller expects a single value, the result is **nil**.

Many constructs that select a form to be returned will return multiple values if the selected form returns multiple values. These include **progn** and constructs where forms are executed in order. Constructs such as **defun**, **defmacro**, **eval-when**, **progv**, **let**, **when**, **block** and forms containing implicit blocks, **catch**, **case**, and **typecase** behave as if a **progn** had been wrapped around the series of forms that they execute.

Other forms that return any supplied multiple values are **eval**, **apply**, **funcall**, **multiple-value-call**, **if**, **return**, **return-from**, **multiple-value-prog1**, **unwind-protect**, and **the**. The macros **and** and **or** return multiple values only from the last subform. The macro **cond** returns multiple values unless the clause selected contains only a single form (the test itself). In that case, the single non-**nil** value of the test is returned.

Forms that always return only a single value include **setq**, **prog1**, and **prog2**.

Categories of Operations

This section groups operations related to control structure according to functionality.

Assignment

set	**shiftf**
setq	**define-modify-macro**
psetq	**defsetf**
setf	**define-setf-method**
psetf	**get-setf-method**
rotatef	**get-setf-method-multiple-value**

These constructs are used for assignment to simple variables and generalized variables.

Sequencing

block	**prog**
compiler-let	**prog***
flet	**prog1**
labels	**prog2**
let	**progn**
let*	**progv**
macrolet	

These constructs enable a group of statements to be executed sequentially. Some of them provide for the introduction of new variable bindings.

Iteration

loop	**dolist**
do	**dotimes**
do*	

These constructs provide facilities for definite and indefinite iteration.

Conditionals

cond	**ecase**
if	**ccase**
when	**typecase**
unless	**etypecase**
case	**ctypecase**

These conditional constructs allow selective execution of a form or groups of forms.

Control Transfer

go	**catch**
tagbody	**throw**
return	**unwind-protect**
return-from	

These constructs provide for local and nonlocal exits.

Multiple Values

multiple-value-bind	**multiple-value-setq**
multiple-value-call	**multiple-values-limit**
multiple-value-list	**values**
multiple-value-prog1	**values-list**

These constructs manipulate multiple values.

block

Purpose: The **block** special form names and evaluates a series of forms. The forms are evaluated in the order in which they are given in the argument list. The result returned by **block** is the result returned by evaluating the last of the *form* arguments.

The execution of a block may be terminated by the use of **return** or **return-from**. In this case, the value returned is that specified by the **return** or **return-from** form.

If the last *form* of the block, a **return** form, or a **return-from** form returns multiple values, those multiple values are returned by **block**. If there are no *form* arguments, **block** returns **nil**.

Syntax: **block** *name* {*form*}* [*Special Form*]

Remarks: The *name* argument is a symbol; it is not evaluated. It has lexical scope.

Examples:

```
> (block empty)
NIL
> (block foo 1 2 (return-from foo) 3 4)
NIL
> (block foo 1 2 (block bar 3 4 (return-from foo (values 5 6)) 7 8) 9 10)
5
6
```

See Also: **return**

return-from

case

Purpose: The **case** macro allows the execution of a group of forms to be dependent on selection by a key match.

The *keyform* argument is evaluated and matched against the *key* arguments; the *key* arguments are not evaluated. If the *keyform* value matches a key, then the forms associated with that key are executed in order.

The **case** macro returns the value of the last *form* executed. If no key matches or the matching key has no associated forms, **case** returns **nil**.

Syntax: **case** *keyform* {({({*key*}*) | *key*} {*form*}*)}* [*Macro*]

Remarks: A given key may appear only once. Keys are compared using **eql**.

If only one key is associated with a group of forms, it is not necessary to include that key in a list unless the key is **nil**, **t**, **otherwise**, or a cons. If **t** or **otherwise** is not enclosed in a list, it has special meaning to **case**; if **nil** is not enclosed in a list, it is treated as the empty list, not as a key.

Either the symbol **t** or the symbol **otherwise** may be used as the last key. If no other key match succeeds, the forms associated with the **t** or **otherwise** key are executed.

Examples:

```
> (dolist (k '(1 2 3 :four #\v () t 'other))
    (format t "~S "
      (case k ((1 2) 'clause1)
              (3 'clause2)
              ((nil) 'nilslot)
              ((:four #\v) 'clause4)
              ((t) 'tslot)
              (otherwise 'others))))
CLAUSE1 CLAUSE1 CLAUSE2 CLAUSE4 CLAUSE4 NILSLOT TSLOT OTHERS
NIL
```

See Also: **cond**

ecase

ccase

typecase

etypecase

ctypecase

catch

Purpose: The **catch** special form is used as the destination of a nonlocal control transfer by **throw**.

The *tag* argument is evaluated first. It serves as the name of the catch. The *form* arguments are then evaluated in order. If a **throw** occurs during the execution of one of the forms, control is transferred to the **catch** construct whose tag is eq to the *tag* argument of the **throw**. The results of the **throw** are returned as the results of the **catch**.

Catch tags have dynamic scope. If several catch tags match the *tag* argument of a **throw**, control is transferred to the most recently occurring such **catch**.

If the **catch** exits normally, the value or values returned by the last *form* are returned as the results of the **catch**. If no *form* arguments are specified, **catch** returns **nil**.

Syntax: **catch** *tag* {*form*}* [*Special Form*]

Remarks: Catch tags are compared using **eq**. Characters and numbers should therefore not be used as tags.

Examples:

```
> (catch 'foo 1 2 (throw 'foo 3) 4)
3
> (catch 'foo 1 2 3 4)
4
> (defun throw-back (tag) (throw tag t))
THROW-BACK
> (catch 'foo (throw-back 'foo) 2)
T
> (catch 'foo (catch 'foo (throw-back 'foo) 2) 3)
3
```

See Also: **throw**

compiler-let

Purpose: The **compiler-let** special form is used to create new variable bindings and to execute a series of forms that use these bindings. These variable bindings have lexical scope.

The *value* arguments are evaluated first, in the order in which they are given. The *var* arguments are then bound to the corresponding values in parallel. If no *value* is specified for a given *var* argument, that variable is bound to **nil**.

Unlike the variable bindings created by **let**, the bindings in **compiler-let** take effect during compilation instead of at run-time; no code is generated for them. The **compiler-let** construct is used for communication between macros.

If a **compiler-let** is evaluated by the Lisp interpreter, the effect is identical to that of a **let** whose variables are all declared special.

The *form* arguments are executed in order. The result returned by **compiler-let** is the value or values returned by the last form executed. If no *form* arguments are specified, **compiler-let** returns **nil**.

Syntax: **compiler-let** ({*var* | (*var value*)}*) {*form*}* [*Special Form*]

Remarks: No declarations may be specified in a **compiler-let**.

Examples:

```
> (defvar *collect-var* nil)
*COLLECT-VAR*
> (defmacro with-collecting (&body body)
    (let ((var (gensym)))
      `(compiler-let ((*collect-var* ',var))
         (let ((,var '()))
         ,@body
         (nreverse ,var)))))
WITH-COLLECTING
> (defmacro collect (value)
    (if *collect-var*
        `(push ,value ,*collect-var*)
        (error "COLLECT can only be used inside WITH-COLLECTING")))
COLLECT
> (collect 1)
>>Error: COLLECT can only be used inside WITH-COLLECTING

IF:
Original code: (IF *COLLECT-VAR* # #)
```

```
:A    Abort to Lisp Top Level
-> :a
Back to Lisp Top Level

> (with-collecting (collect 1) (collect 2) (collect 3))
(1 2 3)
```

See Also: let

cond

Purpose: The **cond** macro allows the execution of a group of forms to be dependent on a test form.

The *test* arguments are evaluated one at a time in the order in which they are given in the argument list until a *test* is found that evaluates to a non-**nil** value.

The *form* arguments associated with this test are then evaluated in order. The **cond** returns immediately after the evaluation of the last of these forms. No additional *test* or associated *form* arguments are evaluated. The **cond** returns the results of the last form evaluated. If no forms were associated with the given *test*, **cond** returns the value of the *test* argument.

If none of the *test* arguments is non-**nil**, **cond** returns **nil**.

Syntax: **cond** {(*test* {*form*}*)}* [*Macro*]

Remarks: If a test succeeds and its associated *form* argument returns multiple values, the multiple values are returned from the **cond**. Only a single value is returned in the case where a test succeeds and has no associated forms.

Examples:

```
> (defun foo ()
   (cond ((= a 1) (setq a 2))
         ((= a 2) (setq a 3))
         ((and (= a 3) (floor a 2)))
         (t (floor a 3))))
FOO
> (setq a 1)
1
> (foo)
2
> a
2
> (foo)
3
> a
3
> (foo)
1
```

```
> (setq a 5)
5
> (foo)
1
2
```

See Also: **if**

case

define-modify-macro

Purpose: The macro **define-modify-macro** is used to define a macro to access and update a generalized variable.

The arguments to the new macro will be a reference to the generalized variable, followed by the arguments that are specified in the *lambda-list* argument of **define-modify-macro**.

When the macro is invoked, the function specified by the *function* argument of **define-modify-macro** is applied to these arguments to obtain the new value, and the generalized variable is updated to contain the result.

The macro **define-modify-macro** returns *name* as its result.

Syntax: **define-modify-macro** *name lambda-list function* [*documentation*] [*Macro*]

Remarks: The *name* argument is a symbol; it is not evaluated.

The *function* argument is not evaluated; it should be the name of a function.

The lambda list may contain the **&optional** and **&rest** keywords only.

A documentation string may be attached to the name of the new macro by the optional *documentation* argument; the documentation type for this string is **setf**.

Examples:

```
> (define-modify-macro appendf (&rest args) append "Append onto list")
APPENDF
> (setq x '(a b c) y x)
(A B C)
> (appendf x '(d e f) '(1 2 3))
(A B C D E F 1 2 3)
> x
(A B C D E F 1 2 3)
> y
(A B C)
```

See Also: **defsetf**

define-setf-method

define-setf-method

Purpose: The macro **define-setf-method** is used to specify the means by which **setf** is to update a generalized variable that is referenced by a given access function.

When **setf** is given a generalized variable that is specified in terms of this access function and a new value for the variable, it is expanded into a call on the update function. The arguments of the access function and the new value are passed to the update function, and the update function is invoked to modify the value of the variable.

The *lambda-list* argument specifies the arguments of the access function. When **setf** is called with the access function, the lambda list parameters are bound to the corresponding access function arguments in the call form.

The *form* arguments must compute the expansion for a call on **setf** that references the generalized variable by means of the given access function.

The evaluation of the *form* arguments must result in the following five values: a list of the temporary variables used; a list of the value forms to whose values the temporary variables are bound; a list consisting of the store variable (the temporary variable that is bound to the new value); the store form (the form that is used to update the generalized variable and return the resulting value); and the access form (the form that is used to access and return the value of the generalized variable).

The **define-setf-method** macro returns the name of the access function as its result.

Syntax: **define-setf-method** *access-fn lambda-list* {*declaration* | *documentation*}* {*form*}* [*Macro*]

Remarks: The *access-fn* argument is the name of a function or macro; it is not evaluated.

A documentation string may be attached to the name of the new macro by the optional *documentation* argument; the documentation type for this string is **setf**.

define-setf-method

Examples:

```
> (defun lastguy (x) (car (last x)))
LASTGUY
> (define-setf-method lastguy (x)
     "Set the last element in a list to the given value."
     (multiple-value-bind (dummies vals newval setter getter)
       (get-setf-method x)
       (let ((store (gensym)))
         (values dummies
         vals
         `(,store)
         `(progn (rplaca (last ,getter) ,store) ,store)
         `(lastguy ,getter)))))
LASTGUY
> (setq a (list 'a 'b 'c 'd)
        b (list 'x)
        c (list 1 2 3 (list 4 5 6)))
(1 2 3 (4 5 6))
> (setf (lastguy a) 3)
3
> (setf (lastguy b) 7)
7
> (setf (lastguy (lastguy c)) 'foo)
FOO
> a
(A B C 3)
> b
(7)
> c
(1 2 3 (4 5 FOO))
```

See Also:

setf

defsetf

get-setf-method

defsetf

Purpose: The macro **defsetf** is used to specify the means by which **setf** is to update a generalized variable that is referenced by a given access function. It specifies an update function that is to be used in conjunction with the given access function.

When **setf** is given a generalized variable that is specified in terms of this access function and a new value for the variable, it is expanded into a call on the update function. The arguments of the access function and the new value are passed to the update function, and the update function is invoked to modify the value of the variable.

The **defsetf** macro returns the name of the access function as its result.

The arguments to **defsetf** include the access function and either the name of an update function or a body of code that will expand the **setf** call, update the given location, and return the new value that was stored.

In the first of these methods, an update function is specified by use of the *update-fn* argument. The *update-fn* argument is the name of a function or macro; it is not evaluated. The update function must take one more argument than the access function. This last argument corresponds to the new value that is to be assigned to the generalized variable. The update function must return the new value as its result.

In the second method, the *form* arguments must compute the expansion for a call on **setf** that references the generalized variable by means of the given access function. This expansion must also return the new value assigned to the variable as its result.

The *lambda-list* argument specifies the arguments of the access function. The *store-variable* corresponds to the value that is to be used to update the generalized variable.

The forms in the body may assume that the lambda list parameters and the store variable are bound to the corresponding arguments in the call to **setf**. When the forms in the body are evaluated, the lambda list parameters and the store variable are actually bound to the names of temporary variables, which, when **setf** is expanded, are bound to the actual argument values.

Syntax: defsetf *access-fn* { *update-fn* [*documentation*] | *lambda-list* (*store-variable*) {*declaration* | *documentation*}* {*form*}*} [*Macro*]

Remarks: The *access-fn* argument is the name of a function or macro; it is not evaluated. The access function must be a function or a macro that evaluates all of its arguments.

The *lambda-list* argument may use the **&optional**, **&rest**, **&key** keywords, default values, and *supplied-p* parameters.

A documentation string may be attached to the name of the new macro by the optional *documentation* argument; the documentation type for this string is **setf**.

Examples:

```
> (defun middleguy (x) (nth (truncate (1- (list-length x)) 2) x))
MIDDLEGUY
> (defun set-middleguy (x v)
     (unless (null x)
       (rplaca (nthcdr (truncate (1- (list-length x)) 2) x) v)
       v))
SET-MIDDLEGUY
> (defsetf middleguy set-middleguy)
MIDDLEGUY
> (setq a (list 'a 'b 'c 'd)
        b (list 'x)
        c (list 1 2 3 (list 4 5 6) 7 8 9))
(1 2 3 (4 5 6) 7 8 9)
> (setf (middleguy a) 3)
3
> (setf (middleguy b) 7)
7
> (setf (middleguy (middleguy c)) 'foo)
FOO
> a
(A 3 C D)
> b
(7)
> c
(1 2 3 (4 FOO 6) 7 8 9)
```

See Also:

setf

define-setf-method

get-setf-method

do, do*

Purpose: The **do** macro is used to iterate over a group of forms while a test condition holds.

It provides for a series of local iteration variables that may be stepped each time through the iteration loop.

An initial value may be specified for each iteration variable by use of the *init* form. The *init* forms are all evaluated first. The iteration variables are then bound in parallel to the corresponding values. If an *init* form is not specified for a given variable, that variable is bound to **nil**.

The *step* form arguments may be used to specify how the variables should be updated on succeeding iterations through the loop. The *step* forms are all evaluated, and then the iteration variables are bound in parallel to the corresponding values. If a *step* form is not specified for a given variable, that variable is not stepped.

The *end-test* form is evaluated at the beginning of each iteration. The **do** terminates when the result of *end-test* is non-**nil**. It is only when *end-test* results in a non-**nil** value that the forms associated with the end test are evaluated. They are evaluated in order. The **do** then returns the value of the last of these forms. If no such forms are specified, **do** returns **nil**.

The body of the **do** is like a **tagbody**. It consists of a series of tags and statements. The *tag* and *statement* arguments are processed in the order in which they occur. The *tag* arguments are not evaluated; they must be symbols or integers. The tags serve the purpose of labeling statements and have lexical scope. The *statement* forms are evaluated. The **go** special form may be used within the body of the **do** to transfer control to a statement labeled by a tag.

The **do*** macro is identical to **do** except that the iteration variables are bound to the initial values and the values of the *step* forms sequentially. A variable may thus refer to the value to which a variable occurring earlier in the variables list has just been bound.

Syntax:

do ({*var* | (*var* [*init* [*step*]])}*) (*end-test* {*form*}*) {*declaration*}* {*tag* | *statement*}* [*Macro*]

do* ({*var* | (*var* [*init* [*step*]])}*) (*end-test* {*form*}*) {*declaration*}* {*tag* | *statement*}* [*Macro*]

Remarks: Declarations may be specified for the iteration variables, the *init* and *step* forms, the *end-test*, the *result* form, and statements in the body of the do construct.

If a declaration is specified for a variable, the initial value of that variable must be consistent with the declaration.

A block with the name of **nil** encloses the do construct. The **return** macro may thus be used to exit from the do.

Examples:

```
> (do ((foo1 1 (1+ foo1))
       (foo2 0 (1- foo2)))
      ((> (- foo1 foo2) 5) foo1))
4
> (do ((foo1 1 (1+ foo1))
       (foo2 0 (1+ foo1)))
      ((= 3 foo2) foo1))
3
> (do* ((foo1 1 (1+ foo1))
        (foo2 0 (1+ foo1)))
       ((= 3 foo2) foo1))
2
```

See Also:

dolist

dotimes

loop

tagbody

return

dolist

Purpose: The macro **dolist** is used to iterate over the elements of a list.

The *listform* argument is evaluated first; it should result in a list. The variable *var* is bound to each element of the list in turn, and the body of **dolist** is executed for that element.

When all the list elements have been processed, the *result* form is evaluated, and its value is returned as the result of the **dolist**. If a *result* form is not specified, **dolist** returns **nil**.

The body of the **dolist** is like a **tagbody**. It consists of a series of tags and statements. The *tag* and *statement* arguments are processed in the order in which they occur. The *tag* arguments are not evaluated; they must be symbols or integers. The tags serve the purpose of labeling statements and have lexical scope. The *statement* forms are evaluated. The go special form may be used within the body of the **dolist** to transfer control to a statement labeled by a tag.

Syntax: **dolist** (*var listform* [*result*]) {*declaration*}* {*tag* | *statement*}* [*Macro*]

Remarks: At the time the *result* form is processed, *var* is bound to **nil**.

A block with the name of **nil** encloses the **dolist** construct. The **return** macro may thus be used to exit from the **dolist**.

Examples:

```
> (setq foo2 '())
NIL
> (dolist (foo1 '(1 2 3 4) foo2) (push foo1 foo2))
(4 3 2 1)
> (setq foo2 0)
0
> (dolist (foo1 '(1 2 3 4)) (incf foo2))
NIL
> foo2
4
```

See Also: **do**

dotimes

Purpose: The **dotimes** macro is used to iterate over a fixed number of integer values.

The *countform* argument is evaluated first; it should result in an integer. The variable *var* is bound in turn to each integer from 0 up to but not including the value of *countform*, and the body of the **dotimes** is executed for that value. When all such integer values have been processed, the *result* form is evaluated, and its value is returned as the result of the **dotimes**. If a *result* form is not specified, **dotimes** returns **nil**.

The body of the **dotimes** is like a **tagbody**. It consists of a series of tags and statements. The *tag* and *statement* arguments are processed in the order in which they occur. The *tag* arguments are not evaluated; they must be symbols or integers. The tags serve the purpose of labeling statements and have lexical scope. The *statement* forms are evaluated. The go special form may be used within the body of the **dotimes** to transfer control to a statement labeled by a tag.

Syntax: **dotimes** (*var countform* [*result*]) {*declaration*}* {*tag* | *statement*}* [*Macro*]

Remarks: At the time the *result* form is processed, *var* is bound to the number of times the body was executed.

A block with the name of **nil** encloses the **dotimes** construct. The **return** macro may thus be used to exit from the **dotimes**.

If the *countform* argument is zero or negative, the body is not executed.

Examples:

```
> (dotimes (fool 10 fool))
10
> (setq foo2 0)
0
> (dotimes (fool 10 t) (incf foo2))
T
> foo2
10
```

See Also: **do**

ecase, ccase

Purpose: The **ecase** and **ccase** macros allow the execution of a group of forms to be dependent on selection by a key match.

The **ecase** macro evaluates its *keyform* argument and matches it against the *key* arguments; the *key* arguments are not evaluated. If the *keyform* value matches a key, the forms associated with that key are executed in order.

The **ecase** macro returns the value of the last *form* executed. If the matching key has no associated forms, **ecase** returns **nil**. If there is no matching key, **ecase** signals a fatal error.

The **ccase** macro matches the value contained in its *keyplace* argument against the *key* arguments; the *key* arguments are not evaluated. If the value in *keyplace* matches a key, the forms associated with that key are executed in order.

If the object in *keyplace* does not match any of the keys, **ccase** signals a continuable error and enters the debugger. If the user continues from this error, **ccase** prompts for a new value to store in *keyplace* and tries the key matching again.

The macro **ccase** returns the value of the last *form* executed. If the matching key has no associated forms, **ccase** returns **nil**.

Syntax:

ecase *keyform* {({({*key*}*) | *key*} {*form*}*)}* [*Macro*]

ccase *keyplace* {({({*key*}*) | *key*} {*form*}*)}* [*Macro*]

Remarks: A given key may appear only once. Keys are compared using **eql**.

If only one key is associated with a group of forms, it is not necessary to include that key in a list unless the key is **nil**, **t**, **otherwise**, or a cons.

Examples:

```
> (setq k 'foo)
FOO
> (ecase k ((foo bar) (setq k 3000))
          (2000 (setq k 'bar)))
3000
```

See Also: **case**

etypecase, ctypecase

Purpose: The **etypecase** and **ctypecase** macros allow the execution of a group of forms to be dependent on selection by a type match.

The **etypecase** macro evaluates its *keyform* argument and matches it against the *type* arguments in turn. The *type* arguments must be type specifiers; they are not evaluated. If the object specified by the *keyform* argument is an instance of a given type, then the forms associated with that type are executed in order. If the object is an instance of more than one such type, only the forms associated with the first of these types are executed.

The **etypecase** macro returns the value of the last *form* executed. If the matching type has no associated forms, it returns **nil**. If no type matches, **etypecase** signals a fatal error.

The **ctypecase** macro matches the value contained in its *keyplace* argument against the *type* arguments in turn. The *type* arguments must be type specifiers; they are not evaluated. If the object contained in the *keyplace* argument is an instance of a given type, then the forms associated with that type are executed in order. If the object is an instance of more than one such *type*, only the forms associated with the first of these types are executed.

If the object in *keyplace* does not match any of the types, **ctypecase** signals a continuable error and enters the debugger. If the user continues from this error, **ctypecase** prompts for a new value to store in *keyplace* and tries the type matching again.

The **ctypecase** macro returns the value of the last *form* executed. If the matched type has no associated forms, **ctypecase** returns **nil**.

Syntax:

etypecase *keyform* {(*type* {*form*}*)}* [*Macro*]

ctypecase *keyplace* {(*type* {*form*}*)}* [*Macro*]

Remarks: The *type* arguments are not evaluated.

It is not permitted to use **t** or **otherwise** as a *type* argument.

Examples:

```
> (etypecase nil (cons "it's a cons")
                 (list "it's nil")
                 (symbol "it's a symbol"))
"it's nil"
```

See Also: **typecase**

flet

Purpose: The special form **flet** is used to define functions whose names are meaningful only locally and to execute a series of forms with these function definition bindings. Any number of such local functions may be defined.

The names of functions defined by **flet** have lexical scope; they retain their local definitions only within the body of the **flet**. Any references within the body of the **flet** to functions whose names are the same as those defined within the **flet** are thus references to the local functions instead of to any global functions of the same names. The scope of these function definition bindings, however, includes only the body of **flet**, not the definitions themselves. Within the function definitions, local function names that match those being defined refer to global functions defined outside the **flet**. It is thus not possible to define recursive functions with **flet**.

The *form* arguments are executed in order. The result returned by **flet** is the value or values returned by the last form executed. If no *form* arguments are specified, **flet** returns **nil**.

Syntax: **flet** ({ (*name lambda-list* {*declaration* | *documentation*}* {*form*}*)}*) {*form*}* [*Special Form*]

Remarks: An **flet** local function definition is identical in form to the function definition part of a **defun**. It contains a name, argument list, optional declarations and documentation string, and a body.

Examples:

```
> (flet ((flet1 (n) (+ n n)))
    (flet ((flet1 (n) (+ 2 (flet1 n))))
      (flet1 2)))
6
> (flet ((+ (&rest args) 'crossed-out))
    (+ 1 2 3))
CROSSED-OUT
```

See Also: **labels**

let

defun

get-setf-method, get-setf-method-multiple-value

Purpose: The function **get-setf-method** returns a multiple value result that characterizes the **setf** method for a given form.

The result consists of the following five values: a list of the temporary variables; a list of the value forms to whose values the temporary variables are bound; a list consisting of the store variable; the store form; and the access form.

The function **get-setf-method-multiple-value** is like **get-setf-method** except that a list containing more than one store variable may be returned.

Syntax: **get-setf-method** *form* [*Function*]

get-setf-method-multiple-value *form* [*Function*]

Remarks: The *form* argument must be a reference to a generalized variable.

Examples:

```
> (get-setf-method 'x)
NIL
NIL
(#:G50)
(SETQ X #:G50)
X
> (define-setf-method multivalue (x)
      (values '() '() '(,(gensym) ,(gensym)) '(setq ,x 3) '4))
MULTIVALUE
> (get-setf-method-multiple-value '(multivalue foo))
NIL
NIL
(#:G59 #:G60)
(SETQ FOO 3)
4
```

See Also: **defsetf**

define-setf-method

setf

go

Purpose: The go special form is used to transfer control to a location within a **tagbody**.

Control is transferred to the statement labeled by a tag that is **eql** to the *tag* argument. Tags have lexical scope. If several tags match the *tag* argument of the go, control is transferred to whichever matching tag is contained in the **tagbody** form that most immediately contains the go.

No value is returned.

Syntax: go *tag* [*Special Form*]

Remarks: The *tag* argument is not evaluated. It must be a symbol or an integer.

It is an error if there is no matching tag.

Examples:

```
> (tagbody
   (setq val 2)
   (go lp)
   (incf val 3)
   lp (incf val 4))
NIL
> val
6
```

See Also: **tagbody**

if

Purpose: The **if** special form allows the execution of a form to be dependent on a single test form.

First, the *test* argument is evaluated before either the *then* or the *else* argument. Next, either the *then* or the *else* argument is evaluated, depending on the result of *test.*

If the *test* argument is non-**nil**, the *then* form is evaluated. The results of the *then* form are returned as the results of **if**.

If the *test* argument is **nil**, the *else* form is evaluated. The results of the *else* form are returned as the results of **if**. If an *else* argument is not specified, **nil** is returned.

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

Examples:

```
> (if t 1)
1
> (if nil 1 2)
2
```

See Also:

and

when

unless

or

labels

Purpose: The special form **labels** is used to define functions whose names are meaningful only locally and to execute a series of forms with these function definition bindings. Any number of such local functions may be defined.

The names of functions defined by **labels** have lexical scope; they retain their local definitions only within the body of the **labels** construct. Any references within the body of the **labels** construct to functions whose names are the same as those defined within the **labels** form are thus references to the local functions instead of to any global functions of the same names. The scope of these function definition bindings includes the definitions themselves as well as the body of the **labels** construct.

The *form* arguments are executed in order. The result returned by **labels** is the value or values returned by the last form executed. If no *form* arguments are specified, **labels** returns **nil**.

Syntax: **labels** ({ (*name lambda-list* {*declaration* | *documentation*}* {*form*}*)}*) {*form*}* [*Special Form*]

Remarks: A **labels** local function definition is identical in form to the function definition part of a **defun**. It contains a name, argument list, optional declarations and documentation string, and a body.

Examples:

```
> (defun recursive-times (k n)
    (labels ((foo (n) (if (zerop n) 0 (+ k (foo (1- n))))))
      (foo n)))
RECURSIVE-TIMES
> (recursive-times 2 3)
6
```

See Also: **flet**

let

defun

let, let*

Purpose: The **let** special form is used to create new variable bindings and to execute a series of forms that use these bindings.

The variable bindings created are lexical bindings unless the appropriate **special** declarations are specified. The bindings have lexical scope.

The *value* arguments are evaluated first, in the order in which they are given. The *var* arguments are then bound to the corresponding values in parallel. If no *value* is specified for a given *var* argument, that variable is bound to **nil**.

The *form* arguments are executed in order. The result returned by **let** is the value or values returned by the last form executed. If no *form* arguments are specified, **let** returns **nil**.

The special form **let*** is identical to **let** except that the variables are bound to the values sequentially. A variable may thus refer to the value to which a variable occurring earlier in the variables list has just been bound.

Syntax:

let ({*var* | (*var value*)}*) {*declaration*}* {*form*}* [*Special Form*]

let* ({*var* | (*var value*)}*) {*declaration*}* {*form*}* [*Special Form*]

Remarks: If a declaration is specified for a variable, the initial value of that variable must be consistent with the declaration.

Examples:

```
> (setq a 'top)
TOP
> (defun foo () a)
FOO
> (let ((a 'inside) (b a))
    (format t "~S ~S ~S" a b (foo)))
INSIDE TOP TOP
NIL
> (let* ((a 'inside) (b a))
    (format t "~S ~S ~S" a b (foo)))
INSIDE INSIDE TOP
NIL
> (let ((a 'inside) (b a))
    (declare (special a))
    (format t "~S ~S ~S" a b (foo)))
INSIDE TOP INSIDE
NIL
```

loop

Purpose: The **loop** macro is used to perform indefinite iteration.

Each *form* argument is evaluated in turn. After the last form is evaluated, the evaluation starts over again with the first. The only way to exit from a loop is by explicit termination, as by a **return**, **go**, or **throw**.

An implicit block named **nil** is created by the **loop** construct. It is thus possible to return a value from **loop** by the use of **return**.

Syntax: **loop** {*form*}* [*Macro*]

Examples:

```
> (let ((i 0))
    (loop (incf i) (if (= i 3) (return i))))
3
> (let ((i 0)(j 0))
    (tagbody
      (loop (incf j 3) (incf i) (if (= i 3) (go exit)))
      exit)
    j)
9
```

See Also:

do

dolist

dotimes

return

go

throw

macrolet

Purpose: The **macrolet** special form is used to define macros whose names are meaningful only locally and to execute a series of forms with these macro definition bindings. Any number of such local macros may be defined.

The names of macros defined by **macrolet** have lexical scope; they retain their local definitions only within the body of the **macrolet**. Any references within the body of the **macrolet** to macros whose names are the same as those defined within the **macrolet** are thus references to the local macros instead of to any global macros of the same names. The scope of these macro definition bindings, however, includes only the body of **macrolet**, not the definitions themselves. Within the macro definitions, local function names that match those being defined refer to global macros or functions defined outside the **macrolet**.

The *form* arguments are executed in order. The result returned by **macrolet** is the value or values returned by the last form executed. If no *form* arguments are specified, **macrolet** returns **nil**.

Syntax: **macrolet** ({ (*name lambda-list* {*declaration* | *documentation*}* {*form*}*)}*) {*form*}* [*Special Form*]

Remarks: The macro expansion functions defined by **macrolet** are defined in the global environment, not in the lexical environment of the **macrolet**; they thus do not have access to items within the lexical scope of the **macrolet**.

A **macrolet** local macro definition is identical in form to the macro definition part of a **defmacro**. It contains a name, argument list, optional declarations and documentation string, and a body.

Examples:

```
> (defmacro mlets (x &environment env)
    (let ((form '(baz ,x)))
      (macroexpand form env)))
MLETS
> (macrolet ((baz (z) '(+ ,z ,z))) (mlets 5))
10
```

See Also:

flet

let

defmacro

multiple-value-bind

Purpose: The **multiple-value-bind** macro is used to create new variable bindings and to execute a series of forms that use these bindings.

The variable bindings created are lexical bindings unless the appropriate special declarations are specified. The bindings have lexical scope.

The *values-form* argument is evaluated first. The *var* arguments are then bound to the values that it returns. If there are more variables than results, the remaining variables are bound to **nil**. If there are more results than variables, the remaining values are discarded.

The *form* arguments are executed in order. The value returned by **multiple-value-bind** is the value or values returned by the last form executed. If no *form* arguments are specified, **multiple-value-bind** returns **nil**.

Syntax: **multiple-value-bind** ({*var*}*) *values-form* {*declaration*}* {*form*}* [*Macro*]

Remarks: If a declaration is specified for a variable, the value to which that variable is bound must be consistent with the declaration.

Examples:

```
> (multiple-value-bind (f r) (floor 130 11) (list f r))
(11 9)
```

See Also: **let**

multiple-value-call

Purpose: The **multiple-value-call** special form applies a function to the values collected from groups of multiple values.

The *function* argument is evaluated first. All of the *form* arguments are then evaluated. The values they produce are passed to the function as arguments. The result of applying the function to these arguments is returned as the result of the **multiple-value-call** form.

Syntax: **multiple-value-call** *function* {*form*}* [*Special Form*]

Examples:

```
> (multiple-value-call #'list 1 '/ (values 2 3) '/ (values) '/ (floor 2.5))
(1 / 2 3 / / 2 .5)
```

multiple-value-list

Purpose: The **multiple-value-list** macro returns as a list the multiple values that are produced as a result of evaluating a given form.

Syntax: **multiple-value-list** *form* [*Macro*]

Examples:

```
> (multiple-value-list (values 1 2 3))
(1 2 3)
```

See Also: **values-list**

multiple-value-prog1

Purpose: The **multiple-value-prog1** special form evaluates a series of forms. The forms are evaluated in the order in which they are given in the argument list. The multiple values returned by **multiple-value-prog1** are the results returned by evaluating the first of the *form* arguments.

Syntax: **multiple-value-prog1** *form* {*form*}* [*Special Form*]

Examples:

```
> (setq foo '(1 2 3))
(1 2 3)
> (multiple-value-prog1
    (values-list foo)
    (setq foo nil)
    (values-list foo))
1
2
3
```

See Also: **prog1**

multiple-value-setq

Purpose: The macro **multiple-value-setq** is used to assign values to a list of variables.

The *form* argument is evaluated first. The values it returns are then assigned to the corresponding variables of the list.

If there are more variables than results, **nil** is assigned to the remaining variables. If there are more results than variables, the remaining values are discarded.

The result of **multiple-value-setq** is the first value returned by *form*. If *form* returns no values, the result is **nil**.

Syntax: **multiple-value-setq** *vars form* [*Macro*]

Remarks: The *vars* argument is a list of variables.

Examples:

```
> (multiple-value-setq (a b c) (values 1 2))
1
> a
1
> b
2
> c
NIL
```

See Also: **setq**

multiple-values-limit

Purpose: The constant **multiple-values-limit** defines the upper exclusive bound on the number of values that any function may return.

The value of **multiple-values-limit** in Sun Common Lisp is 2^9.

Syntax: **multiple-values-limit** [*Constant*]

Examples:

```
> multiple-values-limit
512
```

prog, prog*

Purpose: The **prog** macro is used to create a block, new variable bindings, and an implicit tagbody, and to execute a series of forms that use these items.

The variable bindings created are lexical bindings unless the appropriate **special** declarations are specified.

The *init* arguments are evaluated first, in the order in which they are given. The *var* arguments are then bound to the corresponding values in parallel. If no *init* value is specified for a given *var* argument, that variable is bound to **nil**.

The body of the **prog** is like a **tagbody**. It consists of a series of tags and statements. The *tag* and *statement* arguments are processed in the order in which they occur. The *tag* arguments are not evaluated; they must be symbols or integers. The tags serve the purpose of labeling statements. The *statement* forms are evaluated. The **go** special form may be used within a **prog** body to transfer control to a statement labeled by a tag. The tags have lexical scope.

A block with the name of **nil** encloses the **prog** construct. The **return** macro may thus be used to exit from the **prog**.

The **prog*** macro is identical to **prog** except that variables are bound to the initial values sequentially. A variable may thus refer to the value to which a variable occurring earlier in the variables list has just been bound.

The **prog** and **prog*** macros return **nil**.

Syntax:

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

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

Remarks: If a declaration is specified for a variable, the initial value of that variable must be consistent with the declaration.

Examples:

```
> (setq a 1)
1
> (prog ((a 2) (b a)) (return (if (= a b) '= '/=)))
/=
> (prog* ((a 2) (b a)) (return (if (= a b) '= '/=)))
=
> (prog () 'no-return-value)
NIL
```

prog, prog*

See Also: **block**

let

tagbody

go

return

prog1

Purpose: The **prog1** macro evaluates a series of forms. The forms are evaluated in the order in which they are given in the argument list. The result returned by **prog1** is the result returned by evaluating the *first* form argument.

Syntax: **prog1** *first* {*form*}* [*Macro*]

Remarks: If the *first* form returns multiple values, only the first of these values is returned by **prog1**. If the *first* form returns no values, **prog1** returns **nil**.

Examples:

```
> (setq foo1 1)
1
> (prog1 foo1 (setq foo1 nil))
1
> foo1
NIL
> (prog1 (values t t))
T
```

See Also:

progn

prog2

multiple-value-prog1

prog2

Purpose: The **prog2** macro evaluates a series of forms. The forms are evaluated in the order in which they are given in the argument list. The result returned by **prog2** is the result returned by evaluating the *second* form argument.

Syntax: **prog2** *first second {form}** [*Macro*]

Remarks: If the *second* form returns multiple values, only the first of these values is returned by **prog2**. If the *second* form returns no values, **prog2** returns **nil**.

Examples:

```
> (setq foo t)
T
> (prog2 nil foo (setq foo nil))
T
> foo
NIL
> (prog2 (cons 'x 'y) (values t t))
T
```

See Also: **progn**

prog1

progn

Purpose: The **progn** special form evaluates a series of forms. The forms are evaluated in the order in which they are given in the argument list. The result returned by **progn** is the result returned by evaluating the last of the *form* arguments. If the last *form* returns multiple values, those multiple values are returned as the result of **progn**. If no *form* arguments are specified, **progn** returns **nil**.

Syntax: **progn** {*form*}* [*Special Form*]

Examples:

```
> (progn)
NIL
> (progn 1 2 3)
3
> (progn (values 1 2 3))
1
2
3
> (setq a 1)
1
> (if a
      (progn (setq a nil) 'true)
      (progn (setq a t) 'false))
TRUE
```

progv

Purpose: The **progv** special form is used to create new dynamic variable bindings and to execute a series of forms that use those bindings.

The *symbols* argument specifies a list of dynamic variables. The *values* argument specifies a list of values. The *symbols* and *values* arguments are evaluated, and the variables are bound to the corresponding values. If there are more variables than values, the remaining variables are unbound. If there are more values than variables, the remaining values are discarded.

The *form* arguments are executed in order. The value returned by **progv** is the value or values returned by the last form executed. If no *form* arguments are specified, **progv** returns **nil**.

Syntax: **progv** *symbols values {form}** [*Special Form*]

Remarks: The previous bindings of the dynamic variables are restored when **progv** exits.

Examples:

```
> (setq *x* 1)
1
> (progv '(*x*) '(2) *x*)
2
> *x*
1
> (let ((*x* 3)) (progv '(*x*) '(4) (list *x* (symbol-value '*x*))))
(3 4)
```

See Also: **progn**

return, return-from

Purpose: The **return** and **return-from** constructs are used to return from a lexically enclosing block. Both **return** and **return-from** specify the value or values to be returned from the block.

The *name* argument of **return-from** is a symbol; it is not evaluated. It specifies the lexically enclosing block of the same name from which to return.

The **return** macro is like **return-from** except that it causes a return from the lexically enclosing block whose name is **nil**. Such implicit blocks are created by **prog** and the iteration constructs **do**, **do***, **dolist**, **dotimes**, and **loop**.

Syntax:

return [*result*] — [*Macro*]

return-from *name* [*result*] — [*Special Form*]

Remarks: If the *result* argument is not specified, **nil** is returned.

Examples:

```
> (block foo 1 (return-from foo 2) 3)
2
> (let ((a 0))
    (dotimes (i 10) (incf a) (when (oddp i) (return)))
    a)
2
> (defun foo (x)
    (if x (return-from foo 'bar))
    44)
FOO
> (foo nil)
44
> (foo t)
BAR
```

See Also: **block**

rotatef

Purpose: The **rotatef** macro is used to modify the values of a series of generalized variables by rotating values from one generalized variable into another.

First the values of all the *place* arguments are obtained. The location specified by each *place* argument is then assigned the value corresponding to the argument that follows it in the argument list. The location specified by the last *place* argument is assigned the original value of the first argument.

The **rotatef** macro returns **nil** as its result.

Syntax: **rotatef** {*place*}* [*Macro*]

Remarks: The *place* arguments must be generalized variables acceptable to the macro **setf**.

The **rotatef** macro may be used to assign values to lexical as well as to dynamic variables.

Examples:

```
> (let ((n 0)
        (x (list 'a 'b 'c 'd 'e 'f 'g)))
    (rotatef (nth (incf n) x)
             (nth (incf n) x)
             (nth (incf n) x))
    x)
(A C D B E F G)
```

See Also: **setf**

shiftf

set

Purpose: The function **set** is used to modify the value of a special variable.

It causes the dynamic variable associated with the *symbol* argument to have the specified *value*.

Syntax: **set** *symbol value* [*Function*]

Examples:

```
> (set 'foo 1)
1
> foo
1
```

See Also: **setq**

symbol-value

setf, psetf

Purpose: The **setf** macro is used to update a generalized variable. It modifies the location specified by the *place* argument to contain *newvalue*.

More than one generalized variable may be updated in a single **setf**. In this case the pairs of *place* and *newvalue* arguments are processed sequentially.

The result returned by **setf** is the value of the last *newvalue* argument. If no arguments are given, **setf** returns **nil**.

The macro **psetf** is like **setf** except that if multiple argument pairs are specified, the updates are done in parallel. The result returned by **psetf** is **nil**.

Syntax: **setf** {*place newvalue*}* [*Macro*]

psetf {*place newvalue*}* [*Macro*]

Remarks: The macros **setf** and **psetf** may be used to assign values to lexical as well as to dynamic variables.

The *place* arguments must be generalized variables.

If more than one of the *place* forms given to **psetf** evaluates to the same location, the results are unpredictable.

Examples:

```
> (setq x (cons 'a 'b) y (list 1 2 3))
(1 2 3)
> (setf (car x) 'x (cadr y) 'foo (cdr x) y (cadr y) 'bar)
BAR
> (setf (third y) 7)
7
> x
(X 1 BAR 7)
> y
(1 BAR 7)
```

setq, psetq

Purpose: The **setq** and **psetq** constructs are used to assign values to variables.

The special form **setq** evaluates its *form* arguments sequentially. The first *form* argument is evaluated, and its value is stored in the variable specified by the first *var* argument before the next *form* argument is evaluated. Hence, if a *form* references a variable in the argument list whose value has already been modified, the new value of the variable is used.

The macro **psetq** is like **setq** except that the *form* arguments are all evaluated in parallel, and the resulting values are stored in parallel in the *var* arguments.

The result returned by **setq** is the result returned by evaluating the last of the *form* arguments. If this form produces multiple values, only the first value is returned; if the form produces no values, **setq** returns **nil**.

The **psetq** macro returns **nil**.

Syntax: **setq** {*var form*}* [*Special Form*]

psetq {*var form*}* [*Macro*]

Remarks: The **setq** and **psetq** constructs may be used to assign values to both special and lexical variables.

Examples:

```
> (setq foo 1)
1
> foo
1
```

See Also: **setf**

set

shiftf

Purpose: The **shiftf** macro is used to modify the values of a series of generalized variables by shifting values from one generalized variable into another.

First the values of all the *place* arguments and the value specified by ***newvalue*** are obtained. The location specified by each *place* argument is then assigned the value corresponding to the argument that follows it in the argument list.

The original value of the first *place* argument is returned as the result of **shiftf**.

Syntax: **shiftf** $\{place\}^+$ ***newvalue*** [*Macro*]

Remarks: The *place* arguments must be generalized variables acceptable to the macro **setf**.

The **shiftf** macro may be used to assign values to lexical as well as dynamic variables.

Examples:

```
> (setq x '(1 2 3) y 'trash)
TRASH
> (shiftf y x (cdr x) '(hi there))
TRASH
> x
(2 3)
> y
(1 HI THERE)
```

See Also: **setf**

rotatef

tagbody

Purpose: The **tagbody** special form provides for control transfers within a body of code by means of statement labels called tags.

The tagbody consists of a series of tags and statements. The *tag* and *statement* arguments are processed in the order in which they occur. The *tag* arguments are not evaluated; they must be symbols or integers. The tags serve the purpose of labeling statements. The *statement* forms are evaluated. The **go** special form may be used within a tagbody to transfer control to a statement labeled by a tag.

Tags have lexical scope.

The **tagbody** special form returns **nil**.

Syntax: **tagbody** {*tag* | *statement*}* [*Special Form*]

Remarks: The forms **do**, **do***, **dolist**, **dotimes**, **prog**, and **prog*** all have implicit tagbodies.

Examples:

```
> (let (val)
    (tagbody
      (setq val 1)
      (go point-a)
      (incf val 16)
     point-c
      (incf val 04)
      (go point-b)
      (incf val 32)
     point-a
      (incf val 02)
      (go point-c)
      (incf val 64)
     point-b
      (incf val 08))
    val)
15
```

See Also: go

throw

Purpose: The **throw** special form is used to cause a nonlocal control transfer.

Both the *tag* and the *result* arguments are evaluated. Control is transferred to the **catch** construct whose tag is **eq** to the *tag* argument. The results of **throw** are returned as the results of the **catch** form.

Catch tags have dynamic scope. If several catch tags match the *tag* argument of **throw**, control is transferred to the most recently occurring **catch** form.

Syntax: **throw** *tag result* [*Special Form*]

Remarks: The **throw** special form may return multiple values.

The successful execution of the **throw** form causes the stack to be unwound and any dynamic variable bindings to be restored to their state as of the point of the catch. Any intervening **unwind-protect** code is executed during this process.

There must be a matching tag; otherwise the stack is not unwound and an error is signaled.

Since catch tags are compared using **eq**, characters and numbers should not be used as tags.

Examples:

```
> (catch 'foo
    (setq i 0)
    (loop (incf i) (when (> i 10) (throw 'foo 'bar)))
    i)
BAR
```

See Also: **catch**

unwind-protect

typecase

Purpose: The **typecase** macro allows the execution of a group of forms to be dependent on selection by a type match.

The ***keyform*** argument is evaluated and matched against the ***type*** arguments in turn. The ***type*** arguments must be type specifiers; they are not evaluated. If the object specified by the ***keyform*** argument is an instance of a given type, then the forms associated with that type are executed in order. If the object is an instance of more than one such type, only the forms associated with the first of these types are executed.

The **typecase** macro returns the value of the last *form* executed. If no type matches or the matching type has no associated forms, **typecase** returns **nil**.

Syntax: **typecase** *keyform* {(*type* {*form*}*)}* [*Macro*]

Remarks: Either the symbol **t** or the symbol **otherwise** may be used as the last type specifier. If no other type match succeeds, the forms associated with the **t** or **otherwise** are executed.

Examples:

```
> (typecase '(a b)
     (integer "integer")
     (list    "list")
     (t       "otherwise"))
"list"
> (typecase 'a
     (integer   "integer")
     (list      "list")
     (otherwise "otherwise"))
"otherwise"
```

See Also:

etypecase

ctypecase

case

ecase

ccase

cond

unless

Purpose: The **unless** macro allows the execution of a series of forms to be dependent on a single test form.

If the *test* argument is non-**nil**, none of the *form* arguments are evaluated, and **unless** returns **nil**.

If *test* is **nil**, then the *form* arguments are evaluated in order. The value or values of the last *form* argument are returned as the result of **unless**. If no *form* arguments are specified, **unless** returns **nil**.

Syntax: **unless** *test* {*form*}* [*Macro*]

Examples:

```
> (unless nil 1)
1
> (unless t 2)
NIL
> (unless nil)
NIL
```

See Also: **when**

unwind-protect

Purpose: The **unwind-protect** special form is used to execute a protected form and to guarantee that a series of cleanup forms are executed before the **unwind-protect** exits.

The **unwind-protect** special form returns the value or values that result from the execution of the protected form.

Syntax: **unwind-protect** *protected-form* {*cleanup-form*}* [*Special Form*]

Remarks: The cleanup forms are generally used to ensure that if an exit of any kind causes the execution of the protected form to be aborted, the **unwind-protect** construct is able to perform any necessary actions before it exits.

The cleanup forms are not protected.

Examples:

```
> (defun foo (x)
    (setq state 'running)
    (unless (numberp x) (throw 'abort 'not-a-number))
    (setq state (1+ x)))
FOO
> (catch 'abort (foo 1))
2
> state
2
> (catch 'abort (foo 'trash))
NOT-A-NUMBER
> state
RUNNING
> (catch 'abort (unwind-protect (foo 'trash) (setq state 'aborted)))
NOT-A-NUMBER
> state
ABORTED
```

See Also: **throw**

catch

go

return

return-from

values

Purpose: The function **values** is used to return multiple values. It returns one value for each of its arguments, in order.

Syntax: **values** `&rest` *args* [*Function*]

Remarks: If any argument produces more than one value, only the first of these is returned.

If no arguments are specified, **values** returns no values.

Examples:

```
> (values)
> (values 1 2 3)
1
2
3
> (values (values 1 2 3) 4 5)
1
4
5
```

values-list

Purpose: The function **values-list** returns the elements of its *list* argument as multiple values.

Syntax: **values-list** *list* [*Function*]

Examples:

```
> (values-list nil)
> (values-list '(1 2 3))
1
2
3
```

when

Purpose: The **when** macro allows the execution of a series of forms to be dependent on a single test form.

If the *test* argument is **nil**, none of the *form* arguments are evaluated, and **when** returns **nil**.

If *test* is non-**nil**, then the *form* arguments are evaluated in order. The value or values of the last *form* argument are returned as the result of **when**. If no *form* arguments are specified, **when** returns **nil**.

Syntax: **when** *test* {*form*}* [*Macro*]

Examples:

```
> (when t 1)
1
> (when nil 2)
NIL
> (when t)
NIL
> (setq foo t)
T
> (when foo (setq foo nil) 3)
3
> (when foo 4)
NIL
```

See Also: **unless**

Chapter 6. Macros

Chapter 6. Macros

About Macros

Macros are important tools in constructing programs. Macros enable the user to write forms that do not obey the usual rules for evaluation. They provide facilities for data abstraction that are potentially more efficient to use than functions.

A macro is not a function, but rather a functionlike object that returns a Lisp expression to be evaluated in place of the macro call.

Macros are processed in a special way by the evaluator. When the evaluator encounters a macro call form, it calls the macro whose name is the first element in this form and passes to it the rest of the elements of the macro call form as arguments. These arguments are passed unevaluated to the macro.

Macro Evaluation

The evaluation of the macro is a process known as ***macro expansion***. Its result is an expression that is to be evaluated in place of the macro call form. The result of the macro expansion is substituted for the original macro call form. The evaluator evaluates the results of this macro substitution and returns the results as if they were the results of the macro call. If the result of the macro expansion is again a macro call form, the entire macro evaluation process is repeated. The functions **macroexpand** and **macroexpand-1** are used to perform the macro expansion operation.

When a program is compiled, the compiler manages the process of macro expansion. Macros may thus be used to provide an efficient data abstraction facility like that provided by functions, but without the run-time overhead involved in macro expansion. When a program using macros is compiled, the macro definition must precede the first macro use in the program text. Similarly, when a program is interpreted, all the macros in the body of the program must be known; otherwise they will be interpreted as unknown functions.

Macro Definition

Macro definition is performed by use of the **defmacro** facility. The syntax for defining macros is much like that for function definition.

Defining a macro causes an expansion function for the given macro to be associated with the macro name in the global environment. The body of the macro expansion function consists of the series of *form* arguments specified in the macro definition. When the macro expansion function is applied to the macro call form, the parameters specified in the lambda list given in the macro definition are bound to actual argument values, and the forms in the body of the macro expansion function are executed in the context of these bindings. The result returned by the macro expansion is the result of the last form evaluated. If no forms are evaluated, **nil** is returned.

The syntax for macro definitions is the following:

```
(defmacro name lambda-list {declaration | documentation}* {form}*)

lambda-list::= ([&whole var]
               {var}*
               [&environment var]
               [&optional {var | (var [initform [supplied-p-parameter] ])}*]
               [{&rest | &body} var]
               [&key {var | ({var | (keyword var)} [initform [supplied-p-parameter] ])}*
                     [&allow-other-keys] ]
               [&aux {var | (var [initform] )}*] )
```

Lambda Lists

The lambda list specifies the parameters of the macro expansion function. When the macro call is processed, the parameters specified in the lambda list are bound to the actual argument values occurring in the macro call, and the forms in the body of the lambda expression are executed in the context of these bindings. Unlike the arguments to functions, however, these arguments are passed unevaluated to the macro expansion function.

- The **&whole** keyword argument is optional. If it is specified, it must occur first in the lambda list. It causes the following variable to be bound to the macro call form.
- The specifiers for all required parameters must appear next in the list. If **&whole** is not specified, all parameters preceding the first lambda list keyword are required parameters. Otherwise all parameters following the **&whole** variable and preceding the next lambda list keyword are considered to be required parameters. The required parameters are bound to actual argument values in the order in which they occur. There must be at least as many actual argument forms as there are required parameters. If no further lambda list keywords are specified, there must be exactly as many actual arguments as parameters.
- The **&environment** lambda list keyword may be used to specify a lexical environment in which the macro call is to be evaluated. If it is used, it must follow the required lambda list parameters.
- Any optional parameters must be specified next. They are preceded by the lambda list keyword **&optional**. If optional parameters are specified, they are bound in order to the corresponding remaining values in the argument list. If there are no remaining arguments at any point in the processing of optional parameters, then any remaining optional parameter is bound to the value that results from the evaluation of its associated *initform*, if the latter is given, or to **nil**, if not. A *supplied-p-parameter* variable may be used in conjunction with an *initform*. Its purpose is to indicate

whether an actual argument value was supplied. It is bound to *true* if an actual argument was supplied; otherwise (if the *initform* was evaluated), it is bound to **nil**.

- One rest parameter may be specified next. It is preceded by the **&rest** lambda list keyword. If a rest parameter has been specified, it is bound to a list consisting of all the actual arguments that have not yet been processed. If no arguments remain, the rest parameter is bound to **nil**.

- The **&body** keyword may be used instead of **&rest**. It performs the same function, but it also provides information to formatting functions.

- The use of the lambda list keyword **&key** and keyword parameter specifiers enables keyword arguments to be used in macro calls. If any keyword parameters are to appear in the macro call, they must be preceded by **&key** in the lambda list. These keyword parameters may be followed by the lambda list keyword **&allow-other-keys**.

  A keyword parameter may be specified in one of three ways. These forms differ in whether the name for the keyword to be used in the actual argument list is specified explicitly or implicitly and whether an initial value is to be used if such a keyword argument is not specified.

  If a variable, *var*, specifies the keyword parameter, the keyword argument to be used in the argument list consists of a keyword (in the keyword package) with the same name as *var*. Thus, for example, **&key** *name* in the lambda list corresponds to :*name* in the macro call form. If such a keyword does not appear in the argument list, *var* is bound to **nil**.

  If the form (*var* [*initform* [*supplied-p-parameter*]]) specifies the keyword parameter, the keyword argument to be used is specified in the same way as in the simpler case discussed above. This construct, however, allows the variable to be bound to an initial value if the keyword is not specified in the argument list. The *supplied-p-parameter* may be used to test whether such an argument value was specified.

  The form ((*keyword var*) [*initform* [*supplied-p-parameter*]]) allows the explicit specification of the argument list keyword that is associated with *var*. It also allows the variable to be bound to an initial value if the keyword is not specified in the argument list.

  There must be an even number of actual keyword arguments. Keyword arguments are considered to occur in pairs. The first argument in the pair is a keyword; the second is the value to which the corresponding keyword parameter is to be bound. 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. If a given keyword argument is specified more than once, however, the first keyword-value pair is used in the binding of the keyword parameter. If a rest parameter has been specified, the arguments used in processing keyword parameters are the same as those used in processing the rest parameter.

- The **&allow-other-keys** lambda list keyword is used to specify that the argument list may contain a keyword that does not correspond to a lambda list keyword parameter. Otherwise it is an error if such an argument pair occurs unless the argument list contains a keyword-value pair whose key is **:allow-other-keys** and whose value is non-**nil**. The **&rest** keyword parameter may be used to access values specified by means of the **&allow-other-keys** and **:allow-other-keys** constructs.

  It is an error if there are remaining arguments and neither a rest parameter nor a keyword parameter has been specified.

- Finally, the **&aux** lambda list keyword may be used to specify auxiliary variables. These serve as local variables within the macro expansion function. Auxiliary variables are not bound to argument list values. An auxiliary variable may be bound within the lambda expression itself or by specifying a corresponding *initform* in the lambda list.

Since the lambda list elements are processed in the order in which they occur, any *initform* may reference a parameter variable (including a *supplied-p-parameter* variable) that is bound earlier in the processing of the lambda list.

When the function exits, the variable bindings in effect before the function invocation are restored.

Destructuring Facility

The macro ***destructuring*** facility provides for a generalization of the lambda list syntax. The destructuring facility allows a lambda list to appear wherever a parameter name (but not a list) can appear in a lambda list. When the actual arguments are processed, the embedded lambda list itself is bound to the form to which such a parameter would have been bound. This binding is also performed according to the method described above.

The destructuring facility allows for a dotted lambda list that ends with a parameter name. In this case, the last parameter is treated as if it had been preceded by the **&rest** lambda list keyword instead.

Backquote Facility

The ***backquote*** (\`) mechanism is designed to simplify the writing of macro definitions. It can be used in the macro body to create a template for the macro expansion. A list preceded by a backquote provides a list template into which elements are spliced. The backquote acts just like the quote (') construct, except that it allows the following constructs to be used.

The *comma* (`,`) construct is used in conjunction with the backquote mechanism. If a comma immediately precedes a form in such a template, that form is evaluated and the result is spliced into the resulting list at the position where the comma and its associated form occurred. The comma thus has the effect of "unquoting" the following form.

A comma may also be followed by the at-sign symbol (`@`). The `,@` construct specifies that the evaluation of the following form produces a list of objects. These objects themselves (not the list) are inserted into the resulting list at the position where the `,@` and its associated form occurred.

The `,.` construct is like the `,@` construct, except that it may have the side effect of modifying the list produced by evaluating the associated form.

Any other forms occurring in such a template are not evaluated. They remain at the same position in the resulting list as they occupy in the template.

The backquote facility is discussed further in the chapter "Input/Output."

Categories of Operations

This section groups operations on macros according to functionality.

Macro Definition

defmacro	**macro-function**
define-macro	

These functions are used to define macros.

Macro Expansion

macroexpand	***macroexpand-hook***
macroexpand-1	

These constructs are used to expand macros.

define-macro

Purpose: The function **define-macro** is used by **defmacro** to do the actual defining of a new macro. It replaces the function cell of the named symbol with the specified function object.

If the function is currently traced, it remains traced, but with the new definition.

Syntax: **define-macro** *name function* [*Function*]

Remarks: The *name* argument is a symbol.

The function **define-macro** is an extension to Common Lisp.

Examples:

```
> (define-macro 'foo #'do)
FOO
> (foo ((i 0 (1+ i))) ((> i 2) i))
3
```

See Also: **defmacro**

symbol-function

redefinition-action

defmacro

Purpose: The **defmacro** macro is used for macro definition. It causes an expansion function for the given macro name to be defined in the global environment.

The *name* argument of **defmacro** is a symbol; it is not evaluated. The **defmacro** macro causes a global macro expansion function to be associated with the function cell of the symbol *name*.

The body of the macro expansion function is specified by the *form* arguments. They are executed in order. The value of the last form executed is returned as the result of executing the macro.

The *name* of the new macro is returned as the result of **defmacro**.

Syntax: **defmacro** *name lambda-list* {*declaration* | *documentation*}* {*form*}* [*Macro*]

Remarks: The definition of macros and the syntax of lambda lists are discussed in the section "About Macros."

A documentation string may be attached to the name of the function by use of the optional *documentation* argument; the documentation type for this string is **function**.

The **defmacro** macro can be used to redefine a macro or to replace a function definition with a macro definition. The Common Lisp special forms may not be redefined.

When **defmacro** occurs at the top level in a file, it is implicitly wrapped in the construct `(eval-when (eval compile load) ...)`.

Examples:

```
> (defmacro foo1 (a b) '(+ ,a (* ,b 3)))
FOO1
> (foo1 4 5)
19
> (defmacro foo2 (&optional (a 2 b) (c 3 d) &rest x) ''(,a ,b ,c ,d ,x))
FOO2
> (foo2 6)
(6 T 3 NIL NIL)
> (foo2 6 3 8)
(6 T 3 T (8))
> (defmacro foo3 (&whole r a &optional (b 3) &rest x &key c (d a))
    ''(,r ,a ,b ,c ,d ,x))
FOO3
> (foo3 1 6 :d 8 :c 9 :d 10)
((FOO3 1 6 :D 8 :C 9 :D 10) 1 6 9 8 (:D 8 :C 9 :D 10))
```

```
> (defmacro foo4
   (&whole (su &rest (p &rest q)) a &optional (b 3) &rest x &key c (d a))
   ''(,su ,p ,a ,b ,c ,d ,x))
FOO4
> (foo4 1 6 :d 8 :c 9 :d 10)
(FOO4 1 1 6 9 8 (:D 8 :C 9 :D 10))
```

See Also: **macrolet**

macro-function

Purpose: The function **macro-function** is used to determine whether a given symbol has a global function definition that is a macro definition. If it does, the macro expansion function is returned. If the symbol has no global function definition or is not a macro, **macro-function** returns **nil**.

Syntax: **macro-function** *symbol* [*Function*]

Remarks: The function **macro-function** examines global definitions only.

The macro **setf** can be used with **macro-function** to replace the global macro definition associated with a symbol. The function definition argument to **setf** must be a function of two arguments, a macro call form and an environment. It should compute the macro expansion for the call.

Examples:

```
> (defmacro foo (x) '(macro-function 'foo))
FOO
> (not (macro-function 'foo))
NIL
> (and (setf (macro-function 'foo) #'equal)
       (equal (macro-function 'foo) #'equal))
T
> (macrolet ((foo (x) "local"))
    (equal (macro-function 'foo) #'equal))
T
```

See Also: **defmacro**

macroexpand, macroexpand-1

Purpose: The functions **macroexpand** and **macroexpand-1** are used to expand macros.

If the *form* argument is a macro call, the function **macroexpand-1** expands the macro call once. It returns the macro expansion and **t** as its results. If the *form* argument is not a macro call, **macroexpand-1** returns *form* and **nil** as its results.

The function **macroexpand** is like **macroexpand-1** except that it causes the *form* argument to be expanded until it is no longer a macro call. It returns the macro expansion and **t** as its results. If the *form* argument is not a macro call, **macroexpand** returns *form* and **nil** as its results.

Syntax:

macroexpand *form* `&optional` *env* [*Function*]

macroexpand-1 *form* `&optional` *env* [*Function*]

Remarks: The *env* argument specifies a lexical environment. It may be used to specify an environment in which local macro definitions exist. If it is not specified, the null lexical environment is used.

Examples:

```
> (defmacro outer (x y) `(inner ,x ,y))
OUTER
> (defmacro inner (x y) `(aa ,x ,y))
INNER
> (defun not-a-macro (x y) x)
NOT-A-MACRO
> (defmacro env-sens (x y &environment e)
                    `(macroexpand '(inner ,x ,y) ',e))
ENV-SENS
> (macroexpand-1 '(outer a b))
(INNER A B)
T
> (macroexpand '(outer a b))
(AA A B)
T
> (macroexpand '(not-a-macro a b))
(NOT-A-MACRO A B)
NIL
> (macroexpand-1 'not-a-macro)
NOT-A-MACRO
NIL
> (env-sens a b)
(AA A B)
T
```

```
> (macrolet ((inner (x y) '(+ ,x ,y))) (env-sens a b))
(+ A B)
T
```

See Also: ***macroexpand-hook***

macroexpand-hook

Purpose: The variable ***macroexpand-hook*** is used to control the macro expansion process.

When a macro is expanded, the function to which ***macroexpand-hook*** is bound is called with three arguments: the macro expansion function, the macro call form, and the environment in which the expansion is to take place.

Syntax: ***macroexpand-hook*** [*Variable*]

Remarks: The initial value of ***macroexpand-hook*** is **funcall**.

Examples:

```
> (defun hook (expander form env)
    (format t "Now expanding: ~S~%" form)
    (funcall expander form env))
HOOK
> (defmacro foo (x y) `(/ (+ ,x ,y) 2))
FOO
> (macroexpand '(foo 1 2))
(/ (+ 1 2) 2)
T
> (let ((*macroexpand-hook* #'hook)) (macroexpand '(foo 1 2)))
Now expanding: (FOO 1 2)
(/ (+ 1 2) 2)
T
```

See Also:

macroexpand

macroexpand-1

funcall

Chapter 7. The Evaluator

Chapter 7. The Evaluator

About the Evaluator

The ***evaluator*** executes programs by evaluating forms.

The evaluator is invoked automatically in the top-level read-eval-print loop. This is the normal interpretive mode of interaction with the system in which the user types in a form, the form is read by the Lisp reader, it is evaluated by the evaluator, and the resulting value or values are printed out for the user's inspection. The read-eval-print loop then automatically re-enters a state in which it is again waiting for the user to enter a form. The top-level loop also maintains a number of global variables that enable the user to examine recent forms that have been entered and the results of their evaluation.

The evaluator may also be invoked explicitly by means of the function **eval**. The expression `(eval` *form*`)` applies the function **eval** to the *form* argument. Because **eval** is itself a normal function (and not a special form), the *form* argument is evaluated before it is passed to **eval**. When **eval** itself is explicitly invoked, the result of this argument evaluation is itself evaluated.

Before any form is executed, all the macros in it are expanded. The first time that an interpreted function is called, it is replaced by a function object; all the macros in the function body of this function object have been expanded.

The normal action of the evaluator may be modified by means of the variables ***evalhook*** and ***applyhook*** and the functions **evalhook** and **applyhook**. These allow the user to specify evaluation functions that may be useful for special purposes, such as debugging.

Categories of Operations

These functions and variables are used in evaluation.

*	**applyhook**
**	***applyhook***
***	**decache-eval**
+	**eval**
++	**evalhook**
+++	***evalhook***
-	**grindef**
/	***prompt***
//	**source-code**
///	

*, **, ***

Purpose: The global variables *, **, and *** are maintained by the top-level read-eval-print loop to save the values of results that were printed at the end of the loop.

The variable * is bound to the last result printed, the variable ** is bound to the previous value of *, and the variable *** is bound to the previous value of **.

Syntax:

* [*Variable*]

** [*Variable*]

*** [*Variable*]

Remarks: If more than one value is produced, * is bound to the first value only. If no value is produced, * is bound to **nil**.

The values of these variables are not updated when the evaluation of a form is aborted.

Examples:

```
> 3
3
> "two"
"two"
> (values 'star "second value not retained by *")
STAR
"second value not retained by *"
> (format t "* => ~S~%** => ~S~%*** => ~S~%" * ** ***)
* => STAR
** => "two"
*** => 3
NIL
```

See Also:

/

//

///

+, ++, +++

Purpose: The global variables +, ++, and +++ are maintained by the top-level read-eval-print loop to save forms that were recently evaluated.

The variable + is bound to the last form that was evaluated, the variable ++ is bound to the previous value of +, and the variable +++ is bound to the previous value of ++.

Syntax:

+ [*Variable*]

++ [*Variable*]

+++ [*Variable*]

Examples:

```
> (third '(1 2 3))
3
> (second '(1 2 3))
2
> (first '(1 2 3))
1
> (format t "+ =>~S~%++ => ~S~%+++ => ~S~%" + ++ +++)
+ =>(FIRST (QUOTE (1 2 3)))
++ => (SECOND (QUOTE (1 2 3)))
+++ => (THIRD (QUOTE (1 2 3)))
NIL
```

See Also: –

—

Purpose: The variable – is bound to the form that is currently being evaluated by the read-eval-print loop.

Syntax: – *[Variable]*

Examples:

```
> (format t "- => ~S~%" -)
- => (FORMAT T "- => ~S~%" -)
NIL
```

See Also: +

++

+++

/, //, ///

Purpose: The global variables /, //, and /// are maintained by the top-level read-eval-print loop to save the values of results that were printed at the end of the loop.

The variable / is bound to a list of the values that were last printed, the variable // is bound to the previous value of /, and the variable /// is bound to the previous value of //.

Syntax:

/ [*Variable*]

// [*Variable*]

/// [*Variable*]

Remarks: The values of these variables are not updated when the evaluation of a form is aborted.

Examples:

```
> (values 1 2 3)
1
2
3
> (floor 300/14 23)
0
150/7
> "singleton"
"singleton"
> (format t "/ => ~S~%// => ~S~%/// => ~S~%" / // ///)
/ => ("singleton")
// => (0 150/7)
/// => (1 2 3)
NIL
```

See Also:

*

**

***

decache-eval

Purpose: The function **decache-eval** forces the re-expansion of all function bodies when they are next executed.

Syntax: **decache-eval** *[Function]*

Remarks: The function **decache-eval** should normally be used only in exceptional situations or in the debugging of macros.

The first time that an interpreted function is called, it is replaced by a function object; all the macros in the function body of this function object have been expanded. After a macro has been redefined or a function has been redefined as a macro, the re-expansion of the function body will occur automatically when the body of the function is next entered. The function **decache-eval** need not be invoked in this situation.

Redefinition of a function while that function is running may cause unpredictable results.

The function **decache-eval** is an extension to Common Lisp.

Examples:

```
> (defvar *expand-counter* 0)
*EXPAND-COUNTER*
> (defmacro return-counter () (incf *expand-counter*))
RETURN-COUNTER
> (defun use-return-counter () (return-counter))
USE-RETURN-COUNTER
> *expand-counter*
1
> (use-return-counter)
2
> *expand-counter*
2
> (use-return-counter)
2
> (decache-eval)
T
> *expand-counter*
2
> (use-return-counter)
3
> (use-return-counter)
3
```

See Also: **eval**

eval

Purpose: The function **eval** evaluates its *form* argument and returns the result.

The evaluation takes place in the current dynamic environment and a null lexical environment.

Syntax: **eval** *form* [*Function*]

Remarks: The function **eval** handles its arguments in the normal way. That is, the argument is evaluated before it is passed to the function **eval**. Two levels of evaluation thus take place.

Examples:

```
> (setq form '(1+ a) a 999)
999
> (eval form)
1000
> (eval 'form)
(1+ A)
> (let ((a '(this would break if eval used local value))) (eval form))
1000
```

evalhook, applyhook

Purpose: The functions **evalhook** and **applyhook** rebind the variables ***evalhook*** and ***applyhook*** for the course of the execution of one form or one function respectively.

The function **evalhook** temporarily rebinds the ***evalhook*** variable to the *evalhookfn* function and the ***applyhook*** variable to the *applyhookfn* function and then evaluates the specified *form.*

The function **applyhook** operates similarly. It temporarily rebinds the ***evalhook*** variable to the *evalhookfn* function and the ***applyhook*** variable to the *applyhookfn* function and then applies the *function* argument to the argument list specified by *args.*

Both the **evalhook** and **applyhook** functions rebind the ***evalhook*** and ***applyhook*** variables for the evaluation of the top-level *form* or *function* only and not for the evaluation of subforms.

The optional *env* argument may be used to specify the lexical environment in which the evaluation is to occur. If it is **nil** or not specified, the null lexical environment is used.

Syntax:

evalhook *form evalhookfn applyhookfn* `&optional` *env* [*Function*]

applyhook *function args evalhookfn applyhookfn* `&optional` *env* [*Function*]

Examples:

```
> (defvar *foo*)
*FOO*
> (defun hook1 (x)
   (let ((*evalhook* #'eval-hook-function))
    (eval x)))
HOOK1
> (defun eval-hook-function (form &optional env)
   (setq *foo* form)
    (evalhook form #'eval-hook-function nil env))
EVAL-HOOK-FUNCTION
> (defun hook2 (x)
   (let ((*applyhook* #'apply-hook-function))
    (eval x)))
HOOK2
> (defun apply-hook-function (fun args &optional env)
   (setq *foo* (car args)))
APPLY-HOOK-FUNCTION
```

```
> (let ((*foo* nil)) (hook1 t) *foo*)
(QUOTE T)
> (let ((*foo* nil)) (hook2 '(car (cons t 2))) *foo*)
(CAR (CONS T 2))
```

See Also: **eval**

evalhook

applyhook

evalhook, *applyhook*

Purpose: The global variables ***evalhook*** and ***applyhook*** are used to modify the behavior of **eval**.

If the values of ***evalhook*** and ***applyhook*** are **nil**, **eval** has its usual behavior. By rebinding ***evalhook*** or ***applyhook***, users can replace the evaluator with their own functions for evaluating forms and functions.

The ***evalhook*** variable may be rebound to a function of two arguments: a form and an environment. When any form is to be evaluated, this hook function is invoked instead of **eval** to evaluate the form. The form to be evaluated is passed to the hook function without any prior evaluation. The results of executing the hook function are returned as if they were the results of having executed **eval**.

The ***applyhook*** variable may be rebound to a function of three arguments: a function, a list of arguments, and an environment. The applyhook function is invoked whenever a function is to be applied to arguments. The results of executing the hook function are returned as if they were the results of having applied the function to its arguments in the usual way.

Whenever either of the hook functions is itself invoked, the values of both ***evalhook*** and ***applyhook*** are **nil**. The functions specified by the hooks are thus invoked in the normal way.

Syntax:

evalhook [*Variable*]

applyhook [*Variable*]

Remarks: An environment argument may be used to specify the lexical environment in which the evaluation is to occur. If it is **nil** or not specified, the null lexical environment is used.

The hook function is only relevant to interpreted calls to functions.

If there is a throw back to the top level, both ***evalhook*** and ***applyhook*** are automatically reset to **nil**.

Examples:

```
> (defvar *last-form*)
*LAST-FORM*
> (defun ehook (form &optional env)
    (setq *last-form* form)
    (eval form))
EHOOK
> (let ((*evalhook* #'ehook)) (+ 1 2 3))
6
```

```
> *last-form*
(+ 1 2 3)
> (defun ahook (f args &optional env) (cdr args))
AHOOK
> (let ((*applyhook* #'ahook)) (+ 1 2 3))
(2 3)
```

See Also: **eval**

evalhook

applyhook

grindef

Purpose: The **grindef** macro pretty-prints the source code associated with the name of an interpreted function. The macro **grindef** returns no values.

Syntax: **grindef** `&rest` *function-name* [*Macro*]

Remarks: The *function-name* argument is not evaluated.

The macro **grindef** is an extension to Common Lisp.

Examples:

```
> (defun grist (x y)
    (let ((a 1)(b 2)(c 3))(+ x a b c)))
GRIST
> (grindef grist)

(DEFUN GRIST
       (X Y)
         (LET ((A 1)
               (B 2)
               (C 3))
           (+ X A B C)))
```

prompt

Purpose: The global variable ***prompt*** is used to specify the string to be used as a prompt in the top-level read-eval-print loop. Initially, ***prompt*** is unbound and the default prompt string (>) is used.

Syntax: ***prompt*** [*Variable*]

Remarks: The variable ***prompt*** is an extension to Common Lisp.

Examples:

```
> (setq *prompt* "at your service! " dummy nil)
NIL
at your service! 999
999
at your service! (makunbound '*prompt*)
*PROMPT*
> 999
999
```

source-code

Purpose: The function **source-code** returns the source code of an interpreted function.

The *function* argument may be a function object or a symbol. If the argument is an interpreted function or a symbol that has a function definition that is an interpreted function, the source code of the function is returned. Otherwise **source-code** returns **nil**.

Syntax: **source-code** *function* [*Function*]

Remarks: The function **source-code** is an extension to Common Lisp.

Examples:

```
> (source-code #'car)
NIL
> (source-code #'(lambda (x) (1+ x)))
(LAMBDA (X) (1+ X))
> (defun ink (x) (1+ x))
INK
> (source-code #'ink)
(NAMED-LAMBDA INK (X) (BLOCK INK (1+ X)))
```

Chapter 8. Declarations

Chapter 8. Declarations

About Declarations

Declarations are used to affect the status of variable bindings, to provide advice to the compiler, and to add documentation to a program.

With the exception of **special** declarations, declarations are optional and are used as advice to the compiler. The meaning of a correct program is not affected by declarations other than **special** declarations.

The use of declarations, however, may have a significant impact on the efficiency of compiled code. The user is referred to the *Sun Common Lisp User's Guide* for more information about compiled code.

The **proclaim** function is used to make global declarations. Such global declarations are also called ***proclamations***. The **declare** special form is used for local declarations within other Common Lisp forms. Unless explicitly noted, the term "declaration" is used to refer to both declarations and proclamations.

Syntax for Declaration Specifiers

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`

Types of Declarations

A **special** declaration specifies that the given variables are all special variables. References to the variables 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.

A **type** declaration asserts that the given variables will only have values of the specified type. A short form of this declaration, (*type-specifier* {*var*}*), may be used if the type specifier is one of the atomic types listed in Figure 3–1. The **type** declaration applies only to those variables whose bindings are established by the form in which the declaration occurs. Type proclamations take effect only for special bindings of such variables. The **type** declaration is used to enable the compiler to produce more efficient code.

An **ftype** declaration is used to specify that a series of functions are of a given function type. This means that whenever the arguments of these functions are of the indicated types, the results of the functions will also be of the types specified in the **ftype** declaration. A **function** declaration is equivalent to an **ftype** declaration of the form `(ftype (function` *arglist* `(values` *result-type1 result-type2* ... `)` *name*`))`. This abbreviated form is provided for convenience. An **ftype** or **function** declaration obeys the rules of lexical scoping. The **ftype** and **function** declarations are used to enable the compiler to produce more efficient code.

An **inline** declaration specifies that it is desirable that the code for a given function be compiled in-line, rather than as a function call. An **inline** declaration obeys the rules of lexical scoping. A **notinline** declaration specifies that the code for the given function is not to be compiled in-line, but rather as a function call. The **inline** and **notinline** declaration types apply to all occurrences of the specified function in the body of the form in which the declaration occurs. Both **inline** and **notinline** declarations are ignored by the interpreter.

An **ignore** declaration applies only to those variables whose bindings are established by the form in which the declaration occurs. An **ignore** declaration prevents the compiler from issuing a warning if any of the given variables are not referenced in the body of code.

An **optimize** declaration provides advice to the compiler about what trade-offs should be made in optimizing code. There are four optimization classes: **speed**, **safety**, **space**, and **compilation-speed**. Each class may be assigned an integer value between 0 and 3. This value indicates the priority assigned to that type of optimization; the highest priority is 3, the lowest is 0. In Sun Common Lisp, the default values are **speed** 3, **safety** 1, **space** 0, and **compilation-speed** 0. An **optimize** declaration applies to all of the code in the body of the form in which it occurs.

A **declaration** proclamation specifies that the given declaration names are not names of standard declarations, although they may be used as such. A **declaration** proclamation advises the compiler that warnings are not to be issued if the given names are used as declaration specifiers. The **declaration** declaration specifier may be used in proclamations only.

Categories of Operations

These constructs are used to specify declarations.

declare	**locally**
proclaim	**the**

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

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

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

Macro calls may expand into declarations as long as this syntax is observed.

The declaration specifier argument is not evaluated.

Examples:

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

See Also: **proclaim**

locally

Purpose: The **locally** macro is used to make local declarations that affect only the *form* arguments in its body.

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

Examples:

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

proclaim

Purpose: The **proclaim** function is used to make a global declaration or proclamation.

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

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

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

Type proclamations take effect only for special bindings of variables.

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))        ;the binding of prosp is special
    (set 'prosp 3) (set 'reg 3)   ;due to the preceding proclamation,
    (list prosp reg))             ;whereas the variable reg is lexical
(3 2)
> (list prosp reg)
(1 3)
```

See Also: **declare**

defvar

defparameter

the

Purpose: The **the** special form is used to specify that the value produced by a form will be of a certain type.

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

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

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

The macro **setf** may be used with **the** type declarations. In this case the declaration is transferred to the form that specifies the new value. The resulting **setf** form is then analyzed.

The form **the** is mainly used by the compiler to produce more efficient code.

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 9. Predicates

Chapter 9. Predicates

About Predicates

Predicates are functions that test for some condition involving their arguments. They return **nil** if the condition is false and some non-**nil** value if the condition is true. The symbol **t** is used as the standard true value if no more specific non-**nil** value is available. The values of **t** and **nil** cannot be modified.

A predicate is said to be true of an object if it returns a non-**nil** value and false if it returns **nil**.

Constructs that test for logical values consider any non-**nil** value to be true; only **nil** is considered to be false.

This chapter discusses logical predicates and constants and the predicates that test for the equality of two objects. Data type predicates are discussed in the chapters that follow.

Categories of Operations

These constructs are used in logical operations.

Equality Predicates

eq	**equal**
eql	**equalp**

The predicates **eq**, **eql**, **equal**, **equalp** provide a range of equality tests, from the most specific (**eq**) to the most general (**equalp**). Any objects that satisfy one of the equality tests also satisfy any such equality test that is more general.

Logical Constants

t	**nil**

These constants specify logical values.

Logical Operators

and	**not**
or	

These macros perform logical operations.

and

Purpose: The macro **and** evaluates each of its arguments in turn. As soon as any *form* evaluates to **nil**, **and** returns **nil** without evaluating the remaining forms. If all forms but the last evaluate to non-**nil** values, **and** returns the results produced by evaluating the last of the forms.

Syntax: **and** {*form*}* [*Macro*]

Remarks: The result of evaluating the expression (and) is **t**.

Examples:

```
> (setq foo1 1 foo2 1 foo3 1)
1
> (and (incf foo1) (incf foo2) (incf foo3))
2
> (and (eql 2 foo1) (eql 2 foo2) (eql 2 foo3))
T
> (decf foo3)
1
> (and (decf foo1) (decf foo2) (eq 'foo 'nil) (decf foo3))
NIL
> (and (eql foo1 foo2) (eql foo2 foo3))
T
> (and)
T
```

See Also: **or**

eq

Purpose: The predicate **eq** is true of two objects if they are both the same object, that is, if they both occupy the same memory locations; otherwise it is false.

Syntax: **eq** *x y* [*Function*]

Remarks: Objects that are printed the same may or may not be **eq**. Symbols having the same print name are usually **eq** because of the use of the function **intern**. Numbers having the same value, however, are often not **eq**.

Examples:

```
> (eq 3 3.0)
NIL
> (eq (cons 'a 'b) (cons 'a 'b))
NIL
> (eq "Foo" (copy-seq "Foo"))
NIL
```

See Also: **eql**

equal

equalp

=

eql

Purpose: The predicate **eql** is true of two objects if they are **eq**, if they are both numbers of the same type and the same value, or if they are both character objects that represent the same character; otherwise it is false.

Syntax: **eql** *x y* [*Function*]

Remarks: The predicate **eql** may not be true of floating-point numbers even when they represent the same value. The predicate = should be used instead when comparing mathematical values.

Two complex numbers are **eql** if both their real and imaginary parts are **eql**.

Examples:

```
> (eql 3 3.0)
NIL
> (eql (cons 'a 'b) (cons 'a 'b))
NIL
```

See Also: **eq**

equal

equalp

=

equal

Purpose: The predicate **equal** is true of two objects if they are symbols that are **eq**, if they are numbers that are **eql**, or if they are character objects that are **eql**.

Objects that have components are **equal** if they are both of the same type and if the corresponding components of each are **equal**.

Arrays other than strings and bit-vectors are **equal** only if they are **eq**. Strings and bit-vectors are compared up to the limits specified by fill pointers (if any). They are **equal** if and only if the fill pointers are = and if the corresponding elements of each are **equal**. Character case differences in strings are observed by **equal**.

Pathnames that are printed the same are **equal**.

Syntax: **equal** *x y* [*Function*]

Remarks: The predicate **equal** may fail to terminate if its arguments are circular structures.

Examples:

```
> (equal 3.0 3.0)
T
> (equal (cons 'a 'b) (cons 'a 'b))
T
> (equal "Foo" "Foo")
3
> (equal "Foo" "foo")
NIL
```

See Also:

eq

eql

equalp

=

string=

string-equal

char=

char-equal

tree-equal

equalp

Purpose: The predicate **equalp** is true of two objects if they are **equal**, if they are characters that are **char-equal**, or if they are numbers that are =. Objects that have components are **equalp** if they are both of the same type and if the corresponding elements of each are **equalp**.

Two arrays are **equalp** if and only if they have the same number of dimensions, the dimensions are of the same length, and the corresponding elements of each are **equalp**. If, however, either array has a fill pointer, then the arrays are compared only up to the limits specified by the fill pointers. They are **equalp** if and only if the fill pointers are = and the corresponding elements of each are **equalp**.

Syntax: **equalp** *x y* [*Function*]

Remarks: The predicate **equalp** ignores case differences when comparing character or string objects.

The predicate **equalp** may fail to terminate if its arguments are circular structures.

Examples:

```
> (equalp 3.0 3.0)
T
> (equalp "Foo" "foo")
T
```

See Also:

eq

eql

equal

=

string=

string-equal

char=

char-equal

nil

Purpose: The value of the constant **nil** is **nil**. The constant **nil** represents both the logical *false* value and the empty list. It is also written as `()`.

Syntax: **nil** [*Constant*]

Remarks: It is not possible to modify the value of **nil**.

Examples:

```
> nil
NIL
```

See Also: **t**

not

Purpose: The function **not** returns **t** if its argument is **nil**; otherwise it returns **nil**.

Syntax: **not** *x* [*Function*]

Remarks: The result of applying **not** to an object is the same as that of using **null**. The function **not** is intended to be used in inverting a logical value, whereas **null** is intended to be used in testing for an empty list.

Examples:

```
> (not nil)
T
> (not (integerp 'sss))
T
> (not (integerp 1))
NIL
```

See Also: **null**

or

Purpose: The macro **or** evaluates each of its arguments in turn. As soon as any *form* evaluates to a non-**nil** value, **or** returns that value without evaluating the remaining forms. If all forms but the last evaluate to **nil**, **or** returns the results produced by evaluating that form.

Syntax: **or** {*form*}* [*Macro*]

Remarks: The result of evaluating the expression `(or)` is **nil**.

Examples:

```
> (or)
NIL
> (setq foo1 1 foo2 1 foo3 1)
1
> (or nil nil foo1 (setq foo2 nil))
1
> (eq foo2 nil)
NIL
> (or (incf foo1) (incf foo2) (incf foo3))
2
> foo1
2
> foo2
1
> foo3
1
```

See Also: **and**

t

Purpose: The value of the constant **t** is **t**.

Syntax: **t** *[Constant]*

Remarks: It is not possible to modify the value of **t**.

Examples:

```
> t
T
```

See Also: **nil**

Chapter 10. Symbols

Chapter 10. Symbols

About Symbols

Symbols are data objects with five components: a ***print name***, a ***value cell***, a ***function cell***, a ***property list***, and a ***package cell***.

The ***print name*** is a string that is used to identify and locate the symbol. Symbol names are unique within a package.

The ***value cell*** is the cell that holds the current value of the dynamic variable associated with the symbol. When a new symbol is created, the contents of this cell are normally undefined, and the variable is said to be ***unbound***. An error occurs if such an unbound variable is accessed. A value may be associated with this cell by assignment functions or by constructs that establish new variable bindings. Constructs for accessing the value cell and for binding and unbinding a symbol are discussed in the chapters "Program Structure" and "Control Structure."

The ***function cell*** contains the global function definition associated with the symbol. When a new symbol is created, the contents of this cell are also normally undefined. Accessing it in this state causes an error. A value may be associated with the function cell through the various function definition constructs. Constructs for accessing and modifying the function cell of a symbol are discussed in the chapters "Program Structure" and "Control Structure."

A ***property list*** allows an extensible set of named components to be associated with a symbol. A component may be any Lisp object. Each successive two elements of the property list constitute an entry. The first element of an entry is the ***indicator***, or property name, and the second element is the property value. Indicators within the same property list are unique. When a symbol is created, its property list is empty.

The ***package cell*** refers to a package object. A ***package*** is a Common Lisp object that specifies a correspondence between print name strings and symbols. It is used to locate a symbol. A symbol is ***owned*** by only one package. The package that owns the symbol is called the symbol's ***home package***. The package cell of the symbol specifies the symbol's home package. If a symbol is owned by a package, it is said to be an ***interned*** symbol. Symbols not owned by any package are ***uninterned*** symbols. The package cell of an uninterned symbol is **nil**. An uninterned symbol is normally printed as #: followed by its print name. Packages are discussed in the chapter "Packages."

When a symbol identifier is read by the Lisp reader, an interned symbol is normally created automatically. If a symbol with this name is not already accessible in the current package, a new one is created whose print name corresponds to the identifier. If such a symbol identifier contains lowercase characters, the Lisp reader converts them to uppercase unless they are preceded by the escape character \ or enclosed by the | multiple escape characters.

Categories of Operations

This section groups operations on symbols according to functionality.

Data Type Predicates

symbolp	**keywordp**

These predicates determine whether an object is a symbol.

Functions on Property Lists

get	**remf**
getf	**remprop**
get-properties	**symbol-plist**

These functions may be used to access and alter a symbol's property list.

Functions on Package Cells and Print Names

symbol-name	**symbol-package**

These functions may be used to access package cells and print names.

Creating Symbols

copy-symbol	**gensym**
make-symbol	**gentemp**

These functions provide for the creation of uninterned and interned symbols.

copy-symbol

Purpose: The function **copy-symbol** creates and returns a new uninterned symbol whose print name is the same as that of a given symbol.

If the *copy-props* argument is non-**nil**, the contents of the new symbol's value and function cells and its property list are copied from *symbol.* If the *copy-props* argument is **nil**, the contents of the symbol's value and function cells are undefined, and the symbol's property list is empty.

Syntax: **copy-symbol** *symbol* `&optional` *copy-props* [*Function*]

Examples:

```
> (setq *symbol* t)
T
> (symbol-package (copy-symbol *symbol*))
NIL
> (boundp (copy-symbol *symbol*))
NIL
> (boundp (copy-symbol *symbol* t))
T
```

See Also: **make-symbol**

gensym

Purpose: The function **gensym** creates and returns a new uninterned symbol. The print name of the symbol consists of a prefix followed by a positive decimal integer.

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

Remarks: The optional argument may be a string or a positive integer. If the optional argument is not specified, the prefix for the symbol's print name is **G**, and the number is a count value maintained by **gensym**. If a string is specified, **gensym** uses that string as a prefix for the current and future calls. If a positive integer value is specified, the internal counter of **gensym** is reset to that value.

Examples:

```
> (symbol-package (gensym))
NIL
> (symbol-name (gensym 99))
"G99"
> (symbol-name (gensym "FoO"))
"FoO100"
> (symbol-name (gensym))
"FoO101"
> (symbol-name (gensym 2))
"FoO2"
> (symbol-name (gensym "G1"))
"G13"
```

See Also: **gentemp**

gentemp

Purpose: The function **gentemp** creates and returns a new symbol.

The symbol created is interned in the package *package*. If the *package* argument is not specified, the symbol is interned in the current package. It is guaranteed that the symbol created will be unique within the package.

Syntax: **gentemp** `&optional` *prefix package* [*Function*]

Remarks: The name of the symbol is prefixed by the string specified by the *prefix* argument. If the *prefix* argument is not specified, the name is prefixed by **T** followed by a nonnegative integer value. This number is a counter value maintained by **gentemp**.

Any prefix specified is not retained across separate calls to **gentemp**.

Examples:

```
> (symbol-name (gentemp))
"T0"
> (find-symbol "T0")
T0
:INTERNAL
> (symbol-name (gentemp "likely-unique" (find-package 'lisp)))
"likely-unique0"
> (find-symbol "likely-unique0")
NIL
NIL
> (find-symbol "likely-unique0" (find-package 'lisp))
LISP::|likely-unique0|
:INTERNAL
```

See Also: **gensym**

get

Purpose: The function **get** searches a symbol's property list for an indicator identical (**eq**) to its *indicator* argument. If one is found, the value associated with it is returned. If no such indicator is found and the *default* argument is specified, the default value is returned; otherwise **get** returns **nil**.

Syntax: **get** *symbol indicator* `&optional` *default* [*Function*]

Remarks: The macro **setf** may be used with **get** to replace the value associated with a property name or to insert a new property-value pair.

Examples:

```
> (setq *symbol* (gensym))
#:G2
> (setq *indicator* (gensym))
#:G3
> (setq *value* (gensym))
#:G4
> (get *symbol* *indicator*)
NIL
> (get *symbol* *indicator* 'foo)
FOO
> (setf (get *symbol* *indicator*) *value*)
#:G4
> (eq (get *symbol* *indicator* 'foo) *value*)
T
```

See Also: **getf**

symbol-plist

getf, get-properties

Purpose: The functions **getf** and **get-properties** are used to access property list entries.

The function **getf** searches the property list stored in *place*. It returns the property value associated with the indicator that is identical (**eq**) to the *indicator* argument. If the given indicator is not found and a *default* value is specified, then *default* is returned; otherwise **getf** returns **nil**.

The function **get-properties** is like **getf** except that its second argument is a list of indicators, and it returns three values. It searches the property list for the first entry whose indicator is identical (**eq**) to one of the indicators in the indicator list. If such an entry is found, the first two values returned are the indicator and its associated property value, and the third value is the tail of the sublist of the property list whose **car** is the indicator. If no such entry is found, all three values are **nil**.

Syntax:

getf *place indicator* `&optional` *default* [*Function*]

get-properties *place indicator-list* [*Function*]

Remarks: If the *place* argument is a generalized variable acceptable to the macro **setf**, then **setf** may be used with the function **getf** to replace the value associated with an indicator or to create a new entry (indicator and property value).

Examples:

```
> (setq x (cons () ()))
(NIL)
> (setq *indicator-list* '(prop1 prop2))
(PROP1 PROP2)
> (getf (car x) 'prop1)
NIL
> (setf (getf (car x) 'prop1) 'val1)
VAL1
> (eq (getf (car x) 'prop1) 'val1)
T
> (get-properties (car x) *indicator-list*)
PROP1
VAL1
(PROP1 VAL1)
```

See Also: **get**

keywordp

Purpose: The predicate **keywordp** is true if its argument is a symbol and if that symbol belongs to the keyword package; otherwise it is false.

Syntax: **keywordp** *object* [*Function*]

Remarks: A keyword is written with a leading colon.

A keyword is a constant and evaluates to itself.

Examples:

```
> (keywordp :optional)
T
> (keywordp ':optional)
T
> (keywordp '(:optional))
NIL
> (keywordp 'optional)
NIL
```

make-symbol

Purpose: The function **make-symbol** creates and returns a new uninterned symbol whose print name is the string *print-name*. The contents of the symbol's value and function cells are undefined, and the symbol's property list is empty.

Syntax: **make-symbol** *print-name* [*Function*]

Examples:

```
> (make-symbol "foo")
#:|foo|
> (symbol-name (make-symbol "bar"))
"bar"
> (find-symbol (symbol-name (make-symbol "baz")))
NIL
NIL
```

See Also: **copy-symbol**

remf

Purpose: The macro **remf** is used to remove an entry from a property list. It removes from the property list found in *place* both the property whose indicator is identical (**eq**) to the *indicator* argument and its associated property value. The function **remf** returns a non-nil value if the property was found; otherwise it returns **nil**.

Syntax: **remf** *place indicator* [*Macro*]

Remarks: The *place* argument must be a generalized variable acceptable to the macro **setf**.

Examples:

```
> (setq x (cons () ()))
(NIL)
> (setf (getf (car x) 'prop1) 'val1)
VAL1
> (remf (car x) 'prop1)
T
> (remf (car x) 'prop1)
NIL
```

See Also: **remprop**

getf

remprop

Purpose: The function **remprop** is used to remove an entry from the property list of a symbol. It removes from the property list of *symbol* both the property whose indicator is identical (**eq**) to the *indicator* argument and its associated property value. The function **remprop** returns a non-**nil** value if the specified property was found; otherwise it returns **nil**.

Syntax: **remprop** *symbol indicator* [*Function*]

Examples:

```
> (setq *symbol* (gensym))
#:G51
> (remprop *symbol* 'prop1)
NIL
> (setf (get *symbol* 'prop1) 'val1)
VAL1
> (remprop *symbol* 'prop1)
(PROP1 VAL1)
> (remprop *symbol* 'prop1)
NIL
```

See Also: **remf**

symbol-name

Purpose: The function **symbol-name** returns the print name of a given symbol.

Syntax: **symbol-name** *symbol* [*Function*]

Examples:

```
> (symbol-name 'foo)
"FOO"
> (symbol-name (gensym))
"G52"
```

symbol-package

Purpose: The function **symbol-package** returns the contents of the package cell of a given symbol. If no package object is associated with this cell, **symbol-package** returns **nil**.

Syntax: **symbol-package** *symbol* [*Function*]

Examples:

```
> (symbol-package :optional)
#<Package "KEYWORD" 30DC43>
> (symbol-package (gensym))
NIL
```

symbol-plist

Purpose: The function **symbol-plist** returns the property list of a given symbol.

Syntax: **symbol-plist** *symbol* [*Function*]

Remarks: The macro **setf** may be used with **symbol-plist** to replace the property list of a symbol.

Examples:

```
> (defvar *sym* (gensym))
*SYM*
> (symbol-plist *sym*)
NIL
> (setf (get *sym* 'prop1) 'val1)
VAL1
> (symbol-plist *sym*)
(PROP1 VAL1)
> (setf (get *sym* 'prop2) 'val2)
VAL2
> (symbol-plist *sym*)
(PROP2 VAL2 PROP1 VAL1)
> (setf (symbol-plist *sym*) '(prop3 val3))
(PROP3 VAL3)
> (symbol-plist *sym*)
(PROP3 VAL3)
```

symbolp

Purpose: The predicate **symbolp** is true if its argument is a symbol; otherwise it is false.

Syntax: **symbolp** *object* [*Function*]

Examples:

```
> (symbolp 'sss)
T
> (symbolp 12)
NIL
> (symbolp nil)
T
> (symbolp :test)
T
```

Chapter 11. Packages

Chapter 11. Packages

About Packages

A ***package*** is a Common Lisp object that specifies a correspondence between print name strings and symbols. The package facility may be used to create a hierarchical program name space and to increase program modularity. Packages enable the user to avoid name conflicts that may arise when separate modules become part of the same system. By the use of packages, two different modules using the same name for different internal purposes can do so safely and without name conflicts. There are also constructs that are designed to enable a package to reference the symbols of other packages in a convenient and transparent manner.

Package names are unique. In addition to its name, a package may also have nicknames. A package renaming operation is available, should conflicts among package names arise. Like symbol names, when package names are read by the Lisp reader, lowercase characters are converted to uppercase unless they are preceded by the escape character \ or enclosed by the | multiple escape characters. Operations that compare package names are sensitive to these conventions.

The symbols whose string-to-symbol mappings are defined within the package are said to be ***present*** in the package. Each such symbol is either an ***external*** or an ***internal*** symbol of that package, but not both. An external symbol of a package is part of that package's public interface and is accessible to other packages. An internal symbol is intended for the private use of the package.

Names within a package are unique. Two different symbols with the same name may only exist in separate packages. Note, however, that the same symbol may be an external symbol in some packages and an internal symbol in others.

A symbol is ***owned*** by only one package. The package that owns the symbol is called the symbol's ***home package***. The ***package cell*** of the symbol specifies the symbol's home package. If a symbol is owned by a package, it is said to be an ***interned*** symbol. Symbols not owned by any package are ***uninterned*** symbols. The package cell of an uninterned symbol is **nil**. The name of an uninterned symbol is printed with a leading #:. The **symbol-package** function may be used to determine a symbol's home package.

A symbol is ***accessible*** in a package if it is present in the package or if it has been ***inherited*** from some other package by means of the **use-package** construct. Only the external symbols of a package may be inherited by some other package.

Only one package is ***current*** at any given time. The Lisp reader interprets names as symbols according to the mappings specified by the current package. The current package is the package that is specified by the global variable ***package***.

Any symbol in any package can be referenced, no matter what the current package is. An external, internal, or inherited symbol of the current package may simply be referenced by its name. To reference an external symbol of some other package, the symbol name is ***qualified*** by preceding it with the package name and one colon. To reference a symbol of some other package without regard to whether it is an external or an internal symbol, the symbol name is qualified by preceding it with the package name and two colons. Since internal symbols are normally intended for the private use of a package, accessing the internal symbols of a package that is not current may cause the integrity of the package system to be violated.

There are several functions that influence which symbols are accessible in a package. These functions are briefly described here. For a complete description, the user is referred to the individual function pages.

A package controls which of its symbols may be inherited by another package by means of the **export** construct. The external symbols of a package are those symbols that have been exported from the package. Only exported symbols may be inherited. It is customary to list all external symbols of a package with an **export** at the beginning of the definition of the package.

The **use-package** construct causes the external symbols of the used package to become inherited symbols of the using package. As such they are accessible in the using package. It is not necessary to qualify their names when the using package is current. If any external symbols are added to the used package, they are automatically inherited by the using package.

The function **intern** may be used to create a new symbol and enter it into a package as an internal symbol, as long as a symbol with that name is not already accessible in the package. If such a symbol already exists, the existing symbol is simply returned.

The function **import** may be used to enter any existing symbol into a package as an internal symbol, as long as a symbol with that name is not already accessible in the package. The **shadowing-import** function is used to import a symbol without regard to whether another symbol with the same name is already accessible. The function **shadow** checks whether a symbol with a given name is already present in a package and, if it is not, causes one with that name to be created as an internal symbol of the package.

The functions **unintern**, **unexport**, and **unuse-package** are used to undo the effects of **intern**, **export**, and **use-package** respectively.

A symbol name conflict exists whenever a name can be interpreted as any of two or more different symbols. Sun Common Lisp is designed so that name conflicts will never arise without being noticed. Whenever a function changes the package environment, the changes are carefully checked and an error is signaled if a name conflict occurs. The matching of the names of symbols and of packages is case sensitive.

Built-in Packages

Sun Common Lisp has four basic packages: lisp, user, keyword, and system.

- The ***lisp package*** contains the basis of the Common Lisp system. All the predefined Common Lisp functions, macros, constants, variables, and special forms, are external symbols in the lisp package. As a result, virtually all other packages use the lisp package.
- The ***user package*** is the package that normally becomes current when Common Lisp is started. The user package uses the lisp package.
- The ***keyword package*** consists of all the keyword symbols. Symbols in the keyword package are always external, and all are constants that evaluate to themselves. This eliminates the need to quote keywords in function calls. The names of keywords always start with a colon. Keywords should never be imported. Other packages may not use the keyword package.
- The ***system package*** is an implementation-dependent package reserved for internal system functions. It has the nickname `sys`. The system package uses the lisp package. Any symbol that is the name of a construct that is an extension to Common Lisp described in this manual is an external symbol in the system package.

Loading Files into Packages

The normal way of specifying the package into which a file is to be loaded is to begin the file with a call to **in-package**. This function accepts a package name, a nickname list, and a use-package list. If the **in-package** construct is not used, the file will be loaded into the current package. If a file does not specify what package it should be in, by means of the **in-package** construct, the name of the package that was current when the file was compiled is not retained. Unpredictable results may occur if the package that is current at load time is different from the package that is current at compilation.

Whenever the function **load** is used, it remembers the initial value of ***package*** (the current package) and restores that value to ***package*** after the file loading has finished. Thus, using **load** to process a file will always preserve the current package, even if the file calls **in-package**.

Modules

A ***module*** is a collection of one or more files that are always loaded together to provide some particular capability. A program can indicate that it wants a particular module to be loaded if and only if that module has not already been loaded. To make this work, Common Lisp keeps a list, in the variable ***modules***, of the names of modules that have been loaded. The function **provide** is used to update the ***modules*** list to indicate that a given module has been loaded. Thus, one call to **provide** should occur in each module. The function **require** names a module that is needed; if the module has not already been loaded, it is loaded from the pathname(s) specified in the call to **require**.

Categories of Operations

This section groups operations on packages according to functionality.

Data Type Predicates

packagep

This predicate determines whether an object is a package.

Operations on Packages

delete-package	***package***
do-all-symbols	**package-name**
do-external-symbols	**package-nicknames**
do-symbols	**package-shadowing-symbols**
export	**package-use-list**
find-all-symbols	**package-used-by-list**
find-package	**rename-package**
find-symbol	**shadow**
import	**shadowing-import**
in-package	**unexport**
intern	**unintern**
list-all-packages	**unuse-package**
make-package	**use-package**

These constructs manipulate packages.

Operations on Modules

provide	***modules***
require	

These constructs manipulate modules.

delete-package

Purpose: The function **delete-package** is used to remove a package from the system.

After the deletion, the function **find-package** will no longer find a package of that name or of any of its nicknames.

Any symbol whose home package was the deleted package becomes an uninterned symbol.

Syntax: **delete-package** *package* [*Function*]

Remarks: Once the package has been deleted, its symbols are no longer inherited by any other package. The deleted package is removed from the package-used-by-list of any other package.

An error is signaled if the package is on the package-use-list of any other package.

The *package* argument may be either a package, a string, or a symbol. If a symbol is given, its print name is used.

The function **delete-package** is an extension to Common Lisp.

Examples:

```
> (make-package 'temporary :nicknames '(temp))
#<Package "TEMPORARY" 3C7BOB>
> (find-package 'temp)
#<Package "TEMPORARY" 3C7BOB>
> (delete-package 'temp)
T
> (find-package 'temp)
NIL
```

do-symbols, do-external-symbols, do-all-symbols

Purpose: The macros **do-symbols**, **do-external-symbols**, and **do-all-symbols** iterate over the symbols of a package.

The macro **do-symbols** iterates over all the symbols accessible in a specified package.

The macro **do-external-symbols** iterates over all the external symbols of a specified package.

The macro **do-all-symbols** iterates over all the symbols that are present in any package.

For each symbol in the set, the variable *var* is bound to the symbol, and the statements in the body are executed. When all such symbols have been processed, the *result-form* is evaluated and returned as the value of the macro. If the *result-form* is not specified, the macro returns **nil**.

Syntax:

do-symbols (*var* [*package* [*result-form*]]) {*declaration*}* {*tag* | *statement*}* [*Macro*]

do-external-symbols (*var* [*package* [*result-form*]]) {*declaration*}* {*tag* | *statement*}* [*Macro*]

do-all-symbols (*var* [*result-form*]) {*declaration*}* {*tag* | *statement*}* [*Macro*]

Remarks: If execution of any statement in the body affects which symbols are present in the package, the results are unpredictable.

The macro **do-all-symbols** may cause a symbol that is present in several packages to be processed more than once.

The *package* argument may be either a package, a string, or a symbol. If a symbol is given, its print name is used. If the *package* argument of **do-symbols** or **do-external-symbols** is not specified, the current package is used.

do-symbols, do-external-symbols, do-all-symbols

Examples:

```
> (make-package 'temp :use nil)
#<Package "TEMP" 42D7EB>
> (intern "SHY" 'temp)                  ;shy will be an internal symbol
TEMP::SHY                               ;in the package temp
NIL
> (export (intern "BOLD" 'temp) 'temp)  ;bold will be external
T
> (let ((lst ()))
    (do-symbols (s 'temp) (push s lst))
    lst)
(TEMP::SHY TEMP:BOLD)
> (let ((lst ()))
    (do-external-symbols (s 'temp lst) (push s lst)))
(TEMP:BOLD)
> (let ((lst ()))
    (do-all-symbols (s lst)
      (when (eq (find-package 'temp) (symbol-package s)) (push s lst))))
(TEMP::SHY TEMP:BOLD)
```

export

Purpose: The function **export** is used to make a symbol that is accessible in a package an external symbol of that package. The exported symbol may be present in the package or inherited from some other package.

If the symbol is present as an internal symbol in the package, it is made external. If it is an internal symbol that is inherited, it is imported into the package and then made an external symbol in that package. If the symbol is already an external symbol in the package, **export** has no effect.

The *symbols* argument is a single symbol or a list of symbols to be exported.

The function **export** returns **t** as its result.

Syntax: **export** *symbols* `&optional` *package* [*Function*]

Remarks: A continuable error is signaled if the symbol is not accessible in the package or if a name conflict occurs.

The *package* argument may be either a package, a string, or a symbol. If a symbol is given, its print name is used. If the *package* argument is not specified, the current package is used.

When **export** occurs at the top level in a file, it is implicitly wrapped in an `(eval-when (eval compile load) ...)` construct, so that package systems changes are reflected both in the compiling and in the loading environment.

Examples:

```
> (make-package 'temp :use nil)
#<Package "TEMP" 49024B>
> (use-package 'temp)
T
> (intern "TEMP-SYM" 'temp)
TEMP::TEMP-SYM
NIL
> (find-symbol "TEMP-SYM")
NIL
NIL
> (export (find-symbol "TEMP-SYM" 'temp) 'temp)
T
> (find-symbol "TEMP-SYM")
TEMP-SYM
:INHERITED
```

See Also: **import**

unexport

find-all-symbols

Purpose: The function **find-all-symbols** is used to find any symbol whose print name is specified by the *string-or-symbol* argument. All packages are searched.

The function **find-all-symbols** returns a list of all such symbols as its result.

Syntax: **find-all-symbols** *string-or-symbol* [*Function*]

Remarks: If a symbol argument is given, the symbol's print name is used.

The matching of the names of symbols is case sensitive.

Examples:

```
> (find-all-symbols 'car)
(CAR)
> (intern "CAR" (package-name (make-package 'temp :use nil)))
TEMP::CAR
NIL
> (find-all-symbols 'car)
(TEMP::CAR CAR)         ;order in the result list is not significant
> (delete-package 'temp)
T
> (find-all-symbols 'car)
(CAR)
```

find-package

Purpose: The function **find-package** returns the package whose name or nickname is *name*. If there is no such package, **find-package** returns **nil**.

Syntax: **find-package** *name* [*Function*]

Remarks: The **name** argument may be either a string or symbol. If a symbol is given, its print name is used.

The matching of the names of packages is case sensitive.

Examples:

```
> (find-package 'lisp)
#<Package "LISP" 2D0003>
> (find-package "USER")
#<Package "USER" 2D1023>
> (find-package 'not-there)
NIL
```

find-symbol

Purpose: The function **find-symbol** is used to locate a symbol in a package.

The symbols accessible in the package are searched for one whose name is the same as the *string* argument.

The function **find-symbol** returns two values. The first value is the symbol that was found. The second value indicates the status of that symbol. If the symbol was present in the given package as an internal symbol, it is **:internal**. If the symbol was present in the package as an external symbol, it is **:external**. If the symbol was inherited by the package through the **use-package** construct, it is **:inherited**. If the symbol was not found, both values are **nil**.

Syntax: **find-symbol** *string* `&optional` *package* [*Function*]

Remarks: The *package* argument may be either a package, a string, or a symbol. If a symbol is given, its print name is used. If the *package* argument is not specified, the current package is used.

Examples:

```
> (find-symbol (intern "NEW-GUY"))
NEW-GUY
:INTERNAL
> (find-symbol 'car 'user)
CAR
:INHERITED
> (find-symbol 'car 'lisp)
CAR
:EXTERNAL
> (find-symbol "Not-Likely")
NIL
NIL
```

See Also: **intern**

import

Purpose: The function **import** is used to add a symbol to a package. The imported symbol becomes present as an internal symbol in the package. Once the symbol has been imported, it may be referenced in the importing package without the use of a qualifier.

The *symbols* argument is a single symbol or a list of symbols to be imported.

The function **import** returns `t`.

Syntax: **import** *symbols* `&optional` *package* [*Function*]

Remarks: If the symbol is already present in the package, **import** has no effect.

Importing a symbol does not affect its status in the package from which it is imported, if any.

A continuable error is signaled if a different symbol with the same name is accessible in the package.

The *package* argument may be either a package, a string, or a symbol. If a symbol is given, its print name is used. If the *package* argument is not specified, the current package is used.

When **import** occurs at the top level in a file, it is implicitly wrapped in an `(eval-when (eval compile load) ...)` construct, so that package systems changes are reflected both in the compiling and in the loading environment.

Examples:

```
> (import 'lisp::car (make-package 'temp :use nil))
T
> (find-symbol "CAR" 'temp)
CAR
:INTERNAL
> (find-symbol "CDR" 'temp)
NIL
NIL
```

See Also: **shadow**

in-package

Purpose: The function **in-package** changes the current package. Its main use is to specify the package into which a file should be loaded.

The **:nicknames** argument may be used to specify a list of alternative names for the package.

The **:use** argument may be used to specify a list of packages whose external symbols are to be inherited by the new package. If the **:use** argument is not specified, the lisp package is inherited.

If a package with the given name already exists, it is updated to reflect any new nicknames or used packages that are specified by the arguments. If a package with the given name does not already exist, a new package is created.

Syntax: **in-package** *package-name* `&key :nicknames :use` [*Function*]

Remarks: The function **in-package** causes the ***package*** variable to be reset to the package with the given name. The ***package*** variable retains this value until it is changed by the user or until the loading operation has completed, at which time the **load** function resets ***package*** to the value it had before the loading was begun.

If the **in-package** construct is not used, the file is loaded into the current package.

The *package-name* argument and the nicknames may be either strings or symbols. If a symbol is given, its print name is used.

When **in-package** occurs at the top level in a file, it is implicitly wrapped in an `(eval-when (eval compile load) ...)` construct, so that package systems changes are reflected both in the compiling and in the loading environment.

Examples:

```
> (in-package 'temporary :nicknames '("TEMP"))
#<Package "TEMPORARY" 494753>
> *package*
#<Package "TEMPORARY" 494753>
> (find-symbol "CAR")
CAR
:INHERITED
```

See Also: **make-package**

intern

Purpose: The function **intern** is used to enter a symbol into a package.

If a symbol whose name is the same as the string argument is already accessible in the package as an external, internal, or inherited symbol, it is returned. If no such symbol is accessible in the package, a new symbol with the given name is created and entered into the package as an internal symbol.

The function **intern** returns two values. The first value is the symbol that was found or created. The second value indicates the status of that symbol. If a new symbol was created, the second value is **nil**. If the symbol was present in the given package as an internal symbol, it is **:internal**. If the symbol was present in the package as an external symbol, it is **:external**. If the symbol was inherited by the package through the **use-package** construct, it is **:inherited**.

Syntax: **intern** *string* `&optional` *package* [*Function*]

Remarks: If a new symbol is entered into the keyword package, it becomes an external symbol in that package.

The *package* argument may be either a package, a string, or a symbol. If a symbol is given, its print name is used. If the *package* argument is not specified, the current package is used.

When **intern** occurs at the top level in a file, it is implicitly wrapped in an `(eval-when (eval compile load) ...)` construct, so that package systems changes are reflected both in the compiling and in the loading environment.

Examples:

```
> (in-package 'user)
#<Package "USER" 2D1023>
> (intern "Never-Before")
|Never-Before|
NIL
> (intern "Never-Before")
|Never-Before|
:INTERNAL
```

list-all-packages

Purpose: The function **list-all-packages** returns a list of all existing packages.

Syntax: **list-all-packages** *[Function]*

Examples:

```
> (let ((before (list-all-packages)))
    (make-package 'temp)
    (set-difference (list-all-packages) before))
(#<Package "TEMP" 4973A3>)
```

make-package

Purpose: The function **make-package** creates and returns a new package with the name *package-name.*

The **:nicknames** argument may be used to specify a list of alternative names for the package.

The **:use** argument may be used to specify a list of packages whose external symbols will be inherited by the new package. If the **:use** argument is not specified, the lisp package will be inherited.

Syntax: **make-package** *package-name* `&key :nicknames :use` *[Function]*

Remarks: Any packages specified by the **:use** argument must already exist.

The *package-name* argument and the nicknames may be either strings or symbols. If a symbol is given, its print name is used. These names must not conflict with any existing package names. A continuable error is signaled if they do.

When **make-package** occurs at the top level in a file, it is implicitly wrapped in an `(eval-when (eval compile load) ...)` construct, so that package systems changes are reflected both in the compiling and in the loading environment.

Examples:

```
> (make-package 'temporary :nicknames '("TEMP" "temp"))
#<Package "TEMPORARY" 499413>
> (make-package "OWNER" :use '("temp"))
#<Package "OWNER" 4998BB>
> (package-used-by-list 'temp)
(#<Package "OWNER" 4998BB>)
> (package-use-list 'owner)
(#<Package "TEMPORARY" 499413>)
```

See Also: **use-package**

modules

Purpose: The variable ***modules*** is used to keep track of all the modules that have been loaded into the system. Its value is a list of the names of these modules.

Syntax: ***modules*** *[Variable]*

Examples:

```
> (member "MANGANESE" *modules* :test #'equal)
NIL
> (provide 'manganese)
T
> (member "MANGANESE" *modules* :test #'equal)
("MANGANESE")
```

See Also: **provide**

require

package

Purpose: The value of the variable ***package*** is the current package.

Syntax: ***package*** *[Variable]*

Remarks: The value of ***package*** when Common Lisp is started is normally the user package.

The value of ***package*** may be affected during the course of the **load** operation. The value of ***package*** at the end of the **load** operation is guaranteed to be the same as its value before the operation was begun.

Examples:

```
> (let ((curpack *package*)
        out)
    (in-package 'lisp)
    (setq out *package*)
    (in-package (package-name curpack))
    out)
#<Package "LISP" 2D0003>
```

package-name

Purpose: The function **package-name** returns the name of the given package. Its result is a string.

Syntax: **package-name** *package* [*Function*]

Examples:

```
> (in-package 'user)
#<Package "USER" 2D1023>
> (package-name *package*)
"USER"
> (package-name (symbol-package :test))
"KEYWORD"
> (package-name (find-package 'lisp))
"LISP"
```

package-nicknames

Purpose: The function **package-nicknames** returns a list of all the nicknames for the given package. The nicknames are returned as strings. This list does not include the package name itself.

Syntax: **package-nicknames** *package* [*Function*]

Examples:

```
> (package-nicknames (make-package 'temporary
                                   :nicknames '("TEMP" "temp")))
("temp" "TEMP")
```

package-shadowing-symbols

Purpose: The function **package-shadowing-symbols** returns the shadowing-symbols list of the specified package.

Shadowing symbols are symbols that have been entered into a package by the use of **shadow** or **shadowing-import**.

Syntax: **package-shadowing-symbols** *package* [*Function*]

Remarks: The *package* argument may be either a package, a string, or a symbol. If a symbol is given, its print name is used.

Examples:

```
> (package-shadowing-symbols (make-package 'temp))
NIL
> (shadow 'cdr 'temp)
T
> (package-shadowing-symbols 'temp)
(TEMP::CDR)
> (intern "PILL" 'temp)
TEMP::PILL
NIL
> (shadowing-import 'pill 'temp)
T
> (package-shadowing-symbols 'temp)
(PILL TEMP::CDR)
```

See Also: **shadow**

shadowing-import

package-use-list

Purpose: The function **package-use-list** returns a list of the packages that are used by the specified package.

Syntax: **package-use-list** *package* [*Function*]

Remarks: The *package* argument may be either a package, a string, or a symbol. If a symbol is given, its print name is used.

Examples:

```
> (package-use-list (make-package 'temp))
(#<Package "LISP" 2D0003>)
> (use-package 'user 'temp)
T
> (package-use-list 'temp)
(#<Package "LISP" 2D0003> #<Package "USER" 2D1023>)
```

See Also: **use-package**

unuse-package

package-used-by-list

Purpose: The function **package-used-by-list** returns a list of the packages that use the specified package.

Syntax: **package-used-by-list** *package* [*Function*]

Remarks: The *package* argument may be either a package, a string, or a symbol. If a symbol is given, its print name is used.

Examples:

```
> (package-used-by-list (make-package 'temp))
NIL
> (make-package 'trash :use '(temp))
#<Package "TRASH" 4A1E53>
> (package-used-by-list 'temp)
(#<Package "TRASH" 4A1E53>)
```

See Also: **use-package**

unuse-package

packagep

Purpose: The predicate **packagep** is true if its argument is a package; otherwise it is false.

Syntax: **packagep** *object* [*Function*]

Examples:

```
> (packagep *package*)
T
> (packagep 'lisp)
NIL
```

provide

Purpose: The function **provide** is used to indicate that a module has been loaded. It adds the module's name to the ***modules*** list.

Syntax: **provide** *module-name* [*Function*]

Remarks: One call to **provide** should occur at the head of each module. It should specify the name by which that module is to be known to the system.

The *module-name* argument may be either a string or a symbol. If a symbol is given, its print name is used.

Examples:

```
> (let ((omods *modules*))
    (provide 'new-module)
    (set-difference *modules* omods))
("NEW-MODULE")
```

See Also: ***modules***

rename-package

Purpose: The function **rename-package** is used to replace the name and nicknames of a package.

The package is renamed *new-name*. If a list of nicknames is specified, the package will have these new nicknames. If the list of nicknames is not specified, the package will have no nicknames.

Syntax: **rename-package** *package new-name* &optional *new-nicknames* [*Function*]

Remarks: The *new-name* argument may be either a string or a symbol. The *new-nicknames* argument is a list of strings or symbols. If a symbol is given, its print name is used. These names must not conflict with any existing package names.

The *package* argument may be either a package, a string, or a symbol. If a symbol is given, its print name is used.

Examples:

```
> (make-package 'temporary :nicknames '("TEMP"))
#<Package "TEMPORARY" 4A28A3>
> (rename-package 'temp 'ephemeral)
#<Package "EPHEMERAL" 4A28A3>
> (package-nicknames (find-package 'ephemeral))
NIL
> (find-package 'temporary)
NIL
> (rename-package 'ephemeral 'temporary '(temp fleeting))
#<Package "TEMPORARY" 4A28A3>
> (package-nicknames (find-package 'temp))
("FLEETING" "TEMP")
```

require

Purpose: The function **require** tests whether a given module has been loaded. If not, it causes the module to be loaded.

The *pathname* argument may be used to specify a list of files that are to be loaded in order to achieve the loading of the module.

Syntax: **require** *module-name* `&optional` *pathname* [*Function*]

Remarks: The function **require** uses the ***modules*** variable to check whether a module has been loaded. Whenever a new module is loaded, **require** updates the ***modules*** variable.

The *module-name* argument may be either a string or a symbol. If a symbol is given, its print name is used.

The *pathname* argument may be a single file pathname or a list of pathnames. If it is a list, the files are loaded in the order given.

When **require** occurs at the top level in a file, it is implicitly wrapped in an `(eval-when (eval compile load) ...)` construct, so that package systems changes are reflected both in the compiling and in the loading environment.

Examples:

```
;;; the file /test/require-test.lisp must exist and contain
;;;
;;; (provide 'test-module)
;;;
;;; (setq loaded-flag t)
;;;
;;; for this example to work

> (setq loaded-flag nil)
NIL
> (require 'test-module "/test/require-test")
T
> loaded-flag
T
> (find "TEST-MODULE" *modules* :test #'equal)
"TEST-MODULE"
> (setq loaded-flag nil)
NIL
```

```
> (require 'test-module "/test/require-test")
T
> loaded-flag
NIL
```

See Also: provide

modules

shadow

Purpose: The function **shadow** is used to add a new symbol to a package.

If a symbol with the given name is already present in the package, then **shadow** has no effect. Otherwise a new symbol whose name is the same as the print name of the given symbol is created and entered into the package as an internal symbol. The new symbol is also added to the package's shadowing-symbols list.

The *symbols* argument is a single symbol or a list of symbols to be added.

The function **shadow** returns **t** as its result.

Syntax: **shadow** *symbols* `&optional` *package* [*Function*]

Remarks: The *package* argument may be either a package, a string, or a symbol. If a symbol is given, its print name is used. If the *package* argument is not specified, the current package is used.

When **shadow** occurs at the top level in a file, it is implicitly wrapped in an `(eval-when (eval compile load) ...)` construct, so that package systems changes are reflected both in the compiling and in the loading environment.

Examples:

```
> (package-shadowing-symbols (make-package 'temp))
NIL
> (find-symbol 'car 'temp)
CAR
:INHERITED
> (shadow 'car 'temp)
T
> (find-symbol 'car 'temp)
TEMP::CAR
:INTERNAL
> (package-shadowing-symbols 'temp)
(TEMP::CAR)
```

shadowing-import

Purpose: The function **shadowing-import** is used to import a symbol whose name is the same as a symbol that is already accessible in a package.

The *symbols* argument is a single symbol or a list of symbols to be imported.

The new symbol is added to the shadowing-symbols list of the package.

If a symbol that is shadowed is present in the package, it will be uninterned from the package.

The function **shadowing-import** returns **t** as its result.

Syntax: **shadowing-import** *symbols* `&optional` *package* [*Function*]

Remarks: The *package* argument may be either a package, a string, or a symbol. If a symbol is given, its print name is used. If the *package* argument is not specified, the current package is used.

When **shadowing-import** occurs at the top level in a file, it is implicitly wrapped in an `(eval-when (eval compile load) ...)` construct, so that package systems changes are reflected both in the compiling and in the loading environment.

Examples:

```
> (in-package 'user)
#<Package "USER" 2D1023>
> (setq sym (intern "CONFLICT"))
CONFLICT
> (intern "CONFLICT" (make-package 'temp))
TEMP::CONFLICT
NIL
> (package-shadowing-symbols 'temp)
NIL
> (shadowing-import sym 'temp)
T
> (package-shadowing-symbols 'temp)
(CONFLICT)
```

See Also: **import**

unintern

unexport

Purpose: The function **unexport** returns external symbols in a package to internal status. It is used to undo the effect of **export**.

The *symbols* argument is a single symbol or a list of symbols to be unexported.

The function **unexport** returns **t** as its result.

If the symbol is already an internal symbol in the package, **unexport** has no effect.

Syntax: **unexport** *symbols* `&optional` *package* [*Function*]

Remarks: The symbol must be present in the package.

The *symbols* argument cannot specify a symbol from the keyword package.

The *package* argument may be either a package, a string, or a symbol. If a symbol is given, its print name is used. If the *package* argument is not specified, the current package is used.

When **unexport** occurs at the top level in a file, it is implicitly wrapped in an `(eval-when (eval compile load) ...)` construct, so that package systems changes are reflected both in the compiling and in the loading environment.

Examples:

```
> (in-package 'user)
#<Package "USER" 2D1023>
> (export (intern "CONTRABAND" (make-package 'temp)) 'temp)
T
> (find-symbol "CONTRABAND")
NIL
NIL
> (use-package 'temp)
T
> (find-symbol "CONTRABAND")
CONTRABAND
:INHERITED
> (unexport 'contraband 'temp)
T
> (find-symbol "CONTRABAND")
NIL
NIL
```

See Also: **export**

unintern

Purpose: The function **unintern** is used to remove a symbol from a package.

If the symbol is present in the package, it is removed from the package. If it is a shadowing symbol, it is also removed from the shadowing-symbols list of the package.

If the package is the symbol's home package, the symbol will have no home package.

The function **unintern** returns **t** if the symbol was removed from the package; otherwise it returns **nil**.

Syntax: **unintern** *symbol* `&optional` *package* [*Function*]

Remarks: Even if a symbol has been uninterned, it may still be accessible if it is inherited from another package.

The *package* argument may be either a package, a string, or a symbol. If a symbol is given, its print name is used. If the *package* argument is not specified, the current package is used.

When **unintern** occurs at the top level in a file, it is implicitly wrapped in an `(eval-when (eval compile load) ...)` construct, so that package systems changes are reflected both in the compiling and in the loading environment.

Examples:

```
> (in-package 'user)
#<Package "USER" 2D1023>
> (setq temps-foo (intern "FOO" (make-package 'temp)))
TEMP::FOO
> (unintern temps-foo 'temp)
T
> (find-symbol "FOO" 'temp)
NIL
NIL
> temps-foo
#:FOO
```

unuse-package

Purpose: The function **unuse-package** is used to cause a package to cease inheriting all the external symbols of some other package; it undoes the effects of **use-package**. The packages that are unused are removed from the use-list of the unusing package.

The function **unuse-package** returns **t** as its result.

Syntax: **unuse-package** *packages-to-unuse* `&optional` *package* [*Function*]

Remarks: The *packages-to-unuse* argument may be either a package, a package name, or a list of these. Either a symbol or a string may be given as a package name. If a symbol is given, its print name is used.

If the *package* argument is not specified, the current package is used.

When **unuse-package** occurs at the top level in a file, it is implicitly wrapped in an `(eval-when (eval compile load) ...)` construct, so that package systems changes are reflected both in the compiling and in the loading environment.

Examples:

```
> (in-package 'user)
#<Package "USER" 2D1023>
> (export (intern "SHOES" (make-package 'temp)) 'temp)
T
> (find-symbol "SHOES")
NIL
NIL
> (use-package 'temp)
T
> (find-symbol "SHOES")
SHOES
:INHERITED
> (find (find-package 'temp) (package-use-list 'user))
#<Package "TEMP" 3CC483>
> (unuse-package 'temp)
T
> (find-symbol "SHOES")
NIL
NIL
```

See Also: **use-package**

use-package

Purpose: The function **use-package** is used to cause a package to inherit all the external symbols of another package. The inherited symbols become accessible as internal symbols of the using package. The packages that are used are added to the use-list of the using package.

The function **use-package** returns **t** as its result.

Syntax: **use-package** *packages-to-use* `&optional` *package* [*Function*]

Remarks: The keyword package may not be used.

The *packages-to-use* argument may be either a package, a package name, or a list of these. Either a symbol or a string may be given as a package name. If a symbol is given, its print name is used.

If the *package* argument is not specified, the current package is used.

When **use-package** occurs at the top level in a file, it is implicitly wrapped in an `(eval-when (eval compile load) ...)` construct, so that package systems changes are reflected both in the compiling and in the loading environment.

Examples:

```
> (export (intern "LAND-FILL" (make-package 'trash)) 'trash)
T
> (find-symbol "LAND-FILL" (make-package 'temp))
NIL
NIL
> (package-use-list 'temp)
(#<Package "LISP" 2D0003>)
> (use-package 'trash 'temp)
T
> (package-use-list 'temp)
(#<Package "LISP" 2D0003> #<Package "TRASH" 3CDC83>)
> (find-symbol "LAND-FILL" 'temp)
TRASH:LAND-FILL
:INHERITED
```

See Also: **unuse-package**

Chapter 12. Numbers

Chapter 12. Numbers

About Numbers

Common Lisp provides integers, ratios, floating-point numbers, and complex numbers as separate data types. Integers and ratios together constitute a subtype of numbers called ***rational numbers***.

Some operations on numbers are generic, that is, they accept arguments of any numerical data type and automatically provide any type conversions that are needed. Other operations are type specific and require arguments of a particular numerical data type.

Numbers in Common Lisp are ordinarily not true objects in the sense that **eq** cannot be counted on to operate reliably on them. In Sun Common Lisp, however, fixnums are true objects and **eq** can be counted on to operate reliably on fixnums. Otherwise if numbers are to be tested for equality, = or **eql** should be used.

Numerical Data Types

Integers

The ***integer*** data type consists of fixnums and bignums.

The ***fixnum*** data type is designed to allow integers in the range from **most-negative-fixnum** to **most-positive-fixnum** to be represented efficiently, using a fixed number of bits. The fixnum data type is the default for the representation of integers.

The ***bignum*** data type is provided to allow for the representation of integers of arbitrary magnitude. The distinction between fixnums and bignums is generally not visible to the user. In Sun Common Lisp, the more appropriate representation is used automatically.

Ratios

Ratios give an exact representation of the mathematical quotient of two integers. Ratios can be used to avoid the loss of precision that can result from using floating-point numbers.

Rational numbers are represented in ***canonical form***. If the ratio is not an integer, the canonical representation is a pair of integers, the numerator and denominator, that represent the rational as a fraction in reduced form. The denominator is always positive. If the denominator evenly divides the numerator, the rational number is converted to the resulting integer.

Floating-Point Numbers

Floating-point numbers constitute the type ***float***. Common Lisp designates four floating-point number formats: ***short-float***, ***single-float***, ***double-float***, and ***long-float***. These formats differ in the precision they provide and in the range of exponents they allow. Sun Common Lisp represents all four types of floating-point numbers in the single-float format.

In Sun Common Lisp, single-float numbers are treated in accordance with the IEEE standard for the representation of 32-bit single-precision floating-point numbers. They are represented by a sign bit, a 23-bit unsigned mantissa, and an 8-bit unsigned exponent. The exponent is excess-127; that is, the representation of the exponent is an 8-bit integer whose value is 127 greater than the true exponent value. Floating-point numbers are represented in radix 2.

When an operation involves both a rational and a floating-point argument, the rational number is first converted to floating-point format, and then the operation is performed. This conversion process is called ***floating-point contagion***.

Complex Numbers

Complex numbers are represented as composite objects consisting of a real part and an imaginary part.

The two parts of a complex number must be of the same noncomplex type; if they are not, they are automatically converted to the same type, in accordance with the principle of floating-point contagion. Complex numbers are represented in ***canonical form***. If a complex number whose components are of type integer or ratio has an imaginary part whose value is zero, the canonical representation is an integer or ratio whose value is the same as that of the real part.

Bytes

Common Lisp provides functions that manipulate fields of bits that are contained within integers. Such a field of bits is called a ***byte***. Functions that manipulate bytes use ***byte specifiers*** to indicate a field within an integer according to its size and position. A byte specifier whose size and position attributes are ***size*** and ***position*** designates bits whose weights are $2^{position}$ through $2^{position+size-1}$.

Note that this usage of the term *byte* is distinct from its conventional meaning. Bytes in this chapter refer to the above meaning of the term.

Categories of Operations

This section groups operations on numbers according to functionality.

Data Type Predicates

numberp	**floatp**
complexp	**integerp**
fixnump	**rationalp**

These predicates determine whether an object is a number.

Predicates on Numbers

plusp	**evenp**
minusp	**oddp**
zerop	

These predicates test properties of numbers.

Numerical Comparisons

=	>
/=	>=
<	**max**
<=	**min**

These functions provide comparison operations on numbers.

Arithmetic Operations

+	**incf**
–	**decf**
*****	**gcd**
/	**lcm**
1+	**mod**
1–	**rem**
ash	**conjugate**

These functions perform arithmetic operations on numbers.

Irrational Functions

exp	**asin**
expt	**acos**
log	**atan**
sqrt	**sinh**
isqrt	**cosh**
abs	**tanh**
phase	**asinh**
signum	**acosh**
sin	**atanh**
cos	**cis**
tan	**pi**

These constructs provide exponential, trigonometric, and other transcendental operations.

Type Conversion Operations

rational	**float**
rationalize	**decode-float**
numerator	**scale-float**
denominator	**float-radix**
floor	**float-sign**
ceiling	**float-digits**
truncate	**float-precision**
round	**integer-decode-float**
ffloor	**complex**
fceiling	**realpart**
ftruncate	**imagpart**
fround	

These functions perform conversion operations among the numerical data types.

Logical Operations on Numbers

boole	**logand**
boole-clr	**logior**
boole-set	**logxor**
boole-1	**logeqv**
boole-2	**lognand**
boole-c1	**lognor**
boole-c2	**logandc1**
boole-and	**logandc2**
boole-ior	**logorc1**
boole-xor	**logorc2**
boole-eqv	**lognot**
boole-nand	**logtest**
boole-nor	**logbitp**
boole-andc1	**logcount**
boole-andc2	**integer-length**
boole-orc1	
boole-orc2	

These constructs provide logical operations on integers.

Byte Manipulation Functions

byte	**dpb**
byte-size	**ldb**
byte-position	**ldb-test**
deposit-field	**mask-field**

These functions manipulate fields of bits within integers.

Random Numbers

make-random-state	***random-state***
random	**random-state-p**

These constructs provide a random number generation facility.

Implementation-Dependent Constants

most-positive-fixnum	**least-negative-long-float**
most-negative-fixnum	**most-negative-short-float**
most-positive-short-float	**most-negative-single-float**
most-positive-single-float	**most-negative-double-float**
most-positive-double-float	**most-negative-long-float**
most-positive-long-float	**short-float-epsilon**
least-positive-short-float	**single-float-epsilon**
least-positive-single-float	**double-float-epsilon**
least-positive-double-float	**long-float-epsilon**
least-positive-long-float	**short-float-negative-epsilon**
least-negative-short-float	**single-float-negative-epsilon**
least-negative-single-float	**double-float-negative-epsilon**
least-negative-double-float	**long-float-negative-epsilon**

These constants may be useful in parameterizing code. Their values are implementation dependent.

*

Purpose: The function * returns the product of its arguments. If no arguments are given, it returns 1.

Syntax: * &rest *numbers* [*Function*]

Remarks: Any necessary type conversions are performed automatically.

Examples:

```
> (*)
1
> (* 3 5)
15
> (* 1.0 #c(22 33) 55/98)
#C(12.346938 18.520407)
```

+

Purpose: The function + returns the sum of its arguments. If no arguments are given, it returns 0.

Syntax: + **&rest** ***numbers*** [*Function*]

Remarks: Any necessary type conversions are performed automatically.

Examples:

```
> (+)
0
> (+ 1)
1
> (+ 31/100 69/100)
1
> (+ 1/5 0.8)
1.0
```

–

Purpose: The function – performs arithmetic subtraction and negation.

If – is given more than one argument, it subtracts from the first argument all of the following arguments and returns the result.

If it is given one argument, it returns the negative of that argument.

Syntax: – *number* `&rest` *more-numbers* [*Function*]

Remarks: Any necessary type conversions are performed automatically.

Examples:

```
> (- 55.55)
-55.55
> (- 0)
0
> (- #c(100 45) #c(0 45))
100
> (- 10 1 2 3 4)
0
```

/

Purpose: The function / performs division.

If / is given more than one argument, it divides the first argument by all of the following arguments and returns the result.

If it is given one argument, it returns the reciprocal of that argument.

Syntax: / *number* `&rest` *more-numbers* [*Function*]

Remarks: The function / results in a ratio if the arguments are all integers or ratios and the result is not an exact integer.

Any necessary type conversions are performed automatically.

Examples:

```
> (/ 0.5)
2.0
> (/ 20 5)
4
> (/ 5 20)
1/4
> (/ 60 -2 3 5.0)
-2.0
> (/ 2 #c(2 2))
#C(1/2 -1/2)
```

See Also: **floor**

ceiling

truncate

round

1+, 1−

Purpose: The functions **1+** and **1−** increment and decrement a number by 1 respectively.

Syntax: **1+** *number* [*Function*]

1− *number* [*Function*]

Examples:

```
> (1+ 99)
100
```

See Also: **incf**

decf

<, <=, >, >=

Purpose: The functions <, <=, >, and >= perform arithmetic comparisons on their arguments.

The function < is true if its numerical arguments are in monotonically increasing order; otherwise it is false.

The function <= is true if its numerical arguments are in monotonically nondecreasing order; otherwise it is false.

The function > is true if its numerical arguments are in monotonically decreasing order; otherwise it is false.

The function >= is true if its numerical arguments are in monotonically nonincreasing order; otherwise it is false.

Syntax:

< *number* `&rest` *more-numbers* [*Function*]

< = *number* `&rest` *more-numbers* [*Function*]

> *number* `&rest` *more-numbers* [*Function*]

> = *number* `&rest` *more-numbers* [*Function*]

Remarks: Any necessary type conversions are performed automatically.

The arguments of these functions must be noncomplex numbers.

Examples:

```
> (< 1)
T
> (<= 1 1.0 11/10 1.9 1.99)
T
> (> 1000000000000000000000000 9.9 1.0 1)
NIL
> (>= 1000000000000000000000000 9.9 1.0 1)
T
```

=, /=

Purpose: The predicates = and /= test for arithmetic equality.

The predicate = is true if all of its numerical arguments are the same in value; otherwise it is false.

The predicate /= is true if no two of its numerical arguments are the same in value; otherwise it is false.

Syntax: = *number* &rest *more-numbers* [*Function*]

/ = *number* &rest *more-numbers* [*Function*]

Remarks: The arguments of = and /= may be complex numbers.

Any necessary type conversions are performed automatically.

Examples:

```
> (= 1)
T
> (= 1 5/5 1.0 #c(1 0) #c(1.0 0.0))
T
> (= 1 2)
NIL
> (= 0.0 (- 0.0))
T
> (/= 1)
T
> (/= 1 1.0 2)
NIL
> (/= 1 2 3 4)
T
```

abs

Purpose: The function **abs** returns the absolute value of its numerical argument.

Syntax: **abs** *number* [*Function*]

Remarks: If *number* is noncomplex, the result is of the same type as the argument.

If *number* is complex, the result is equivalent to the following:

```
(sqrt (+ (expt (realpart number) 2) (expt (imagpart number) 2)))
```

Examples:

```
> (abs 0)
0
> (abs 12/13)
12/13
> (abs -1.09)
1.09
> (abs #c(3 -4))
5.0
```

ash

Purpose: The function **ash** performs the arithmetic shift operation on the binary representation of its *integer* argument and returns the result.

The *count* argument is an integer. If *count* is positive, the *integer* argument is shifted left *count* positions. If *count* is negative, the *integer* argument is shifted right −*count* positions.

Syntax: **ash** *integer count* [*Function*]

Remarks: The sign of the result is the same as the sign of the *integer* argument.

Examples:

```
> (ash 16 1)
32
> (ash 16 0)
16
> (ash 16 -1)
8
> (ash -100000000000000000000000000000000 -100)
-79
```

asin, acos, atan

Purpose: The functions **asin**, **acos**, and **atan** compute the arc sine, arc cosine, and arc tangent respectively. The results are given in radians.

Syntax:

asin *number* [*Function*]

acos *number* [*Function*]

atan *number1* `&optional` *number2* [*Function*]

Remarks: The functions **asin** and **acos** may have complex arguments. The function **atan** may have a complex argument if only one argument is specified.

If the optional argument *number2* of **atan** is specified, both it and the *number1* argument must be noncomplex numbers. In this case, the result is the arc tangent of *number1*/*number2*.

Examples:

```
> (asin 0)
0.0
> (/ (realpart (acos #c(0 1))) (/ pi 2))
1.0
> (/ (atan 1 (sqrt 3)) (/ pi 6))
1.0000001
```

boole, boole-clr, boole-set, boole-1, boole-2, boole-c1, boole-c2, boole-and, boole-ior, boole-xor, boole-eqv, boole-nand, boole-nor, boole-andc1, boole-andc2, boole-orc1, boole-orc2

Purpose: The function **boole** is used to perform bit-wise logical operations on integers. The operation to be performed is specified by *op*, which must be one of the constants listed below. The result of the operation is returned as an integer.

The **boole-clr** operation always returns the value 0.

The **boole-set** operation always returns the value 1.

The **boole-1** operation returns the value of its first operand.

The **boole-2** operation returns the value of its second operand.

The **boole-c1** operation returns the *logical complement* of the value of its first operand.

The **boole-c2** operation returns the *logical complement* of the value of its second operand.

The **boole-and** operation returns the *logical and* of its operands.

The **boole-ior** operation returns the *logical inclusive or* of its operands.

The **boole-xor** operation returns the *logical exclusive or* of its operands.

The **boole-eqv** operation returns the *logical equivalence* of its operands.

The **boole-nand** operation returns the *logical complement* of the *logical and* of its operands.

The **boole-nor** operation returns the *logical complement* of the *logical inclusive or* of its operands.

The **boole-andc1** operation returns the result of performing the *logical and* operation on the second operand and the *logical complement* of the first operand.

The **boole-andc2** operation returns the result of performing the *logical and* operation on the first operand and the *logical complement* of the second operand.

The **boole-orc1** operation returns the result of performing the *logical inclusive or* operation on the second operand and the *logical complement* of the first operand.

The **boole-orc2** operation returns the result of performing the *logical inclusive or* operation on the first operand and the *logical complement* of the second operand.

boole, boole-clr, boole-set, boole-1, boole-2, boole-c1, boole-c2, ...

Syntax:

boole *op integer1 integer2*	[*Function*]
boole-clr	[*Constant*]
boole-set	[*Constant*]
boole-1	[*Constant*]
boole-2	[*Constant*]
boole-c1	[*Constant*]
boole-c2	[*Constant*]
boole-and	[*Constant*]
boole-ior	[*Constant*]
boole-xor	[*Constant*]
boole-eqv	[*Constant*]
boole-nand	[*Constant*]
boole-nor	[*Constant*]
boole-andc1	[*Constant*]
boole-andc2	[*Constant*]
boole-orc1	[*Constant*]
boole-orc2	[*Constant*]

Remarks: Negative integers are treated as if they were in two's complement representation.

Examples:

```
> (boole boole-ior 1 16)
17
> (boole boole-and -2 5)
4
> (boole boole-eqv 17 15)
-31
```

See Also:

- **logand**
- **logandc1**
- **logandc2**
- **logeqv**
- **logior**
- **lognand**
- **lognor**
- **logorc1**
- **logorc2**
- **logxor**

byte, byte-size, byte-position

Purpose: The function **byte** constructs byte specifiers; the functions **byte-size** and **byte-position** return the attributes of byte specifiers.

The function **byte** constructs and returns a byte specifier for use by byte manipulation functions. The resulting byte specifier indicates a byte consisting of *size* bits whose weights are $2^{position+size-1}$ through $2^{position}$.

The function **byte-size** returns the number of bits in the specified byte as an integer value.

The function **byte-position** returns the position of the specified byte as a positive integer.

Syntax:

byte *size position* [*Function*]

byte-size *bytespec* [*Function*]

byte-position *bytespec* [*Function*]

Remarks: The *size* and *position* arguments must be integers. Their values must be in the range from 1 to 4095 inclusive.

Examples:

```
> (setq b (byte 100 200))
#.(BYTE 100. 200.)
> (byte-size b)
100
> (byte-position b)
200
```

cis

Purpose: The function **cis** returns the value of $e^{i \cdot radians}$. The result is a complex number in which the real part is equal to the cosine of the *radians* argument, and the imaginary part is equal to the sine of the *radians* argument.

Syntax: **cis** ***radians*** [*Function*]

Remarks: The ***radians*** argument must be a noncomplex number.

Examples:

```
> (cis 0)
#C(1.0 0.0)
```

complex

Purpose: The function **complex** returns a complex number whose real and imaginary parts have the specified values.

If the *imagpart* argument is not specified, the imaginary part is a zero of the same type as the real part.

Syntax: **complex** *realpart* `&optional` *imagpart* [*Function*]

Remarks: The *realpart* and *imagpart* arguments must be noncomplex numbers. If one of these arguments is a floating-point number, the rules of floating-point contagion apply.

If *realpart* is a rational number and *imagpart* is zero, the result of **complex** is a rational number.

Examples:

```
> (complex 0)
0
> (complex 0.0)
#C(0.0 0.0)
> (complex 1 1/2)
#C(1 1/2)
> (complex 1 .99)
#C(1.0 .99)
> (complex 3/2 0.0)
#C(1.5 0.0)
```

complexp

Purpose: The predicate **complexp** is true if its argument is a complex number; otherwise it is false.

Syntax: **complexp** *object* [*Function*]

Examples:

```
> (complexp 1.2d2)
NIL
> (complexp #c(5/3 7.2))
T
```

conjugate

Purpose: The function **conjugate** returns the complex conjugate of its numerical argument.

Syntax: **conjugate** *number* [*Function*]

Examples:

```
> (conjugate #c(0 -1))
#C(0 1)
> (conjugate #c(1 1))
#C(1 -1)
> (conjugate 1.5)
1.5
```

decode-float, integer-decode-float

Purpose: The function **decode-float** returns three values that characterize its floating-point argument.

The first result is the value of the mantissa of *float*, scaled so that it is greater than or equal to 1/*radix* and less than 1, where *radix* is the radix of the floating-point representation of *float*.

The second result is the integer exponent to which the radix must be raised to obtain the value that, when multiplied with the first result, produces the absolute value of the original *float* value.

The third result is a floating-point number of the same type as *float*. Its value is 1.0 if *float* is greater than or equal to zero; otherwise its value is -1.0.

The function **integer-decode-float** is similar to **decode-float**. It returns three values. The first value is the result of scaling the mantissa of *float* so that it is an integer. The second result is the integer exponent to which the radix must be raised to obtain the value that, when multiplied with the first result, produces the absolute value of the original *float* value. The third result is 1 if *float* is greater than or equal to zero; otherwise it is -1.

Syntax:

decode-float *float* [*Function*]

integer-decode-float *float* [*Function*]

Remarks: The product of the first result of **decode-float** or **integer-decode-float**, of the radix raised to the power of the second result, and of the third result is exactly equal to the value of the argument.

In Sun Common Lisp, all floating-point numbers are represented in single-float format.

Examples:

```
> (decode-float 0)
0.0
0
1.0
> (decode-float .5)
.5
0
1.0
> (decode-float 1.0)
.5
1
1.0
```

```
> (integer-decode-float 1)
8388608
-23
1
> (* 8388608 (expt 2 -23) 1)
1
```

deposit-field

Purpose: The function **deposit-field** is used to replace a field of bits within an integer. It returns a copy of its *integer* argument in which the bits of the specified byte have been replaced by the bits from the corresponding positions of the integer *newbyte*.

Syntax: **deposit-field** *newbyte bytespec integer* [*Function*]

Remarks: The *bytespec* argument is a byte specifier.

Examples:

```
> (deposit-field 7 (byte 2 1) 0)
6
> (deposit-field -1 (byte 4 0) 0)
15
> (deposit-field 0 (byte 2 1) -3)
-7
```

See Also: **byte**

dpb

dpb

Purpose: The deposit byte function **dpb** is used to replace a field of bits within an integer. It returns a copy of its *integer* argument in which the bits of the specified byte have been replaced by the corresponding number of low-order bits from the integer *newbyte*.

Syntax: **dpb** *newbyte bytespec integer* [*Function*]

Remarks: The *bytespec* argument is a byte specifier.

Examples:

```
> (dpb 1 (byte 1 10) 0)
1024
> (dpb -2 (byte 2 10) 0)
2048
> (dpb 1 (byte 2 10) 2048)
1024
```

See Also: **byte**

deposit-field

evenp, oddp

Purpose: The predicate **evenp** is true if its *integer* argument is even; otherwise it is false.

The predicate **oddp** is true if its *integer* argument is odd; otherwise it is false.

Syntax: **evenp** *integer* [*Function*]

oddp *integer* [*Function*]

Examples:

```
> (evenp 0)
T
> (oddp 10000000000000000000000)
NIL
> (oddp -1)
T
```

exp, expt

Purpose: The functions **exp** and **expt** are used for exponentiation.

The function **exp** returns **e**, the base of the natural logarithm function, raised to the power *number*.

The function **expt** returns the argument *base-number* raised to the power *power-number*.

Syntax: **exp** *number* [*Function*]

expt *base-number power-number* [*Function*]

Examples:

```
> (exp 0)
1.0
> (exp 1)
2.7182817
> (exp (log 10))
10.0
> (expt 2 8)
256
> (expt 4 .5)
2.0
> (expt #c(0 1) 2)
-1
```

See Also: **log**

fixnump

Purpose: The predicate **fixnump** is true if its argument is a fixnum; otherwise it is false.

Syntax: **fixnump** *object* [*Function*]

Remarks: The predicate **fixnump** is an extension to Common Lisp.

Examples:

```
> (fixnump 1)
T
> (fixnump 12)
T
> (fixnump (expt 2 130))
NIL
> (fixnump 6/5)
NIL
```

See Also: **most-positive-fixnum**

most-negative-fixnum

float

Purpose: The function **float** converts a noncomplex number to a floating-point number.

If the optional argument is not specified and the number is already in a floating-point format, then it is simply returned; otherwise the number is converted to single-float format.

An optional floating-point argument may be given. In this case, ***number*** is converted to the same floating-point format as *float.*

Syntax: **float** ***number*** `&optional` *float* [*Function*]

Remarks: In Sun Common Lisp, all floating-point numbers are represented in single-float format.

Examples:

```
> (float 0)
0.0
> (float 1 .5)
1.0
> (float 1.0)
1.0
```

See Also: **coerce**

float-digits, float-precision, float-radix

Purpose: The functions **float-digits**, **float-precision**, and **float-radix** are used to examine the attributes of floating-point numbers.

The function **float-digits** returns the number of digits available in the radix of the floating-point representation for representing the mantissa of a floating-point number of the same type as the *float* argument. This result includes the hidden bit that is used in the normalization of floating-point numbers.

The function **float-precision** returns the number of significant digits in the radix of the floating-point representation of its *float* argument. If *float* is 0, the result of **float-precision** is 0.

The function **float-radix** returns the radix of the floating-point representation of its *float* argument.

Syntax:

float-digits *float* [*Function*]

float-precision *float* [*Function*]

float-radix *float* [*Function*]

Remarks: For normalized floating-point numbers, the results of **float-digits** and **float-precision** are the same.

In Sun Common Lisp, all floating-point numbers are represented in single-float format.

Examples:

```
> (float-radix 1.0)
2
> (float-precision 1.0)
24
> (float-digits 1.0)
24
> (float-precision least-positive-single-float)
1
```

float-sign

Purpose: The function **float-sign** returns a floating-point number whose sign is the same as that of *float1* and whose absolute value is the same as that of *float2*. If the *float2* argument is not specified, a floating-point 1.0 of the appropriate sign is returned.

Syntax: **float-sign** *float1* `&optional` *float2* [*Function*]

Examples:

```
> (float-sign 5.0)
1.0
> (float-sign -5.0)
-1.0
> (float-sign 0.0)
1.0
> (float-sign 1.0 0.0)
0.0
> (float-sign 1.0 -10.0)
10.0
> (float-sign -1.0 10.0)
-10.0
```

floatp

Purpose: The predicate **floatp** is true if its argument is a floating-point number; otherwise it is false.

Syntax: **floatp** *object* [*Function*]

Examples:

```
> (floatp 1.2d2)
T
> (floatp 1.212)
T
> (floatp 1.2s2)
T
> (floatp (expt 2 130))
NIL
```

floor, ceiling, ffloor, fceiling

Purpose: The functions **floor**, **ceiling**, **ffloor**, and **fceiling** perform type conversion operations on noncomplex numbers.

The functions **floor** and **ceiling** convert their arguments to integers. The function **floor** returns the largest integer that is equal to or less than its argument. The function **ceiling** returns the smallest integer that is equal to or greater than its argument.

The functions **ffloor** and **fceiling** produce results identical to those of **floor** and **ceiling** except that the results are returned in floating-point format.

If the optional argument *divisor* is given, then the first value of the result is mathematically equivalent to dividing *number* by *divisor* and then applying the floor or ceiling operation. The *divisor* must also be a noncomplex number.

Syntax:

floor *number* `&optional` *divisor* [*Function*]

ceiling *number* `&optional` *divisor* [*Function*]

ffloor *number* `&optional` *divisor* [*Function*]

fceiling *number* `&optional` *divisor* [*Function*]

Remarks: Each of these functions returns the remainder of the result as a second value. If the arguments of **floor** and **ceiling** are of the same type, the remainder is of that type also; otherwise it is a floating-point number.

Examples:

```
> (floor 3/2)
1
1/2
> (ceiling 3 2)
2
-1
> (ffloor 3 2)
1.0
1.0
> (fceiling 3/2)
2.0
-.5
```

See Also: **truncate**

round

gcd

Purpose: The function **gcd** returns the greatest common divisor of its arguments. If no arguments are specified, **gcd** returns 0.

Syntax: **gcd** **&rest** *integers* [*Function*]

Remarks: The result of **gcd** is always nonnegative.

Examples:

```
> (gcd)
0
> (gcd 60 42)
6
> (gcd 3333 -33 101)
1
> (gcd 3333 -33 1002001)
11
```

incf, decf

Purpose: The macros **incf** and **decf** are used for incrementing and decrementing the value of a variable.

The macro **incf** adds the number specified by *delta* to the number stored in *place* and returns the result.

The macro **decf** subtracts the number specified by *delta* from the number stored in *place* and returns the result.

Syntax:

incf *place* [*delta*] [*Macro*]

decf *place* [*delta*] [*Macro*]

Remarks: The *place* argument must be a generalized variable acceptable to the macro **setf**.

If *delta* is not specified, the number in *place* is incremented or decremented by 1.

Any necessary type conversions are performed automatically.

Examples:

```
> (setq a '(1))
(1)
> (incf (car a))
2
> a
(2)
> (decf (car a) 2)
0
> a
(0)
```

See Also: **1+**

1−

integer-length

Purpose: The function **integer-length** determines how many bits are needed to represent a given integer.

If the integer is nonnegative, it can be represented as an unsigned binary number in `(integer-length` *integer*`)` bits. Any integer can be represented in signed two's complement representation in `(1+ (integer-length` *integer*`))` bits.

Syntax: **integer-length** *integer* [*Function*]

Remarks: The value returned by **integer-length** is equal to the following:

`(ceiling (log (if (minusp` *integer*`) (-` *integer*`) (1+` *integer*`)) 2))`

Examples:

```
> (integer-length 0)
0
> (integer-length 1)
1
> (integer-length -1)
0
> (integer-length (expt 2 9))
10
> (integer-length (1- (expt 2 9)))
9
> (integer-length (- (expt 2 9)))
9
> (integer-length (- (1+ (expt 2 9))))
10
```

integerp

Purpose: The predicate **integerp** is true if its argument is an integer; otherwise it is false.

Syntax: **integerp** *object* [*Function*]

Examples:

```
> (integerp 1)
T
> (integerp (expt 2 130))
T
> (integerp 6/5)
NIL
> (integerp nil)
NIL
```

lcm

Purpose: The function **lcm** returns the least common multiple of its integer arguments. The result is a nonnegative integer.

Syntax: **lcm** *integer* `&rest` *more-integers* [*Function*]

Examples:

```
> (lcm 10)
10
> (lcm 25 30)
150
> (lcm -24 18 10)
360
```

ldb

Purpose: The load byte function **ldb** extracts from its *integer* argument the bits of the specified byte and returns the result as a nonnegative integer.

Syntax: **ldb** *bytespec integer* [*Function*]

Remarks: The *bytespec* argument is a byte specifier.

If the *integer* argument is a generalized variable that is acceptable to the macro **setf**, then **setf** may be used with **ldb** to modify the specified byte.

Examples:

```
> (ldb (byte 2 1) 10)
1
> (setq a '(8))
(8)
> (setf (ldb (byte 2 1) (car a)) 1)
1
> a
(10)
```

See Also: **byte**

dpb

ldb-test

Purpose: The predicate **ldb-test** is true if any of the bits of the specified byte from *integer* are nonzero; otherwise it is false.

Syntax: **ldb-test** *bytespec integer* [*Function*]

Remarks: The *bytespec* argument is a byte specifier.

Examples:

```
> (ldb-test (byte 4 1) 16)
T
> (ldb-test (byte 3 1) 16)
NIL
> (ldb-test (byte 3 2) 16)
T
```

log

Purpose: The function **log** returns the logarithm of its *number* argument in the base *base*. If no base is specified, **e**, the base of the natural logarithms, is used.

Syntax: **log** *number* `&optional` *base* [*Function*]

Remarks: If the *number* argument is negative, **log** always produces a complex result.

Examples:

```
> (log (exp 3))
3.0
> (log 100 10)
2.0
> (log #c(0 1) #c(0 -1))
#C(-1.0 0.0)
```

See Also: **exp**

expt

logand, logandc1, logandc2, logeqv, logior, lognand, lognor, logorc1, logorc2, logxor

Purpose: The functions **logand**, **logandc1**, **logandc2**, **logeqv**, **logior**, **lognand**, **lognor**, **logorc1**, **logorc2**, and **logxor** perform bit-wise logical operations on their integer arguments.

The function **logand** returns the *logical and* of its integer arguments. If no arguments are given, it returns −1.

The function **logandc1** returns the *logical and* of its first argument with the *logical complement* of its second argument.

The function **logandc2** returns the *logical and* of its second argument with the *logical complement* of its first argument.

The function **logeqv** returns the *logical equivalence* of its integer arguments. If no arguments are given, it returns −1.

The function **lognand** performs the *logical and* operation on its integer arguments and returns the *logical complement* of the result.

The function **lognor** performs the *logical inclusive or* operation on its integer arguments and returns the *logical complement* of the result.

The function **logior** returns the *logical inclusive or* of its integer arguments. If no arguments are given, it returns 0.

The function **logorc1** returns the *logical inclusive or* of its first argument with the *logical complement* of its second argument.

The function **logorc2** returns the *logical inclusive or* of its second argument with the *logical complement* of its first argument.

The function **logxor** returns the *logical exclusive or* of its integer arguments. If no arguments are given, it returns 0.

Syntax:

logand `&rest` *integers* [*Function*]

logandc1 *integer1 integer2* [*Function*]

logandc2 *integer1 integer2* [*Function*]

logeqv `&rest` *integers* [*Function*]

logior `&rest` *integers* [*Function*]

lognand *integer1 integer2* [*Function*]

lognor *integer1 integer2* [*Function*]

logorc1 *integer1 integer2* [*Function*]

logorc2 *integer1 integer2* [*Function*]

logxor &rest *integers* [*Function*]

Remarks: Negative integers are treated as if they were in two's complement representation.

Examples:

```
> (logior 1 2 4 8)
15
> (logxor 1 3 7 15)
10
> (logeqv)
-1
> (logand 16 31)
16
```

See Also: **lognot**

boole

logbitp

Purpose: The predicate **logbitp** is used to test the value of a particular bit in an integer. The predicate **logbitp** is true if the value of the bit in *integer* whose weight is 2^{index} is 1; otherwise it is false.

Syntax: **logbitp** *index integer* [*Function*]

Remarks: Negative integers are treated as if they were in two's complement representation.

Examples:

```
> (logbitp 1 1)
NIL
> (logbitp 0 1)
T
> (logbitp 3 10)
T
> (logbitp 1000000000000000000000000000 -1)
T
```

logcount

Purpose: The function **logcount** counts the values of individual bits of integers.

If the *integer* argument is positive, it returns the number of bits that have the value 1.

If *integer* is negative, it returns the number of bits that have the value 0 in the two's complement representation of the negative value.

Syntax: **logcount** *integer* [*Function*]

Examples:

```
> (logcount 0)
0
> (logcount -1)
0
> (logcount 7)
3
> (logcount -15)
3
> (logcount (expt 2 100))
1
```

lognot

Purpose: The function **lognot** complements all the bits in its integer argument and returns the result.

Syntax: **lognot** *integer* [*Function*]

Remarks: Negative integers are treated as if they were in two's complement representation.

Examples:

```
> (lognot 0)
-1
> (lognot 1)
-2
> (lognot -1)
0
> (lognot (1+ (lognot 1000)))
999
```

logtest

Purpose: The predicate **logtest** is true if any bit that has the value 1 in *integer1* also has the value 1 in *integer2* and if *integer1* is not 0; otherwise it is false.

Syntax: **logtest** *integer1 integer2* [*Function*]

Remarks: Negative integers are treated as if they were in two's complement representation.

Examples:

```
> (logtest 1 7)
T
> (logtest 1 2)
NIL
> (logtest -2 -1)
T
> (logtest 0 -1)
NIL
```

make-random-state

Purpose: The function **make-random-state** creates and returns an object of type random state.

If the optional *state* argument is not specified or is **nil**, **make-random-state** returns a copy of the current value of the variable ***random-state***. If *state* is **t**, **make-random-state** creates a new state object and initializes it in a random way. If the *state* specifies a state object, **make-random-state** returns a copy of that object.

Syntax: **make-random-state** `&optional` *state* [*Function*]

Examples:

```
> (random-state-p (setq randy (make-random-state)))
T
> (= (setq r1 (random 1000)) (random 1000))
NIL
> (progn (setq randy-clone (make-random-state randy)) nil)
NIL
> (= r1 (random 1000 randy))
T
> (= r1 (random 1000 randy))
NIL
> (= r1 (random 1000 randy-clone))
T
```

See Also: **random**

mask-field

Purpose: The function **mask-field** performs a mask operation on its integer argument. It returns an integer that agrees with the *integer* argument on the bits of the specified byte and that contains zero-bits elsewhere.

Syntax: **mask-field** *bytespec integer* [*Function*]

Remarks: The *bytespec* argument is a byte specifier.

If *integer* is a generalized variable that is acceptable to the macro **setf**, then **setf** may be used with **mask-field** to modify the specified byte.

Examples:

```
> (mask-field (byte 1 5) -1)
32
> (setq a 15)
15
> (mask-field (byte 2 0) a)
3
> a
15
> (setf (mask-field (byte 2 0) a) 1)
1
> a
13
```

See Also: **byte**

max, min

Purpose: The function **max** returns the largest of its numerical arguments.

The function **min** returns the smallest of its numerical arguments.

Syntax: **max** *number* **&rest** *more-numbers* [*Function*]

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

Remarks: The arguments must be noncomplex numbers.

Examples:

```
> (max 1)
1
> (min 1 -1.0)
-1.0
> (max 1 2.0 5/2 4)
4
```

minusp, plusp

Purpose: The predicate **minusp** is true if its numerical argument is strictly less than zero; otherwise it is false.

The predicate **plusp** is true if its numerical argument is strictly greater than zero; otherwise it is false.

Syntax: **minusp** *number* [*Function*]

plusp *number* [*Function*]

Remarks: The *number* argument must be a noncomplex number.

Examples:

```
> (minusp -1)
T
> (plusp 0)
NIL
> (plusp least-positive-single-float)
T
```

mod, rem

Purpose: The functions **mod** and **rem** are generalizations of the modulus and remainder functions respectively.

The function **mod** performs the **floor** operation on its arguments and returns the remainder as its result.

The function **rem** performs the **truncate** operation on its arguments and returns the remainder as its result.

Syntax: **mod** *number divisor* [*Function*]

rem *number divisor* [*Function*]

Remarks: Both arguments must be noncomplex numbers.

Examples:

```
> (mod 17 4)
1
> (rem -1 5)
-1
> (mod -1 5)
4
```

See Also: **floor**

truncate

most-positive-fixnum, most-negative-fixnum

Purpose: The constants **most-positive-fixnum** and **most-negative-fixnum** define the implementation-dependent limits on the values of fixnums.

The value of **most-positive-fixnum** is the positive fixnum of the largest magnitude provided by the implementation. The value of **most-positive-fixnum** in Sun Common Lisp is $2^{29} - 1$.

The value of **most-negative-fixnum** is the negative fixnum of the largest magnitude provided by the implementation. The value of **most-negative-fixnum** in Sun Common Lisp is -2^{29}.

Syntax:

most-positive-fixnum *[Constant]*

most-negative-fixnum *[Constant]*

Examples:

```
> most-positive-fixnum
536870911
> most-negative-fixnum
-536870912
> (expt 2 29)
536870912
```

most-positive-short-float, most-positive-single-float, most-positive-double-float, most-positive-long-float, least-positive-short-float, least-positive-single-float, least-positive-double-float, least-positive-long-float, least-negative-short-float, least-negative-single-float, least-negative-double-float, least-negative-long-float, most-negative-short-float, most-negative-single-float, most-negative-double-float, most-negative-long-float

Purpose: These constants define the implementation-dependent limits on the values of floating-point numbers.

The constants **most-positive-short-float**, **most-positive-single-float**, **most-positive-double-float**, and **most-positive-long-float** designate the positive floating-point number of the largest magnitude of the given format.

The constants **least-positive-short-float**, **least-positive-single-float**, **least-positive-double-float**, and **least-positive-long-float** designate the smallest positive (nonzero) floating-point number of the given format.

The constants **least-negative-short-float**, **least-negative-single-float**, **least-negative-double-float**, and **least-negative-long-float** designate the negative (nonzero) floating-point number of the smallest magnitude of the given format.

The constants **most-negative-short-float**, **most-negative-single-float**, **most-negative-double-float**, and **most-negative-long-float** designate the negative floating-point number of the largest magnitude of the given format.

Syntax:

most-positive-short-float [*Constant*]

most-positive-single-float [*Constant*]

most-positive-double-float [*Constant*]

most-positive-long-float [*Constant*]

least-positive-short-float [*Constant*]

least-positive-single-float [*Constant*]

least-positive-double-float [*Constant*]

least-positive-long-float [*Constant*]

least-negative-short-float [*Constant*]

least-negative-single-float [*Constant*]

least-negative-double-float [*Constant*]

least-negative-long-float [*Constant*]

most-negative-short-float [*Constant*]

most-negative-single-float [*Constant*]

most-negative-double-float [*Constant*]

most-negative-long-float [*Constant*]

Remarks: In Sun Common Lisp, all floating-point numbers are represented in single-float format.

Examples:

```
> (zerop least-negative-double-float)
NIL
> (zerop (/ least-negative-double-float 2))
T
```

numberp

Purpose: The predicate **numberp** is true if its argument is a number; otherwise it is false.

Syntax: **numberp** *object* [*Function*]

Examples:

```
> (numberp 12)
T
> (numberp (expt 2 130))
T
> (numberp #c(5/3 7.2))
T
> (numberp nil)
NIL
> (numberp (cons 1 2))
NIL
```

numerator, denominator

Purpose: The functions **numerator** and **denominator** operate on rational numbers. They reduce the rational number to its canonical form and then return the numerator or denominator respectively.

Syntax: **numerator** *rational* [*Function*]

denominator *rational* [*Function*]

Remarks: The denominator of a reduced rational number is a positive integer.

Examples:

```
> (numerator 1/2)
1
> (denominator 12/36)
3
> (numerator -1)
-1
> (denominator (/ -33))
33
```

phase

Purpose: The function **phase** computes the phase of a number. The phase is the angle between the vector representing the complex number and the positive real axis.

The result is in radians and is greater than $-\pi$ and less than or equal to π.

Syntax: **phase** *number* [*Function*]

Remarks: The phase of a complex number is equivalent to the following:

`(atan (imagpart number) (realpart number))`

Examples:

```
> (phase 1)
0.0
> (phase -1)
3.1415925
> (phase 0)
0.0
> (phase #c(0 1))
1.5707963
```

pi

Purpose: The constant **pi** is a long floating-point number that approximates the value of the constant π.

Syntax: **pi** [*Constant*]

Remarks: In Sun Common Lisp, all floating-point numbers are represented in single-float format.

Examples:

```
> pi
3.1415925
> (cos pi)
-1.0
```

random

Purpose: The function **random** is used to generate a random number. It returns a number that is of the same type as its *number* argument and whose value is greater than or equal to 0 and less than *number*.

An object of type random state may be specified as the *state* argument. This argument encodes the internal state maintained by the random number generator. If the *state* argument is not specified, the value of the ***random-state*** variable is used. The *state* argument is modified as a side-effect of the call to **random**.

Syntax: **random** *number* `&optional` *state* [*Function*]

Remarks: The *number* argument must be a positive integer or a positive floating-point number.

Examples:

```
> (<= 0 (random 1000) 1000)
T
> (progn
    (setq rstate1 (make-random-state)
          rstate2 (make-random-state))
    nil)
NIL
> (= (random 1000 rstate1) (random 1000 rstate2))
T
```

See Also: **make-random-state**

random-state

random-state

Purpose: The value of the ***random-state*** variable is an object of type random state. It is used to encode the internal state maintained by the random number generator.

Syntax: ***random-state*** [*Variable*]

Examples:

```
> (random-state-p *random-state*)
T
> (random-state-p (setq snap-shot (make-random-state)))
T
> (equalp snap-shot *random-state*)
T
> (random 100)  ;while this number is random, the next one will be the same
50              ;because a snapshot of the random state object is used
> (equalp snap-shot *random-state*)
NIL
> (random 100 snap-shot)
50
> (equalp snap-shot *random-state*)
T
```

See Also: **random**

make-random-state

random-state-p

Purpose: The predicate **random-state-p** is true if its argument is a random state object; otherwise it is false.

Syntax: **random-state-p** *object* [*Function*]

Examples:

```
> (random-state-p *random-state*)
T
> (random-state-p (make-random-state))
T
> (random-state-p 'foo)
NIL
```

See Also: make-random-state

rational, rationalize

Purpose: The functions **rational** and **rationalize** convert noncomplex numbers to rational numbers.

If the *number* argument is a rational number, both **rational** and **rationalize** simply return that number.

If the argument is a floating-point number, **rational** returns a rational number that is exactly equal in value to the floating-point number. The function **rationalize** returns a rational number that approximates the floating-point number to the accuracy of the underlying floating-point representation.

Syntax:

rational *number* [*Function*]

rationalize *number* [*Function*]

Examples:

```
> (rational 0)
0
> (rationalize -11/100)
-11/100
> (rationalize .1)
1/10
```

rationalp

Purpose: The predicate **rationalp** is true if its argument is a rational number; otherwise it is false.

Syntax: **rationalp** *object* [*Function*]

Remarks: Both ratios and integers are rational numbers.

Examples:

```
> (rationalp 12)
T
> (rationalp 6/5)
T
> (rationalp 1.212)
NIL
```

realpart, imagpart

Purpose: The functions **realpart** and **imagpart** return the real and imaginary parts of a complex number respectively.

Syntax: **realpart** *number* [*Function*]

imagpart *number* [*Function*]

Remarks: If the *number* argument is noncomplex, the imaginary part is returned as a zero of the same type as the real part.

Examples:

```
> (realpart #c(23 41))
23
> (imagpart #c(23 41.0))
41.0
> (realpart #c(23 41.0))
23.0
> (imagpart 23.0)
0.0
```

scale-float

Purpose: The function **scale-float** scales its *float* argument.

It returns the value of `(*` *float* `(expt (float` *radix float*`)` *integer*`))`, where *radix* is the radix of the floating-point representation of *float*.

Syntax: **scale-float** *float integer* [*Function*]

Examples:

```
> (scale-float 1.0 1)
2.0
> (scale-float 10.01 -2)
2.5025
> (scale-float 23 0)
23.0
```

short-float-epsilon, single-float-epsilon, double-float-epsilon, long-float-epsilon, short-float-negative-epsilon, single-float-negative-epsilon, double-float-negative-epsilon, long-float-negative-epsilon

Purpose: The values of these constants are implementation dependent.

The value of each of the constants **short-float-epsilon**, **single-float-epsilon**, **double-float-epsilon**, and **long-float-epsilon** is the smallest positive floating-point number ϵ of the given format, such that the following expression is true when evaluated:

```
(not (= (float 1 ϵ) (+ (float 1 ϵ) ϵ)))
```

The value of each of the constants **short-float-negative-epsilon**, **single-float-negative-epsilon**, **double-float-negative-epsilon**, and **long-float-negative-epsilon** is the smallest positive floating-point number ϵ of the given format, such that the following expression is true when evaluated:

```
(not (= (float 1 ϵ) (- (float 1 ϵ) ϵ)))
```

Syntax:

short-float-epsilon [*Constant*]

single-float-epsilon [*Constant*]

double-float-epsilon [*Constant*]

long-float-epsilon [*Constant*]

short-float-negative-epsilon [*Constant*]

single-float-negative-epsilon [*Constant*]

double-float-negative-epsilon [*Constant*]

long-float-negative-epsilon [*Constant*]

Remarks: In Sun Common Lisp, all floating-point numbers are represented in single-float format.

Examples:

```
> (= 1.0 (+ 1.0 single-float-epsilon))
NIL
> (= 1.0 (+ 1.0 (/ single-float-epsilon 2)))
T
> (minusp long-float-negative-epsilon)
NIL
```

signum

Purpose: The function **signum** returns a numerical value that indicates whether its argument is negative, zero, or positive.

If the *number* argument is a rational number, **signum** returns -1 if *number* is negative, 0 if it is zero, and 1 if it is positive.

If *number* is a floating-point number, results equivalent to these are returned in the same floating-point format as *number*.

If *number* is a complex zero, the result is the same as the argument. Otherwise if *number* is any other complex number, the phase of the result is the same as that of the argument, and its magnitude is 1.

Syntax: **signum** *number* [*Function*]

Examples:

```
> (signum 99)
1
> (signum -99/100)
-1
> (signum 0.0)
0.0
> (signum #c(0 33))
#C(0.0 1.0)
```

sin, cos, tan

Purpose: The functions **sin**, **cos**, and **tan** compute trigonometric functions. They compute the sine, cosine, and tangent functions respectively.

Syntax:

sin *radians* [*Function*]

cos *radians* [*Function*]

tan *radians* [*Function*]

Remarks: The *radians* argument may be a complex number.

Examples:

```
> (sin 0)
0.0
> (cos pi)
-1.0
> (tan #c(0 1))
#C(0.0 .7615941)
```

sinh, cosh, tanh, asinh, acosh, atanh

Purpose: The functions **sinh**, **cosh**, **tanh**, **asinh**, **acosh**, and **atanh** compute hyperbolic trigonometric functions.

The function **sinh** computes the hyperbolic sine, **cosh** the hyberbolic cosine, and **tanh** the hyperbolic tangent; **asinh** computes the hyperbolic arc sine, **acosh** the hyperbolic arc cosine, and **atanh** the hyperbolic arc tangent.

Syntax:

sinh *number* [*Function*]

cosh *number* [*Function*]

tanh *number* [*Function*]

asinh *number* [*Function*]

acosh *number* [*Function*]

atanh *number* [*Function*]

Remarks: The *number* argument may be complex.

Examples:

```
> (sinh 0)
0.0
> (cosh (complex 0 pi))
#C(-1.0 0.0)
```

sqrt, isqrt

Purpose: The functions **sqrt** and **isqrt** compute square roots.

The function **sqrt** returns the principal square root of its numerical argument.

The function **isqrt** is the integer square root function. It returns the greatest integer less than or equal to the principal square root of its integer argument. The *integer* argument must be a nonnegative integer.

Syntax: **sqrt** *number* [*Function*]

isqrt *integer* [*Function*]

Examples:

```
> (sqrt 25)
5.0
> (isqrt 25)
5
> (sqrt -1)
#C(0.0 1.0)
> (sqrt #c(0 2))
#C(1.0 1.0)
```

truncate, round, ftruncate, fround

Purpose: The functions **truncate**, **round**, **ftruncate**, and **fround** perform type conversion operations on noncomplex numbers.

The functions **truncate** and **round** convert their arguments to integers. The function **truncate** truncates towards zero. It returns the largest integer in magnitude that is less than or equal to its argument in magnitude and that is of the same sign as its argument. The function **round** rounds to the nearest integer. If its argument lies exactly between two integers, it returns the nearest even integer.

The functions **ftruncate** and **fround** produce results identical to those of **truncate** and **round**, except that the results are returned in floating-point format.

If the optional argument *divisor* is given, then the result is mathematically equivalent to dividing *number* by *divisor* and then applying the truncate or round operation. The *divisor* must also be a noncomplex number.

Syntax:

truncate *number* `&optional` *divisor* [*Function*]

round *number* `&optional` *divisor* [*Function*]

ftruncate *number* `&optional` *divisor* [*Function*]

fround *number* `&optional` *divisor* [*Function*]

Remarks: Each of these functions returns the remainder of the result as a second value. If both arguments are of the same type, the remainder is of that type also; otherwise it is a floating-point number.

Examples:

```
> (truncate 1)
1
0
> (truncate .5)
0
.5
> (round .5)
0
.5
> (ftruncate -7 2)
-3.0
-1.0
> (fround -7 2)
-4.0
1.0
```

zerop

Purpose: The predicate **zerop** is true if its numerical argument is equal to zero; otherwise it is false.

Syntax: **zerop** *number* [*Function*]

Examples:

```
> (zerop 0)
T
> (zerop 1)
NIL
> (zerop -0.0)
T
> (zerop 0/100)
T
> (zerop #c(0 0.0))
T
```

Chapter 13. Characters

Chapter 13. Characters

About Characters

Characters in Common Lisp are data objects that represent printed symbols or operations for formatting text.

Character Set

Sun Common Lisp supports an 8-bit ASCII character set in the following manner. The set of characters that corresponds to the values between 0 and 127 inclusive represents the standard 7-bit ASCII character set. The collating sequence for this set of characters is defined in Figure 13–1. This table uses the standard ASCII character names; the corresponding values are given in octal format. The octal value for a particular character is given by the sum of its column and row numbers. Characters whose values lie between 127 and 255 inclusive are printed in hexadecimal form. For example, #\cC1 represents the character whose value is 193.

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
000	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL
010	BS	HT	NL	VT	NP	CR	SO	SI
020	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB
030	CAN	EM	SUB	ESC	FS	GS	RS	US
040	SP	!	"	#	$	%	&	'
050	(	)	*	+	,	-	.	/
060	0	1	2	3	4	5	6	7
070	8	9	:	;	<	=	>	?
100	@	A	B	C	D	E	F	G
110	H	I	J	K	L	M	N	O
120	P	Q	R	S	T	U	V	W
130	X	Y	Z	[	\	]	^	_
140	`	a	b	c	d	e	f	g
150	h	i	j	k	l	m	n	o
160	p	q	r	s	t	u	v	w
170	x	y	z	{	|	}	~	DEL

Figure 13–1. 7-bit ASCII Table

Character Attributes

Each character has three ***attributes***: ***code***, ***bits***, and ***font***. A character is uniquely defined by its code attribute, bits attribute, and font attribute. All attributes are nonnegative fixnums. In Sun Common Lisp, the font attribute of all characters is 0.

Sun Common Lisp explicitly names four bits of the bits attributes. These are the **:control**, **:meta**, **:super**, and **:hyper** bits. Their weights are defined by the constants **char-control-bit**, **char-meta-bit**, **char-super-bit**, and **char-hyper-bit**. The weight of the control bit is 1; of the meta bit, 2; of the super bit, 4; and of the hyper bit, 8.

Standard Characters and Printing Characters

Common Lisp defines a standard character set as a subtype of characters called ***standard characters***. The standard characters consist of the newline character and the 95 ***printing characters***. The font and bits attributes of all standard characters are 0.

The following table lists the printing characters according to the collating sequence. The printing characters include the space character. The printing characters are also known as ***graphic characters***.

```
␣ ! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?
@ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [ \ ] ^ _
` a b c d e f g h i j k l m n o p q r s t u v w x y z { | } ~
```

Figure 13–2. Printing Characters

The ***alphabetic characters*** are a subset of the graphic characters. The bits attribute of any alphabetic character is 0.

String Characters

String characters are a subtype of characters that can be contained in strings. A string character is any character whose bits and font attributes are 0. All of the standard characters are thus string characters.

Categories of Operations

This section groups operations on characters according to functionality.

Data Type Predicates

characterp

This predicate determines whether an object is a character.

Character Attributes

char-bit	**char-hyper-bit**
char-bits	**char-meta-bit**
char-bits-limit	**char-super-bit**
char-code	**char-font**
char-code-limit	**char-font-limit**
char-control-bit	

These functions and constants provide information on the properties of character objects.

Predicates on Characters

alpha-char-p	**lower-case-p**
alphanumericp	**upper-case-p**
both-case-p	**standard-char-p**
digit-char-p	**string-char-p**
graphic-char-p	

These predicates test properties of characters.

Character Comparison Operations

char=	**char-equal**
char/=	**char-not-equal**
char<	**char-lessp**
char>	**char-greaterp**
char<=	**char-not-greaterp**
char>=	**char-not-lessp**

These functions compare character objects.

Character Construction and Conversion Operations

character	**set-char-bit**
char-upcase	**char-int**
char-downcase	**int-char**
code-char	**char-name**
digit-char	**name-char**
make-char	

These functions are used to create and modify character objects and to convert between different representations for characters.

alpha-char-p

Purpose: The predicate **alpha-char-p** is true if its character argument is an alphabetic character; otherwise it is false.

Syntax: **alpha-char-p** *char* [*Function*]

Remarks: The alphabetic characters are a subset of the graphic characters. The bits attribute of any alphabetic character is 0.

The standard characters **A** through **Z** and **a** through **z** are alphabetic characters.

Examples:

```
> (alpha-char-p #\a)
T
> (alpha-char-p #\5)
NIL
> (alpha-char-p #\Bell)
NIL
```

alphanumericp

Purpose: The predicate **alphanumericp** is true if its character argument is an alphabetic character or a numeric character; otherwise it is false.

Syntax: **alphanumericp** *char* [*Function*]

Remarks: The alphanumeric characters are a subset of the graphic characters. The bits attribute of any alphanumeric character is 0.

The standard characters **A** through **Z**, **a** through **z**, and **0** through **9** are alphanumeric characters.

Examples:

```
> (alphanumericp #\Z)
T
> (alphanumericp #\9)
T
> (alphanumericp #\@)
NIL
> (alphanumericp #\Bell)
NIL
```

See Also: **alpha-char-p**

digit-char-p

char-bit

Purpose: The function **char-bit** tests the bit whose name is *name* in the given character object. It returns a non-**nil** value if the bit is set and **nil** if the bit is not set.

Syntax: **char-bit** *char name* [*Function*]

Remarks: The function **char-bit** recognizes the names **:control**, **:meta**, **:super**, and **:hyper** as names of character bits.

If the character argument is specified in a form that is acceptable to the macro **setf**, then **setf** may be used with **char-bit** to modify the given bit.

Examples:

```
> (char-bit (make-char #\a 1) :control)
T
> (char-bit #\a :meta)
NIL
> (let ((tmp #\a))
    (setf (char-bit tmp :hyper) t)
    (char-bit tmp :hyper))
T
```

See Also: **set-char-bit**

char-bits

Purpose: The function **char-bits** returns the bits attribute of its character argument. The result is a nonnegative integer less than the value of the constant **char-bits-limit**.

Syntax: **char-bits** *char* [*Function*]

Examples:

```
> (char-bits #\b)
0
> (char-bits (make-char #\@ 5))
5
> (char-bits #\Control-A)
1
```

See Also: **char-bits-limit**

char-bits-limit

Purpose: The constant **char-bits-limit** is a nonnegative integer that defines the upper exclusive bound on the value of the result of the function **char-bits**.

The value of **char-bits-limit** in Sun Common Lisp is 2^4.

Syntax: **char-bits-limit** [*Constant*]

Remarks: The function **char-bits** returns the bits attribute of a given character.

Examples:

```
> char-bits-limit
16
```

See Also: **char-bits**

char-code

Purpose: The function **char-code** returns the code attribute of its character argument. The result is a nonnegative integer less than the value of the constant **char-code-limit**.

Syntax: **char-code** *char* [*Function*]

Examples:

```
> (char-code #\Bell)
7
> (char-code #\a)
97
> (char-code #\Control-a)
65
> (char-code #\Control-\a)
97
```

See Also: **char-code-limit**

char-code-limit

Purpose: The constant **char-code-limit** is a nonnegative integer that defines the upper exclusive bound on the value of the result of the function **char-code**.

The value of **char-code-limit** in Sun Common Lisp is 2^8.

Syntax: **char-code-limit** *[Constant]*

Remarks: The function **char-code** returns the code attribute of a given character.

Examples:

```
> char-code-limit
256
```

See Also: **char-code**

char-control-bit, char-meta-bit, char-super-bit, char-hyper-bit

Purpose: The constants **char-control-bit**, **char-meta-bit**, **char-super-bit**, and **char-hyper-bit** define the weights of the four named bits attributes.

The value of **char-control-bit** is 1; **char-meta-bit**, 2; **char-super-bit**, 4; and **char-hyper-bit**, 8.

Syntax:

char-control-bit [*Constant*]

char-meta-bit [*Constant*]

char-super-bit [*Constant*]

char-hyper-bit [*Constant*]

Examples:

```
> char-control-bit
1
> char-meta-bit
2
> char-super-bit
4
> char-hyper-bit
8
```

char-font

Purpose: The function **char-font** returns the font attribute of its character argument. The result is a nonnegative integer less than the value of the constant **char-font-limit**.

Syntax: **char-font** *char* [*Function*]

Remarks: In Sun Common Lisp, the font attribute of all characters is 0.

Examples:

```
> (char-font #\Control-A)
0
```

See Also: **char-font-limit**

char-font-limit

Purpose: The constant **char-font-limit** is a nonnegative integer that defines the upper exclusive bound on the value of the result of the function **char-font**.

The value of **char-font-limit** in Sun Common Lisp is 1.

Syntax: **char-font-limit** [*Constant*]

Remarks: The function **char-font** returns the font attribute of a given character.

Examples:

```
> char-font-limit
1
```

See Also: **char-font**

char-int

Purpose: The function **char-int** returns the nonnegative integer that encodes its character argument.

Syntax: **char-int** *char* [*Function*]

Remarks: If the font and bits attributes of the character are 0, the results of **char-int** and **char-code** are the same.

Examples:

```
> (char-int #\Null)
0
> (char-int #\Control-Null)
256
> (char-int #\Meta-Null)
512
> (char-int #\Super-Null)
1024
> (char-int #\Hyper-Null)
2048
```

See Also: **char-code**

int-char

char-name, name-char

Purpose: The functions **char-name** and **name-char** provide mappings between characters and character names.

The function **char-name** returns the name of its character argument as a string. If the character has no name, **char-name** returns **nil**.

The function **name-char** returns the character object whose name is *name*. If such a character does not exist, then **name-char** returns **nil**.

Characters having names may be written as #\ followed by the character name. Character names are not case sensitive. All nongraphic characters whose font and bits attributes are 0 have names.

Syntax:

char-name *char* [*Function*]

name-char *name* [*Function*]

Remarks: The function **name-char** recognizes, and the function **char-name** returns, the character names in the following table. The ASCII values of the characters are given in octal format.

Null	0	Page	14	ETB	27
SOH	1	Return	15	CAN	30
STX	2	SO	16	EM	31
ETX	3	SI	17	SUB	32
EOT	4	DLE	20	ESC	33
ENQ	5	DC1	21	FS	34
ACK	6	DC2	22	GS	35
Bell	7	DC3	23	RS	36
Backspace	10	DC4	24	US	37
Tab	11	NAK	25	Space	40
Newline	12	SYN	26	Rubout	177
VT	13				

The function **name-char** also recognizes the following standard ASCII names, although the longer names above are returned by **char-name**:

NUL	0	HT	11	CR	15
BEL	7	NL	12	SP	40
BS	10	NP	14	DEL	177

The function **name-char** also recognizes the following character name as a name for the newline character:

Linefeed   12

The function **name-char** also recognizes names of the form c*xx*, where *xx* are hexadecimal digits, and returns the character with the given code. The function **char-name** returns names of this form for characters with codes greater than 127.

The *name* argument of **name-char** must be a symbol or a string. The *name* argument is not case sensitive.

The function **name-char** is used by the reader to parse characters entered with the #\ syntax. The function **char-name** is used by the printer to print characters when ***print-escape*** is non-**nil**.

Examples:

```
> (char-name #\Cr)
"Return"
> (char-name #\a)
NIL
> (char-name #\Control-Null)
NIL
> (name-char 'linefeed)
#\Newline
> (name-char "A")
NIL
> (name-char "space")
#\Space
```

char-upcase, char-downcase

Purpose: The functions **char-upcase** and **char-downcase** perform case conversions upon characters.

The function **char-upcase** attempts to convert its character argument to its uppercase equivalent. It returns a character with the same bits and font attributes as its argument. If the code attribute of the result differs from that of the argument, the case conversion has succeeded.

The function **char-downcase** attempts to convert its character argument to its lowercase equivalent. It returns a character with the same bits and font attributes as its argument. If the code attribute of the result differs from that of the argument, the case conversion has succeeded.

Syntax:

char-upcase *char* [*Function*]

char-downcase *char* [*Function*]

Remarks: These functions perform case conversions only on alphabetic characters. If the bits attribute of *char* is nonzero, **char-upcase** and **char-downcase** have no effect.

Examples:

```
> (char-upcase #\a)
#\A
> (char-upcase #\A)
#\A
> (char-downcase #\A)
#\a
> (char-downcase #\9)
#\9
```

See Also:

upper-case-p

lower-case-p

char=, char/=, char<, char<=, char>, char>=, char-equal, char-not-equal, char-lessp, char-not-greaterp, char-greaterp, char-not-lessp

Purpose: These predicates compare character objects. If the characters are standard characters, they are compared according to the ordering given by Figure 13-1.

If characters differ in either their code, bits, or font attributes, they are considered to be different.

If characters agree in their bits and font attributes, they are compared according to the ordering of their code attributes.

The predicate **char=** is true if the specified characters are all the same; otherwise it is false.

The predicate **char/=** is true if the specified characters are all different; otherwise it is false.

The predicate **char<** is true if the specified characters are all in increasing order; otherwise it is false.

The predicate **char<=** is true if the specified characters are all in nondecreasing order; otherwise it is false.

The predicate **char>** is true if the specified characters are all in decreasing order; otherwise it is false.

The predicate **char>=** is true if the specified characters are all in nonincreasing order; otherwise it is false.

The predicates **char-equal**, **char-not-equal**, **char-lessp**, **char-not-greaterp**, **char-greaterp**, and **char-not-lessp** are like **char=**, **char/=**, **char<**, **char<=**, **char>**, and **char>=** respectively but ignore differences in case and in bits attributes.

Syntax:

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*]

char-equal *character* `&rest` *more-characters* [*Function*]

char-not-equal *character* `&rest` *more-characters* [*Function*]

char-lessp *character* `&rest` *more-characters* [*Function*]

char-greaterp *character* `&rest` *more-characters* [*Function*]

char-not-greaterp *character* `&rest` *more-characters* [*Function*]

char-not-lessp *character* `&rest` *more-characters* [*Function*]

Examples:

```
> (char= #\A #\A)
T
> (char= #\A #\a)
NIL
> (char>= #\z #\x #\n #\a #\a)
T
> (char-equal #\a #\A #\a)
T
> (char-greaterp #\b #\a #\A)
NIL
```

character

Purpose: The function **character** coerces its object to be a character. If the coercion is not possible, an error is signaled.

The following objects may be coerced into characters: strings of length 1, symbols whose print names are of length 1, and nonnegative integers *n* for which `(int-char n)` is defined.

Syntax: **character** *object* [*Function*]

Remarks: Coercing a string of length 1 results in the character contained in that string. Coercing a symbol whose print name is of length 1 results in the character contained in that print name string. Coercing a nonnegative integer for which `(int-char n)` is defined results in the character defined by `(int-char n)`.

Examples:

```
> (character "a")
#\a
> (character 'a)
#\A
> (character 120)
#\x
```

See Also: **coerce**

int-char

characterp

Purpose: The predicate **characterp** is true if its argument is a character; otherwise it is false.

Syntax: **characterp** *object* [*Function*]

Examples:

```
> (characterp #\rubout)
T
> (characterp #\newline)
T
> (characterp #\a)
T
> (characterp 12)
NIL
```

code-char

Purpose: The function **code-char** creates and returns a character object with the specified *code*, *bits*, and *font* attributes. If it is not possible to create such a character, **code-char** returns **nil**.

Syntax: **code-char** *code* `&optional` (*bits* 0) (*font* 0) [*Function*]

Remarks: The *code*, *bits*, and *font* arguments must all be nonnegative integers.

Examples:

```
> (code-char 65)
#\A
> (code-char 65 15)
#\Control-Meta-Super-Hyper-A
> (code-char 65 0 1)
NIL
```

See Also: **make-char**

digit-char

Purpose: The function **digit-char** creates and returns a character object that represents the value *weight* in the specified radix and that has the specified *font* attribute.

If it succeeds, **digit-char** returns the new character. If it is not possible to create such a character object, **digit-char** returns **nil**.

If the *font* argument is 0, if *radix* is an integer greater than or equal to 2 and less than or equal to 36, and if *weight* is a nonnegative integer less than *radix*, then **digit-char** always succeeds.

Syntax: **digit-char** *weight* `&optional` (*radix* 10) (*font* 0) [*Function*]

Remarks: If more than one such character is possible, one of these is returned consistently; uppercase characters are favored over lowercase characters.

Examples:

```
> (digit-char 0)
#\0
> (digit-char 10 11)
#\A
> (digit-char 10 10)
NIL
```

digit-char-p

Purpose: The predicate **digit-char-p** tests whether its character argument is a digit of the specified radix. If it is, **digit-char-p** returns a nonnegative integer (in radix 10) that represents the weight of the character in the specified radix; otherwise it returns **nil**.

Syntax: **digit-char-p** *char* `&optional` (*radix* 10) [*Function*]

Remarks: The *radix* argument must be a nonnegative integer.

The standard characters **0** through **9** and **A** through **Z** (or, equivalently, **a** through **z**) can be digit characters. The weights in radix 10 of the standard characters **A** through **Z** (or **a** through **z**) are 10 through 35 respectively.

Examples:

```
> (digit-char-p #\0)
0
> (digit-char-p #\a 11)
10
> (digit-char-p #\Z 36)
35
> (digit-char-p #\D 13)
NIL
```

graphic-char-p

Purpose: The predicate **graphic-char-p** tests whether its character argument is a graphic, or printing, character. It is true if *char* is a printing character; otherwise it is false.

Syntax: **graphic-char-p** *char* [*Function*]

Remarks: The graphic characters consist of all of the standard characters except the newline character.

The bits attribute of any graphic character is 0.

Examples:

```
> (graphic-char-p #\~)
T
> (graphic-char-p #\Space)
T
> (graphic-char-p #\Bell)
NIL
```

int-char

Purpose: The function **int-char** returns the character object that is encoded by the *integer* argument. If no such character exists, **int-char** returns **nil**.

Syntax: **int-char** *integer* [*Function*]

Remarks: For any character object *c* that is returned by **int-char**, the value of `(char-int c)` is equal to the value of the *integer* argument.

Examples:

```
> (int-char 65)
#\A
> (int-char 97)
#\a
```

See Also: **char-int**

make-char

Purpose: The function **make-char** creates and returns a character object whose code attribute is the same as that of its character argument and whose bits and font attributes are specified by the *bits* and *font* arguments.

If it is not possible to create such a character, **make-char** returns **nil**.

Syntax: **make-char** *char* `&optional` (*bits* 0) (*font* 0) [*Function*]

Remarks: The *bits* and *font* arguments must be nonnegative integers.

If both the *bits* and *font* arguments are 0, **make-char** always succeeds.

Examples:

```
> (make-char #\a)
#\a
> (make-char #\A 15)
#\Control-Meta-Super-Hyper-A
> (make-char #\a 0 char-font-limit)
NIL
```

See Also: **code-char**

set-char-bit

Purpose: The function **set-char-bit** is used to set a bit in a character object. It returns a new character object in which the bit with the given *name* has the specified logical value.

Syntax: **set-char-bit** *char name logical-value* [*Function*]

Remarks: The function **set-char-bit** recognizes the names **:control**, **:meta**, **:super**, and **:hyper** as names of character bits.

Examples:

```
> (set-char-bit #\a :control nil)
#\a
> (set-char-bit #\0 :control t)
#\Control-0
```

standard-char-p

Purpose: The predicate **standard-char-p** tests whether its character argument is a standard character. It is true if *char* is a standard character; otherwise it is false.

Syntax: **standard-char-p** *char* [*Function*]

Remarks: The font and bits attributes of any standard character are 0.

Examples:

```
> (standard-char-p #\Space)
T
> (standard-char-p #\~)
T
> (standard-char-p #\Bell)
NIL
```

string-char-p

Purpose: The predicate **string-char-p** tests whether its character argument is a character that can be an element of a string. It is true if the character is a string character; otherwise it is false.

Syntax: **string-char-p** *char* [*Function*]

Remarks: The standard characters are a subset of the string characters. The bits and font attributes of any string character are 0.

Examples:

```
> (string-char-p #\~)
T
> (string-char-p #\Space)
T
> (string-char-p #\Bell)
T
> (string-char-p (code-char 32 15))
NIL
```

upper-case-p, lower-case-p, both-case-p

Purpose: The predicates **upper-case-p**, **lower-case-p**, and **both-case-p** test the case of a character.

The predicate **upper-case-p** is true if its argument is an uppercase character; otherwise it is false.

The predicate **lower-case-p** is true if its argument is a lowercase character; otherwise it is false.

The predicate **both-case-p** is true if its argument is an uppercase character and a corresponding lowercase character exists or if its argument is a lowercase character and a corresponding uppercase character exists.

Syntax:

upper-case-p *char* [*Function*]

lower-case-p *char* [*Function*]

both-case-p *char* [*Function*]

Remarks: Any character that is either an uppercase character or a lowercase character is an alphabetic character and therefore a printing character. Its bits attribute is 0.

The standard characters **A** through **Z** are uppercase, and **a** through **z** are lowercase.

Examples:

```
> (upper-case-p #\A)
T
> (upper-case-p #\a)
NIL
> (lower-case-p #\Bell)
NIL
> (both-case-p #\a)
T
> (both-case-p #\5)
NIL
```

See Also: **char-upcase**

char-downcase

Chapter 14. Sequences

Chapter 14. Sequences

About Sequences

Sequences are ordered sets of elements and include both lists and vectors (one-dimensional arrays). Common Lisp provides operations for searching, modifying, sorting, merging, mapping, concatenating, and reducing sequences.

The operations presented here apply to all types of sequences. Operations that are specific to lists and to vectors are discussed in the chapters "Lists" and "Arrays" respectively.

Categories of Operations

This section groups operations on sequences according to functionality.

Basic Sequence Operations

copy-seq	**make-sequence**
elt	**subseq**
length	

These functions create new sequences and perform basic sequence operations.

Searching Sequences

count	**mismatch**
count-if	**position**
count-if-not	**position-if**
find	**position-if-not**
find-if	**search**
find-if-not	

These functions search sequences to locate elements that meet some criterion.

Sorting and Merging Sequences

merge	**stable-sort**
sort	

These functions sort and merge sequences.

Modifying Sequences

delete	**remove-if-not**
delete-if	**remove-duplicates**
delete-if-not	**replace**
delete-duplicates	**reverse**
fill	**nreverse**
nsubstitute	**substitute**
nsubstitute-if	**substitute-if**
nsubstitute-if-not	**substitute-if-not**
remove	
remove-if	

These functions modify sequences or produce modified copies of their sequence arguments.

Concatenating, Mapping, and Reducing Sequences

every	**concatenate**
some	**map**
notevery	**reduce**
notany	

These functions perform concatenation, mapping, and reduction operations on sequences.

concatenate

Purpose: The function **concatenate** creates and returns a new sequence that contains copies of the individual elements of all of the sequence arguments in the order in which they occur in the argument list. The new sequence is of type *result-type*, which must be a subtype of the sequence data type and compatible with the type of the sequence elements.

Syntax: **concatenate** *result-type* `&rest` *sequences* [*Function*]

Examples:

```
> (concatenate 'string "all" " " "together" " " "now")
"all together now"
> (concatenate 'list)
NIL
> (concatenate 'list "all" "boy")
(#\a #\l #\l #\b #\o #\y)
```

See Also: **append**

copy-seq

Purpose: The function **copy-seq** creates and returns a copy of its *sequence* argument.

Syntax: **copy-seq** *sequence* [*Function*]

Remarks: The resulting sequence is **equalp** to the original.

Examples:

```
> (setq str "a string")
"a string"
> (equalp str (copy-seq str))
8
> (eql str (copy-seq str))
NIL
```

count, count-if, count-if-not

Purpose: The functions **count**, **count-if**, and **count-if-not** count either the number of sequence elements that match a particular item or the number of elements that satisfy some test predicate. The count value is returned as a nonnegative integer.

If the **:start** and **:end** keyword arguments are specified, only the subsequence they delimit is searched.

Syntax:

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]*

Remarks: The keyword arguments **:start** and **:end** take integer values that specify offsets into the original sequence. The **:start** argument marks the beginning position of the subsequence; the **:end** argument marks the position following the last element of the subsequence. The start value defaults to 0; the end value defaults to the length of the sequence.

The function **eql** is the default that is used by **count** for matching the *item* argument against the sequence elements. Either the keyword argument **:test** or the keyword argument **:test-not** may be used with **count** to specify a test function other than **eql**.

The keyword **:key** may be used to specify that a part of a sequence element should be tested. The arguments passed to the test function of **count** or to the test predicate of **count-if** or **count-if-not** are extracted from the sequence elements according to the **:key** function. If **:key** is not specified, the elements themselves are used.

The **:from-end** argument has no effect on the result.

Examples:

```
> (count #\a "how many A's are there in here?")
2
> (count-if-not #'oddp '((1) (2) (3) (4)) :key #'car)
2
```

elt

Purpose: The function **elt** accesses and returns the sequence element specified by *index*.

Syntax: **elt** *sequence index* [*Function*]

Remarks: The index is an offset value from the beginning of the sequence; indexing is zero-origin. The index value must be a nonnegative integer less than the length of the sequence. If the sequence is a vector having a fill pointer, **elt** observes the length specified by the fill pointer.

The macro **setf** may be used with **elt** to destructively replace a sequence element.

Examples:

```
> (setq str "0123456789")
"0123456789"
> (elt str 6)
#\6
> (setf (elt str 0) #\#)
#\#
> str
"#123456789"
```

See Also: **aref**

nth

every, some, notevery, notany

Purpose: The functions **every**, **some**, **notevery**, and **notany** test sequence elements for satisfaction of a given predicate. These functions operate on as many sequence arguments as the given predicate takes arguments. The predicate is invoked on each successive set of elements, one from each sequence.

The function **every** returns **nil** as soon as any invocation of the predicate returns **nil**. If the end of any sequence is reached, **every** returns a non-**nil** value.

The function **some** returns the first non-**nil** value that is returned by the invocation of the predicate. If the end of any sequence is reached, **some** returns **nil**.

The function **notevery** returns a non-**nil** value as soon as any invocation of the predicate returns **nil**. If the end of any sequence is reached, **notevery** returns **nil**.

The function **notany** returns **nil** as soon as any invocation of the predicate returns a non-**nil** value. If the end of any sequence is reached, **notany** returns a non-**nil** value.

Syntax:

every *predicate sequence* `&rest` *more-sequences* [*Function*]

some *predicate sequence* `&rest` *more-sequences* [*Function*]

notevery *predicate sequence* `&rest` *more-sequences* [*Function*]

notany *predicate sequence* `&rest` *more-sequences* [*Function*]

Examples:

```
> (every #'string-char-p "abc")
T
> (some #'= '(1 2 3 4 5) '(5 4 3 2 1))
T
> (notevery #'< '(1 2 3 4) '(5 6 7 8) '(9 10 11 12))
NIL
```

fill

Purpose: The function **fill** destructively modifies its *sequence* argument by replacing each element with the specified *item*. It returns the modified sequence.

If the **:start** and **:end** keyword arguments are specified, only the subsequence they delimit is modified.

Syntax: **fill** *sequence item* `&key :start :end` [*Function*]

Remarks: The *item* argument must be an object that is compatible with the sequence type.

The keyword arguments **:start** and **:end** take integer values that specify offsets into the original sequence. The **:start** argument marks the beginning position of the subsequence; the **:end** argument marks the position following the last element of the subsequence. The start value defaults to 0; the end value defaults to the length of the sequence.

Examples:

```
> (fill '(0 1 2 3 4 5) '(444))
((444) (444) (444) (444) (444) (444))
> (fill "01234" #\e :start 3)
"012ee"
```

See Also: **replace**

find, find-if, find-if-not

Purpose: The functions **find**, **find-if**, and **find-if-not** each search a sequence for an element that matches a particular item or an element that satisfies some test predicate. If they succeed, the leftmost such element found is returned; otherwise **nil** is returned.

If the **:start** and **:end** keyword arguments are specified, only the subsequence they delimit is searched.

If the **:from-end** keyword argument is non-**nil**, these functions search for the rightmost element that meets the test criterion.

Syntax:

find *item sequence* `&key :from-end :test :test-not :start :end :key` [*Function*]

find-if *test sequence* `&key :from-end :start :end :key` [*Function*]

find-if-not *test sequence* `&key :from-end :start :end :key` [*Function*]

Remarks: The keyword arguments **:start** and **:end** take integer values that specify offsets into the original sequence. The **:start** argument marks the beginning position of the subsequence; the **:end** argument marks the position following the last element of the subsequence. The start value defaults to 0; the end value defaults to the length of the sequence.

The function **eql** is the default that is used by **find** for matching the *item* argument against the sequence elements. Either the keyword argument **:test** or the keyword argument **:test-not** may be used with **find** to specify a test function other than **eql**.

The keyword **:key** may be used to specify that a part of a sequence element should be tested. The arguments passed to the test function of **find** or to the test predicate of **find-if** or **find-if-not** are extracted from the sequence elements according to the **:key** function. If **:key** is not specified, the elements themselves are used.

Examples:

```
> (find #\d "here are some letters that can be looked at" :test #'char>)
#\Space
> (find-if #'oddp '(1 2 3 4 5) :end 3 :from-end t)
3
```

length

Purpose: The function **length** returns the length of its *sequence* argument as an integer value.

If the sequence is a vector having a fill pointer, **length** returns the value specified by the fill pointer.

Syntax: **length** *sequence* [*Function*]

Remarks: The function **length** may loop infinitely on circular lists, unlike **list-length**.

Examples:

```
> (length "abc")
3
> (setq str (make-array '(3) :element-type 'string-char
                              :initial-contents "abc"
                              :fill-pointer t))
"abc"
> (length str)
3
> (setf (fill-pointer str) 2)
2
> (length str)
2
```

See Also: **list-length**

make-sequence

Purpose: The function **make-sequence** creates and returns a sequence of the specified type and length.

If the **:initial-element** argument is specified, all the sequence elements are initialized to its value.

Syntax: **make-sequence** *type size* `&key :initial-element` [*Function*]

Remarks: The *type* argument must specify a subtype of the sequence data type.

The **:initial-element** argument must be of a type compatible with the type of the sequence.

Examples:

```
> (make-sequence 'list 0)
NIL
> (make-sequence 'string 26 :initial-element #\.)
".........................."
```

See Also: **make-array**

make-list

map

Purpose: The function **map** creates and returns a new sequence. The mapping operation involves applying a function to successive sets of arguments in which one argument is obtained from each sequence. The resulting sequence contains the results returned by the function.

The *function* argument must take as many arguments as there are sequence arguments.

The resulting sequence is the same length as the shortest of the sequence arguments. It is of type *result-type*, which must be a subtype of the sequence data type and compatible with the types of the sequence elements.

Syntax: **map** *result-type function sequence* &rest *more-sequences* [*Function*]

Remarks: The *result-type* argument may be specified as **nil**. In this case, the *function* argument is invoked only for its side effects, and **map** returns **nil**.

Examples:

```
> (map 'string #'(lambda(x y)
                   (char "01234567890ABCDEF" (mod (+ x y) 16)))
       '(1 2 3 4)
       '(10 9 8 7))
"AAAA"
> (setq seq '("lower" "UPPER" "" "123"))
("lower" "UPPER" "" "123")
> (map nil #'nstring-upcase seq)
NIL
> seq
("LOWER" "UPPER" "" "123")
```

See Also: **mapcar**

merge

Purpose: The function **merge** destructively merges two sequences and returns the resulting sequence. The sequence arguments are merged according to the order determined by the *predicate* and **:key** arguments. The resulting sequence is of type *result-type*, which must be a subtype of the sequence data type and compatible with the types of the sequence elements.

The order of the elements in the result sequence is determined by the *predicate* argument. The *predicate* must be a function of two elements. It should return a non-**nil** value if the element corresponding to its first argument is to precede the element corresponding to the second in the result sequence; otherwise it should return **nil**.

If the sequences were originally sorted according to the given predicate, the result sequence is sorted in like manner. If not, the result is an interleaving of the two sequences in which the order of the elements of each individual sequence is preserved in the result sequence.

Syntax: **merge** *result-type sequence1 sequence2 predicate* `&key :key` [*Function*]

Remarks: The **merge** operation is stable. That is, if two elements are considered equivalent by the *predicate* function, the element from *sequence1* precedes the element from *sequence2* in the resulting sequence.

The keyword **:key** may be used to specify that a part of a sequence element should be tested. Its argument should be a function of one argument that extracts the part to be tested from the sequence element. If **:key** is not specified, the element itself is treated as the key.

Examples:

```
> (merge 'list '(1 3 5) '(2 4 6) #'<)
(1 2 3 4 5 6)
```

mismatch

Purpose: The function **mismatch** compares two sequences element by element. If they are of the same length and if each corresponding pair of elements satisfies the test, then **mismatch** returns **nil**. Otherwise **mismatch** returns the offset of the leftmost nonmatching element from the beginning of *sequence1*.

If one sequence is shorter than the other, but the two otherwise match, the result is the offset from the beginning of *sequence1* of the element following the last one tested.

If the **:start** and **:end** keyword arguments are specified, only the subsequences they delimit are compared.

If the **:from-end** keyword argument is non-**nil**, the effect is as if the two sequences were compared from right to left. If they fail to match, **mismatch** returns one plus the offset from the beginning of *sequence1* of the rightmost position in which the sequences differ.

Syntax: **mismatch** *sequence1 sequence2* `&key :from-end :test :test-not :key :start1 :start2 :end1 :end2` [*Function*]

Remarks: The **:start** and **:end** keyword arguments take integer values that specify offsets into the original sequences. The **:start** arguments mark the beginning positions of the subsequences; the **:end** arguments mark the positions following the last elements of the subsequences. The start values default to 0; the end values default to the length of the sequences.

Whether or not a sequence element matches another sequence element is determined by the functions specified by the **:test** and **:key** arguments. If a test argument is not specified, **eql** is used. Either the keyword **:test** or the keyword **:test-not** may be used to specify a test function other than **eql**.

The keyword **:key** may be used to specify that a part of a sequence element should be tested. Its argument should be a function of one argument that extracts the part to be tested from the element. If **:key** is not specified, the elements themselves are used.

Examples:

```
> (mismatch "abcd" "ABCDE" :test #'char-equal)
4
> (mismatch '(3 2 1 1 2 3) '(1 2 3) :from-end t)
3
> (mismatch '(1 2 3) '(2 3 4) :test-not #'eq :key #'oddp)
NIL
```

position, position-if, position-if-not

Purpose:

The functions **position**, **position-if**, and **position-if-not** each search a sequence for an element that matches a particular item or for an element that satisfies some test predicate. If they succeed, the offset of the leftmost such element from the beginning of the sequence is returned as an integer value; otherwise **nil** is returned.

If **:start** and **:end** keyword arguments are specified, only the subsequence they delimit is searched.

If the **:from-end** keyword argument is non-**nil**, these functions search for the rightmost element that meets the test criterion.

In all cases the offset value returned is relative to the entire sequence, not to the subsequence, regardless of the direction of search.

Syntax:

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*]

Remarks:

The keyword arguments **:start** and **:end** take integer values that specify offsets into the original sequence. The **:start** argument marks the beginning position of the subsequence; the **:end** argument marks the position following the last element of the subsequence. The start value defaults to 0; the end value defaults to the length of the sequence.

The function **eql** is the default that is used by **position** for matching the *item* argument against the sequence elements. Either the keyword argument **:test** or the keyword argument **:test-not** may be used with **position** to specify a test function other than **eql**.

The keyword **:key** may be used to specify that a part of a sequence element should be tested. The arguments passed to the test function of **position** or to the test predicate of **position-if** or **position-if-not** are extracted from the sequence elements according to the **:key** function. If **:key** is not specified, the elements themselves are used.

Examples:

```
> (position #\a "baobab" :from-end t)
4
> (position-if #'oddp '((1) (2) (3) (4)) :start 1 :key #'car)
2
> (position 595 '())
NIL
```

reduce

Purpose: The function **reduce** performs a reduction operation on the elements of a sequence. The reduction uses the binary operator specified by *function*. The resulting value is returned.

If the **:start** and **:end** arguments are specified, only the subsequence they delimit is reduced.

The reduction operation is left-associative if the **:from-end** argument is defaulted or **nil**; otherwise it is right-associative.

If the **:initial-value** argument is specified, its value is used as the first operand in the reduction operation.

Syntax: **reduce** *function sequence* &key :from-end :start :end :initial-value [*Function*]

Remarks: If there is exactly one element in the subsequence and the **:initial-value** argument is not specified, that element is returned. If the subsequence is empty and the **:initial-value** argument is specified, that initial value is returned. In neither of these cases is the reduction function invoked. If the subsequence is empty and **:initial-value** is not specified, then **reduce** returns the result of calling *function* with no arguments.

The keyword arguments **:start** and **:end** take integer values that specify offsets into the original sequence. The **:start** argument marks the beginning position of the subsequence; the **:end** argument marks the position following the last element of the subsequence. The start value defaults to 0; the end value defaults to the length of the sequence.

Examples:

```
> (reduce #'* '(1 2 3 4 5))
120
> (reduce #'append '((1) (2)) :initial-value '(i n i t))
(I N I T 1 2)
```

remove, remove-if, remove-if-not, delete, delete-if, delete-if-not

Purpose: The functions **remove**, **remove-if**, and **remove-if-not** return a copy of their sequence argument from which the elements that match a particular item or the elements that satisfy some test predicate have been removed. The elements of the resulting sequence remain in the same order as in the original sequence.

If the **:start** and **:end** keyword arguments are specified, only the subsequence they delimit is affected.

If the **:count** argument is specified, only the leftmost number of elements specifed by **:count** that satisfy the test condition are removed.

The **:from-end** argument has an effect only if **:count** is specified and **:from-end** is non-**nil**. In this case only the rightmost number of elements specified by **:count** that satisfy the test condition are removed.

The functions **delete**, **delete-if**, and **delete-if-not** are like **remove**, **remove-if**, and **remove-if-not** respectively, but they may modify their *sequence* argument.

Syntax:

remove *item sequence* &key `:from-end :test :test-not :start :end :count :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*]

delete *item sequence* &key `:from-end :test :test-not :start :end :count :key` [*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*]

Remarks: The result of **remove**, **remove-if**, or **remove-if-not** may share cells with the original sequence if the sequence is a list.

The result of **delete**, **delete-if**, or **delete-if-not** may or may not be **eq** to the original sequence.

The keyword arguments **:start** and **:end** take integer values that specify offsets into the original sequence. The **:start** argument marks the beginning position of the subsequence; the **:end** argument marks the position following the last element of the subsequence. The start value defaults to 0; the end value defaults to the length of the sequence.

The function **eql** is the default that is used by **remove** and **delete** for matching the *item* argument against the sequence elements. Either the keyword argument **:test** or the keyword argument **:test-not** may be used with **remove** or **delete** to specify a test function other than **eql**.

The keyword **:key** may be used to specify that a part of a sequence element should be tested. The arguments passed to the test function of **remove** and **delete** or to the test predicate of **remove-if**, **remove-if-not**, **delete-if**, or **delete-if-not** are extracted from the sequence elements according to the **:key** function. If **:key** is not specified, the elements themselves are used.

Examples:

```
> (remove 4 '(1 3 4 5 9))
(1 3 5 9)
> (setq list '(list of four elements))
(LIST OF FOUR ELEMENTS)
> (setq list2 (copy-seq list))
(LIST OF FOUR ELEMENTS)
> (setq list3 (delete 'four list))
(LIST OF ELEMENTS)
> (equal list list2)
NIL
> (remove-if-not #'evenp '(1 2 3 4 5 6 7 8 9) :count 2 :from-end t)
(1 2 3 4 5 6 8)
```

remove-duplicates, delete-duplicates

Purpose: The function **remove-duplicates** returns a modified copy of its sequence argument from which any element that duplicates an element occurring later in the sequence has been removed.

If the **:from-end** argument is non-**nil**, then any element that duplicates an element occurring earlier in the sequence is removed.

If the **:start** and **:end** keyword arguments are specified, only the subsequence they delimit is involved in the operation.

The elements of the resulting sequence remain in the same order as in the original sequence.

The function **delete-duplicates** is like **remove-duplicates**, but **delete-duplicates** may modify its *sequence* argument.

Syntax: **remove-duplicates** *sequence* `&key :from-end :test :test-not :start :end :key` [*Function*]

delete-duplicates *sequence* `&key :from-end :test :test-not :start :end :key` [*Function*]

Remarks: The keyword arguments **:start** and **:end** take integer values that specify offsets into the original sequence. The **:start** argument marks the beginning position of the subsequence; the **:end** argument marks the position following the last element of the subsequence. The start value defaults to 0; the end value defaults to the length of the sequence.

The function **eql** is the default that is used by **remove-duplicates** and **delete-duplicates** for determining whether two sequence elements match. Either the keyword argument **:test** or the keyword argument **:test-not** may be used with **remove-duplicates** or **delete-duplicates** to specify a test function other than **eql**.

The keyword **:key** may be used to specify that a part of a sequence element should be tested. The arguments passed to the test function of **remove-duplicates** or **delete-duplicates** are extracted from the sequence elements according to the **:key** function. If **:key** is not specified, the elements themselves are used.

Examples:

```
> (remove-duplicates "aBcDAbCd" :test #'char-equal :from-end t)
"aBcD"
> (delete-duplicates '(0 1 2 3 4 5 6) :key #'oddp :start 1 :end 6)
(0 4 5 6)
```

replace

Purpose: The function **replace** destructively modifies a sequence by replacing one subsequence with another. The *sequence1* argument is modified by replacing the subsequence delimited by the keyword arguments **:start1** and **:end1** with the subsequence of *sequence2* that is delimited by the keyword arguments **:start2** and **:end2**. The resulting sequence is returned.

If the subsequences are not of equal length, the length of the shorter subsequence determines the number of elements that are replaced. The remaining elements at the end of the longer subsequence do not take part in the operation.

The elements of *sequence2* must be of a type compatible with *sequence1*.

Syntax: **replace** *sequence1 sequence2* &key :start1 :end1 :start2 :end2 [*Function*]

Remarks: If the two sequences are the same object, the effect of the operation is that of the simultaneous replacement of one subsequence by the other. If, however, the two sequences are not the same but share some substructure, the contents of the resulting sequence are unpredictable.

The **:start** and **:end** keyword arguments take integer values that specify offsets into the original sequences. The **:start** arguments mark the beginning positions of the subsequences; the **:end** arguments mark the positions following the last elements of the subsequences. The start values default to 0; the end values default to the length of the sequences.

Examples:

```
> (replace "abcdefghij" "0123456789" :start1 4 :end1 7 :start2 4)
"abcd456hij"
> (setq lst "012345678")
"012345678"
> (replace lst lst :start1 2 :start2 0)
"010123456"
> lst
"010123456"
```

See Also: **fill**

reverse, nreverse

Purpose: The functions **reverse** and **nreverse** return a sequence in which the order of the elements of the *sequence* argument is reversed.

The functions **reverse** and **nreverse** differ in that **reverse** creates and returns a new sequence, whereas **nreverse** may modify its argument.

Syntax:

reverse *sequence* [*Function*]

nreverse *sequence* [*Function*]

Remarks: The sequence produced by **nreverse** may or may not be **eq** to its argument.

Examples:

```
> (setq str "abc")
"abc"
> (reverse str)
"cba"
> str
"abc"
> (nreverse str)
"cba"
```

search

Purpose: The function **search** searches *sequence2* for a subsequence that matches *sequence1*. If the search succeeds, **search** returns the offset into *sequence2* of the first element of the leftmost matching subsequence; otherwise **search** returns **nil**.

If the **:start** and **:end** keyword arguments are specified, only the subsequences they delimit are involved in the search.

If the **:from-end** keyword argument is non-**nil**, the index of the first element of the rightmost such subsequence is returned.

Syntax: **search** *sequence1 sequence2* &key `:from-end :test :test-not :key :start1 :start2 :end1 :end2` [*Function*]

Remarks: The **:start** and **:end** keyword arguments take integer values that specify offsets into the original sequences. The **:start** arguments mark the beginning positions of the subsequences; the **:end** arguments mark the positions following the last elements of the subsequences. The start values default to 0; the end values default to the length of the sequences.

The function **eql** is the default that is used by **search** for matching the *item* argument against the sequence elements. Either the keyword argument **:test** or the keyword argument **:test-not** may be used with **search** to specify a test function other than **eql**.

The keyword **:key** may be used to specify that a part of a sequence element should be tested. The arguments passed to the test function of **search** are extracted from the sequence elements according to the **:key** function. If **:key** is not specified, the elements themselves are used.

Examples:

```
> (search "dog" "it's a dog's life")
7
> (search '(0 1) '(2 4 6 1 3 5) :key #'oddp)
2
```

sort, stable-sort

Purpose: The functions **sort** and **stable-sort** destructively sort their *sequence* arguments according to the order determined by their *predicate* and **:key** arguments and return the new sequence.

The order of the elements in the result sequence is determined by the *predicate* argument. The *predicate* argument must be a function of two elements. It should return a non-**nil** value if the element corresponding to its first argument is to precede the element corresponding to the second in the result sequence; otherwise it should return **nil**.

The arguments passed to *predicate* are extracted from the elements according to the **:key** function. If **:key** is not specified, the elements themselves are used.

Syntax:

sort *sequence predicate* `&key :key` [*Function*]

stable-sort *sequence predicate* `&key :key` [*Function*]

Remarks: The **stable-sort** operation is stable. That is, if two elements are considered equivalent by the *predicate* function, they remain in their original order in the resulting sequence. The sorting operation of **sort** is not stable; it may, however, be faster than **stable-sort**.

Examples:

```
> (sort "lkjashd" #'char-lessp)
"adhjkls"
> (stable-sort '(1 2 3 4 5 6 7 8 9 0)
               #'(lambda (x y) (and (oddp x) (evenp y))))
(1 3 5 7 9 2 4 6 8 0)
```

subseq

Purpose: The function **subseq** creates and returns a sequence that is a copy of the subsequence of *sequence* delimited by *start* and *end*.

Syntax: **subseq** *sequence start* `&optional` *end* [*Function*]

Remarks: The arguments *start* and *end* take integer values that specify offsets into the original sequence. The *start* argument marks the beginning position of the subsequence; the *end* argument marks the position following the last element of the subsequence. The start value defaults to 0; the end value defaults to the length of the sequence.

The macro **setf** may be used with **subseq** to destructively replace a subsequence with a new sequence. If the subsequence and the new sequence are not of equal length, the shorter length determines the number of elements that are replaced. The remaining elements at the end of the longer sequence do not take part in the operation.

Examples:

```
> (setq str "012345")
"012345"
> (subseq str 2)
"2345"
> (subseq str 3 5)
"34"
> (setf (subseq str 4) "abc")
"abc"
> str
"0123ab"
> (setf (subseq str 0 2) "A")
"A"
> str
"A123ab"
```

See Also: **replace**

substitute, substitute-if, substitute-it-not, nsubstitute, nsubstitute-if, nsubstitute-if-not

Purpose: The functions **substitute**, **substitute-if**, and **substitute-if-not** return a modified copy of their *sequence* arguments in which each element that matches a particular item or each element that satisfies some test predicate has been replaced with a new item.

The functions **substitute**, **substitute-if**, and **substitute-if-not** are nondestructive operations. They return a modified copy of their *sequence* argument.

The functions **nsubstitute**, **nsubstitute-if**, and **nsubstitute-if-not** are like **substitute**, **substitute-if**, and **substitute-if-not** respectively, but they may modify their *sequence* argument.

Syntax:

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*]

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*]

Remarks: The result of **substitute**, **substitute-if**, or **substitute-if-not** may share cells with the original sequence if the sequence is a list.

The result of **nsubstitute**, **nsubstitute-if**, or **nsubstitute-if-not** may or may not be **eq** to the original sequence.

If **:count** is specified, only the leftmost number of elements specified by **:count** that satisfy the test condition are replaced.

The **:from-end** argument has an effect only if **:count** is specified and **:from-end** is non-**nil**. In this case only the rightmost number of elements specified by **:count** that satisfy the test condition are replaced.

The keyword arguments **:start** and **:end** take integer values that specify offsets into the original sequence. The **:start** argument marks the beginning position of the subsequence; the **:end** argument marks the position following the last element of the subsequence. The start value defaults to 0; the end value defaults to the length of the sequence.

The function **eql** is the default that is used by **substitute** and **nsubstitute** for matching the *item* argument against the sequence elements. Either the keyword argument **:test** or the keyword argument **:test-not** may be used with **substitute** or **nsubstitute** to specify a test function other than **eql**.

The keyword **:key** may be used to specify that a part of a sequence element should be tested. The arguments passed to the test function of **substitute** and **nsubstitute** or to the test predicate of **substitute-if**, **substitute-if-not**, **nsubstitute-if**, or **nsubstitute-if-not** are extracted from the sequence elements according to the **:key** function. If **:key** is not specified, the elements themselves are used.

Examples:

```
> (substitute #\. #\SPACE "0 2 4 6")
"0.2.4.6"
> (substitute-if 0 #'evenp '((1) (2) (3) (4)) :start 2 :key #'car)
((1) (2) (3) 0)
> (nsubstitute-if "function was here" #'fboundp '(a car b cdr c)
                                       :count 1 :from-end t)
(A CAR B "function was here" C)
```

See Also:

subst

nsubst

Chapter 15. Lists

Chapter 15. Lists

About Lists

The ***list*** data type is the union of the ***cons*** and ***null*** data types and therefore includes both true lists and dotted lists.

Lists are sequences of linked elements, called ***conses*** (dotted pairs). A ***cons*** is an object containing two components, a ***car*** and a ***cdr***, which can be any Lisp objects. Conses in a list are linked by their cdr components. The car components become the elements of the list. A ***true list*** is terminated by **nil**, the empty list. A ***dotted list*** is not terminated by **nil**, but by some non-**nil** data object. The ***tail*** of a list is that portion of the list that remains when any number of elements are removed from the front of the list (as by the **car** operation). The tail of a list is a cons; **nil** is not considered to be a tail of a list.

An ***association list*** is a list whose elements are conses. Each cons is regarded as a pair of associated objects. The car is called the ***key*** and the cdr the ***datum***. An association list can be treated as a mapping from keys to data. New entries are always added to the front of the list, and the list is always searched from the front. Thus it is possible to update the mapping without removing items from the list.

In Common Lisp a list of items can be treated as a set. There are functions for set union, intersection, and difference, and also for adding, removing, and searching for items in a list.

Categories of Operations

This section groups operations on lists according to functionality.

Data Type Predicates

atom	**listp**
consp	**null**

These predicates test for atoms and lists.

Operations on Conses

car	**caaddr**
cdr	**cadaar**
caar	**cadadr**
cadr	**caddar**
cdar	**cadddr**
cddr	**cdaaar**
caaar	**cddadr**
caadr	**cdadar**
cadar	**cdaddr**
caddr	**cddaar**
cdaar	**cddadr**
cdadr	**cdddar**
cddar	**cddddr**
cdddr	**cons**
caaaar	**rplaca**
caaadr	**rplacd**
caadar	

These basic operations on conses access and modify their components and construct new conses.

Basic List Operations

append	**list**
butlast	**list***
nbutlast	**list-length**
copy-list	**list-reverse**
copy-tree	**list-nreverse**
endp	**make-list**
first	**nconc**
second	**nreconc**
third	**nth**
fourth	**nthcdr**
fifth	**pop**
sixth	**push**
seventh	**pushnew**
eighth	**rest**
ninth	**revappend**
tenth	**tailp**
last	**tree-equal**
ldiff	

These operations construct lists, modify lists, access components of lists, and obtain information about various list attributes.

Mapping Operations

mapcar	**mapl**
maplist	**mapcan**
mapc	**mapcon**

These functions are used to perform mapping operations on lists.

Substitution Operations

subst	**nsubst-if**
subst-if	**nsubst-if-not**
subst-if-not	**sublis**
nsubst	**nsublis**

These functions allow for the regular substitution of list elements.

Set Operations

adjoin	**set-difference**
intersection	**nset-difference**
nintersection	**set-exclusive-or**
member	**nset-exclusive-or**
member-if	**subsetp**
member-if-not	**union**
memq	**nunion**

These functions allow lists to be treated as sets. They perform set operations on lists.

Operations on Association Lists

acons	**copy-alist**
assoc	**pairlis**
assoc-if	**rassoc**
assoc-if-not	**rassoc-if**
assq	**rassoc-if-not**

These functions manipulate association lists.

acons

Purpose: The function **acons** is used to add to association lists. It adds the entry (*key* . *datum*) to the front of the association list specified by the *a-list* argument and returns the result.

Syntax: **acons** *key datum a-list* [*Function*]

Examples:

```
> (setq alist ())
NIL
> (acons 1 "one" alist)
((1 . "one"))
> alist
NIL
> (setq alist (acons 1 "one" (acons 2 "two" alist)))
((1 . "one") (2 . "two"))
> (assoc 1 alist)
(1 . "one")
> (setq alist (acons 1 "uno" alist))
((1 . "uno") (1 . "one") (2 . "two"))
> (assoc 1 alist)
(1 . "uno")
```

See Also: **pairlis**

adjoin

Purpose: The function **adjoin** tests whether its *item* argument is the same as an existing element of a list. If the item is not, **adjoin** adds it to the list and returns the resulting list; otherwise nothing is added and the original list is returned.

Syntax: **adjoin** *item list* `&key :test :test-not :key` [*Function*]

Remarks: Whether an item is the same as a list element is determined by the functions specified by the keyword arguments. If a test argument is not specified, **eql** is used. Either the keyword **:test** or the keyword **:test-not** may be used to specify a test function other than **eql**.

The keyword **:key** may be used to specify that a part of an element should be tested. Its argument should be a function of one argument that extracts the part to be tested from both the *item* argument and the list element.

Examples:

```
> (setq slist ())
NIL
> (adjoin 'a slist)
(A)
> slist
NIL
> (setq slist (adjoin '(foo 1) slist))
((FOO 1))
> (adjoin '(foo 1) slist)
((FOO 1) (FOO 1))
> (adjoin '(foo 1) slist :test 'equal)
((FOO 1))
> (adjoin '(bar 1) slist :key #'cadr)
((FOO 1))
> (adjoin '(bar 1) slist)
((BAR 1) (FOO 1))
```

See Also: **pushnew**

append

Purpose: The function **append** creates and returns a list that is the concatenation of its list arguments. The original lists are left unchanged.

Syntax: **append** `&rest` *lists* [*Function*]

Remarks: The last argument to **append** can be any object. If it is not a list, it becomes the cdr of the final dotted pair of the new list.

The function **append** copies the top-level list structure of all its arguments except the last.

Examples:

```
> (append '(a b c) '(d e f) '() '(g))
(A B C D E F G)
> (append '(a b c) 'd)
(A B C . D)
> (setq lst '(a b c))
(A B C)
> (append lst '(d))
(A B C D)
> lst
(A B C)
> (append)
NIL
```

See Also: **nconc**

concatenate

assoc, assoc-if, assoc-if-not

Purpose: The functions **assoc**, **assoc-if**, and **assoc-if-not** search association lists. They return the first pair in the association list whose car is the same as a given item or satisfies the test condition or predicate. If no such entry is found, **nil** is returned.

Syntax:

assoc *item a-list* `&key :test :test-not :key` [*Function*]

assoc-if *predicate a-list* [*Function*]

assoc-if-not *predicate a-list* [*Function*]

Remarks: If **nil** appears in an association list in place of a pair, it is ignored.

If a test argument for **assoc** is not specified, **eql** is used. Either the keyword **:test** or the keyword **:test-not** may be used to specify a test function other than **eql**.

The keyword **:key** may be used to specify that a part of an element should be tested. Its argument should be a function of one argument that extracts the part to be tested from the car of the association list entry.

Examples:

```
> (setq alist '((1 . "one")(2 . "two")(3 . "three")))
((1 . "one") (2 . "two") (3 . "three"))
> (assoc 2 alist)
(2 . "two")
> (assoc-if #'evenp alist)
(2 . "two")
> (assoc-if-not #'(lambda(x)(< x 3)) alist)
(3 . "three")
> (setq alist '(("one" . 1)("two" . 2)))
(("one" . 1) ("two" . 2))
> (assoc "one" alist)
NIL
> (assoc "one" alist :test #'equalp)
("one" . 1)
> (assoc "two" alist :key #'(lambda(x)(char x 2)))
NIL
> (assoc #\o alist :key #'(lambda(x)(char x 2)))
("two" . 2)
```

See Also:

rassoc

rassoc-if

rassoc-if-not

assq

assq

Purpose: The function **assq** searches an association list for the first pair whose car is **eq** to its *object* argument. It returns the first such pair that it finds; if no such entry is found, it returns **nil**.

Syntax: **assq** *object a-list* [*Function*]

Remarks: The function **assq** passes over any element in the list that is not a dotted pair without producing an error.

The function **assq** is an extension to Common Lisp.

Examples:

```
> (setq alist '(("one" . 1) (2 . "two") 3 ((4) . "four")))
(("one" . 1) (2 . "two") 3 ((4) . "four"))
> (assq "one" alist)
NIL
> (assq 2 alist)
(2 . "two")
> (assq 3 alist)
NIL
> (assq '(4) alist)
NIL
> (assq nil alist)
NIL
```

See Also: **assoc**

assoc-if

assoc-if-not

atom

Purpose: The predicate **atom** is true if its argument is not a cons; otherwise it is false.

Syntax: **atom** *object* [*Function*]

Examples:

```
> (atom 'sss)
T
> (atom (cons 1 2))
NIL
> (atom nil)
T
```

butlast, nbutlast

Purpose: The function **butlast** creates and returns a copy of its *list* argument from which the last *n* elements have been omitted. If there are fewer than *n* elements in the original list, **nil** is returned. If *n* is not specified, the last element is omitted from the list.

The function **nbutlast** is like **butlast**, but **nbutlast** may modify its *list* argument.

Syntax:

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

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

Remarks: If **nbutlast** is given a *list* argument of fewer than *n* elements, it returns **nil** without modifying the argument.

Examples:

```
> (setq lst '(1 2 3 4 5 6 7 8 9))
(1 2 3 4 5 6 7 8 9)
> (butlast lst)
(1 2 3 4 5 6 7 8)
> (butlast lst 5)
(1 2 3 4)
> (butlast lst (+ 5 5))
NIL
> lst
(1 2 3 4 5 6 7 8 9)
> (nbutlast lst 3)
(1 2 3 4 5 6)
> lst
(1 2 3 4 5 6)
> (nbutlast lst 99)
NIL
> lst
(1 2 3 4 5 6)
> (butlast '(1 2 . 3))
(1)
```

car, cdr

Purpose: The function **car** returns the car of a list. If the list is a cons, **car** returns the first element of the list. If the list is **nil**, **car** returns **nil**.

The function **cdr** returns the cdr of a list. If the list is a cons, **cdr** returns the portion that follows the first element. If the list is **nil**, **cdr** returns **nil**.

Compositions of up to four **car** and **cdr** operations are also defined as functions. The names of these functions consist of **c**, followed by two, three, or four occurrences of **a** or **d**, and then **r**. The sequence of **a**'s and **d**'s in the name specify the sequence of **car** and **cdr** operations that is performed by the function. The order in which the **a**'s and **d**'s appear is the inverse of the order in which the corresponding operations are performed. For example, the expression `(cadddr x)` is the same as `(car (cdr (cdr (cdr x))))`.

Syntax:

car *list* [*Function*]

cdr *list* [*Function*]

caar *list* [*Function*]

cadr *list* [*Function*]

cdar *list* [*Function*]

cddr *list* [*Function*]

caaar *list* [*Function*]

caadr *list* [*Function*]

cadar *list* [*Function*]

caddr *list* [*Function*]

cdaar *list* [*Function*]

cdadr *list* [*Function*]

cddar *list* [*Function*]

cdddr *list* [*Function*]

caaaar *list* [*Function*]

caaadr *list* [*Function*]

caadar *list* [*Function*]

caaddr *list* [*Function*]

cadaar *list* [*Function*]

cadadr *list* [*Function*]

caddar *list* [*Function*]

cadddr *list* [*Function*]

cdaaar *list* [*Function*]

cdaadr *list* [*Function*]

cdadar *list* [*Function*]

cdaddr *list* [*Function*]

cddaar *list* [*Function*]

cddadr *list* [*Function*]

cdddar *list* [*Function*]

cddddr *list* [*Function*]

Remarks: The macro **setf** may be used with any of these functions to change an element of a list.

Examples:

```
> (car nil)
NIL
> (cdr '(1 . 2))
2
> (cdr '(1 2))
(2)
> (cadr '(1 2))
2
```

See Also: **rplaca**

rplacd

cons

Purpose: The function **cons** creates and returns a new cons cell whose car is *object1* and whose cdr is *object2*.

The function **cons** is designed for use in constructing lists.

Syntax: **cons** *object1 object2* [*Function*]

Examples:

```
> (cons 1 2)
(1 . 2)
> (cons 1 nil)
(1)
> (cons nil 2)
(NIL . 2)
> (cons nil nil)
(NIL)
> (cons 1 (cons 2 (cons 3 (cons 4 nil))))
(1 2 3 4)
> (cons 'a '(b c d e))
(A B C D E)
```

See Also: **list**

consp

Purpose: The predicate **consp** is true if its argument is a cons; otherwise it is false.

Syntax: **consp** *object* [*Function*]

Examples:

```
> (consp nil)
NIL
> (consp (cons 1 2))
T
```

See Also: **listp**

copy-alist

Purpose: The function **copy-alist** is used to copy association lists. It returns a copy of its *list* argument.

Syntax: **copy-alist** *list* [*Function*]

Remarks: The top level of the list structure is copied, and new copies are made of each list element that is a cons. The rest of the list structure is shared.

Examples:

```
> (setq alist '((1 . "one") (2 . "two")))
((1 . "one") (2 . "two"))
> (setq clist (copy-list alist))
((1 . "one") (2 . "two"))
> (setq calist (copy-alist alist))
((1 . "one") (2 . "two"))
> (setf (cdr (assoc 2 calist)) "deux")
"deux"
> calist
((1 . "one") (2 . "deux"))
> alist
((1 . "one") (2 . "two"))
> (setf (cdr (assoc 1 clist)) "uno")
"uno"
> clist
((1 . "uno") (2 . "two"))
> alist
((1 . "uno") (2 . "two"))
```

See Also: **copy-list**

copy-list

Purpose: The function **copy-list** returns a copy of its *list* argument. The copy is **equal** to the list argument, but not **eq**.

Syntax: **copy-list** *list* [*Function*]

Remarks: Only the top level of the list structure is copied; the rest of the list structure is shared.

Examples:

```
> (setq lst '(1 (2 3)))
(1 (2 3))
> (setq slst lst)
(1 (2 3))
> (setq clst (copy-list lst))
(1 (2 3))
> (eq slst lst)
T
> (eq clst lst)
NIL
> (equal clst lst)
T
> (rplaca lst "one")
("one" (2 3))
> slst
("one" (2 3))
> clst
(1 (2 3))
> (setf (caadr lst) "two")
"two"
> lst
("one" ("two" 3))
> slst
("one" ("two" 3))
> clst
(1 ("two" 3))
```

See Also: **copy-alist**

copy-seq

copy-tree

copy-tree

Purpose: The function **copy-tree** is used to copy trees of conses.

If the argument of **copy-tree** is not a cons, it is returned. If it is a cons, **copy-tree** returns a new cons whose car and cdr consist of the result of calling **copy-tree** on the car and cdr of the argument cons respectively. The recursion stops only when an object that is not a cons is reached.

Syntax: **copy-tree** *object* [*Function*]

Examples:

```
> (setq lst '((1 . "one") (2 . (a b c))))
((1 . "one") (2 A B C))
> (setq slst lst
        clst (copy-list lst)
        calst (copy-alist lst)
        ctlst (copy-tree lst))
((1 . "one") (2 A B C))
> (eq ctlst lst)
NIL
> (eql ctlst lst)
NIL
> (equal ctlst lst)
T
> (setf (cadadr lst) "a"
        (caadr lst) "two"
        (car lst) "1 . one")
"1 . one"
> lst
("1 . one" ("two" "a" B C))
> slst
("1 . one" ("two" "a" B C))
> clst
((1 . "one") ("two" "a" B C))
> calst
((1 . "one") (2 "a" B C))
> ctlst
((1 . "one") (2 A B C))
```

endp

Purpose: The predicate **endp** tests for the end of a list. The predicate **endp** is true if its argument is **nil**; it is false if the argument is a cons.

Syntax: **endp** *list* [*Function*]

Examples:

```
> (endp nil)
T
> (endp '(1 2))
NIL
> (endp (cddr '(1 2)))
T
```

first, second, third, fourth, fifth, sixth, seventh, eighth, ninth, tenth

Purpose: These functions may be used to access particular elements of a list. The function **first** is identical to **car**, **second** to **cadr**, and so on.

Syntax:

first *list* [*Function*]

second *list* [*Function*]

third *list* [*Function*]

fourth *list* [*Function*]

fifth *list* [*Function*]

sixth *list* [*Function*]

seventh *list* [*Function*]

eighth *list* [*Function*]

ninth *list* [*Function*]

tenth *list* [*Function*]

Remarks: These functions were designed for use in list operations for stylistic reasons.

Examples:

```
> (setq lst '(1 2 3 (4 5 6) ((V)) vi 7 8 9 10))
(1 2 3 (4 5 6) ((V)) VI 7 8 9 10)
> (first lst)
1
> (tenth lst)
10
> (fifth lst)
((V))
> (second (fourth lst))
5
> (sixth '(1 2 3))
NIL
> (setf (fourth lst) "four")
"four"
> lst
(1 2 3 "four" ((V)) VI 7 8 9 10)
```

See Also: **car**

nth

intersection, nintersection

Purpose: The functions **intersection** and **nintersection** take two lists and return a list that contains every element that occurs in both of the list arguments.

The result list of **intersection** may share cells with one of the list arguments. The function **nintersection** may modify its list arguments.

Syntax: **intersection** *list1 list2* `&key :test :test-not :key` [*Function*]

nintersection *list1 list2* `&key :test :test-not :key` [*Function*]

Remarks: If one of the lists contains duplicate elements, there may be duplication in the result.

Whether a list element is the same as another list element is determined by the functions specified by the keyword arguments. If a test argument is not specified, **eql** is used. Either the keyword **:test** or the keyword **:test-not** may be used to specify a test function other than **eql**.

The keyword **:key** may be used to specify that a part of a list element should be tested. Its argument should be a function of one argument that extracts the part to be tested from the list element.

Examples:

```
> (setq lst1 '(1 1 2 3 4 a b c "A" "B" "C" "d")
        lst2 '(1 4 5 b c d "a" "B" "c" "D"))
(1 4 5 B C D "a" "B" "c" "D")
> (intersection lst1 lst2)
(C B 4 1 1)
> (intersection lst1 lst2 :test 'equal)
("B" C B 4 1 1)
> (intersection lst1 lst2 :test #'equalp)
("d" "C" "B" "A" C B 4 1 1)
> (nintersection lst1 lst2)
(1 1 4 B C)
> lst1
(1 1 4 B C)
> lst2
(1 4 5 B C D "a" "B" "c" "D")
```

last

Purpose: The function **last** returns the last cons of its *list* argument. If the list is empty, it returns **nil**.

Syntax: **last** *list* [*Function*]

Examples:

```
> (last nil)
NIL
> (last '(1 2 3))
(3)
> (last '(1 2 . 3))
(2 . 3)
```

ldiff

Purpose: The function **ldiff** tests whether its *sublist* argument forms a tail of the given *list*. If **ldiff** succeeds, it returns a new list that is a copy of the portion of the original list that precedes the sublist. If the sublist is **nil** or is not a tail of the original list, **ldiff** returns a copy of the original list.

Syntax: **ldiff** *list sublist* [*Function*]

Examples:

```
> (setq x '(a b c d e))
(A B C D E)
> (setq y (cddr x))
(C D E)
> (setq z '(c d e))
(C D E)
> (ldiff x x)
NIL
> (ldiff x y)
(A B)
> (ldiff x z)
(A B C D E)
> (eq y z)
NIL
> (eq x (ldiff x z))
NIL
```

See Also: **tailp**

list, list*

Purpose: The function **list** creates and returns a list containing the specified objects.

The function **list*** is like **list** except that its last argument becomes the cdr of the last cons in the resulting list. If **list*** is called with exactly one argument, it returns that argument, not a cons.

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

list* *object* `&rest` *more-objects* [*Function*]

Examples:

```
> (list 1)
(1)
> (list* 1)
1
> (setq a 1)
1
> (list a 2)
(1 2)
> '(a 2)
(A 2)
> (list 'a 2)
(A 2)
> (list* a 2)
(1 . 2)
> (list)
NIL
> (setq a '(1 2))
(1 2)
> (eq a (list* a))
T
```

See Also: **cons**

list-length

Purpose: The function **list-length** returns the length of its *list* argument as an integer value. If the list is circular, **list-length** returns **nil**.

Syntax: **list-length** *list* [*Function*]

Examples:

```
> (list-length '(1 2 3))
3
> (list-length nil)
0
> (setq lst '(1 2))
(1 2)
> (list-length (rplacd lst lst))
NIL
```

See Also: **length**

list-reverse, list-nreverse

Purpose: The function **list-reverse** returns a true list consisting of the elements of the original list in reverse order (but omitting the last cdr, which is **nil** for true lists).

The function **list-nreverse** is like **list-reverse** but modifies its argument.

Syntax: **list-reverse** *list* [*Function*]

list-nreverse *list* [*Function*]

Remarks: The functions **list-reverse** and **list-nreverse** are extensions to Common Lisp.

Examples:

```
> (setq lst '(1 2 3))
(1 2 3)
> (list-reverse lst)
(3 2 1)
> lst
(1 2 3)
> (list-nreverse lst)
(3 2 1)
> lst
(1)
> (list-reverse '(1 2 . 3))
(2 1)
```

See Also: **reverse**

listp

Purpose: The predicate **listp** is true if its argument is a cons or the empty list, (); otherwise it is false.

Syntax: **listp** *object* [*Function*]

Examples:

```
> (listp nil)
T
> (listp (cons 1 2))
T
> (listp (make-array 6))
NIL
> (listp t)
NIL
```

See Also: **consp**

make-list

Purpose: The function **make-list** creates and returns a list consisting of *size* elements. If the **:initial-element** argument is specified, each of the elements of the new list is initialized to its value; otherwise the elements are **nil**.

Syntax: **make-list** *size* `&key :initial-element` [*Function*]

Examples:

```
> (make-list 4)
(NIL NIL NIL NIL)
> (make-list 2 :initial-element '(1 2 3))
((1 2 3) (1 2 3))
> (make-list 0)
NIL
> (make-list 0 :initial-element 'foo)
NIL
```

mapcar, maplist, mapc, mapl, mapcan, mapcon

Purpose: The functions **mapcar**, **maplist**, **mapc**, **mapl**, **mapcan**, and **mapcon** are used to map over lists.

The mapping operation involves applying a function to successive sets of arguments in which one argument is obtained from each list.

The *function* argument must take as many arguments as there are list arguments. The resulting list is the same length as the shortest of the list arguments. It contains the results returned by the function.

The mapping functions differ in how they obtain their arguments and present their results.

The function **mapcar** applies its *function* argument to successive elements of the list arguments. The function is applied to the first element of each list, then to the second, and so on. A list consisting of the results of applying the function is returned as the result of **mapcar**.

The function **mapc** is like **mapcar** except that the results of applying the function are not returned. The function is applied for its side effects only. The function **mapc** returns its first *list* argument as its result.

The function **maplist** is like **mapcar** except that the *function* argument is applied to successive sublists of the list arguments. First, the function is applied to the lists themselves, then to the cdr of each list, then to the cddr of each list, and so on. A list consisting of the results of applying the function is returned as the result of **maplist**.

The function **mapl** is like **maplist** except that the results of applying the function are not returned. The function is applied for its side effects only. The function **maplist** returns its first *list* argument as its result.

The functions **mapcan** and **mapcon** are like **mapcar** and **maplist** respectively, except that the results of applying the function are combined into a list by the use of **nconc** rather than **list**.

Syntax:

mapcar *function list* **&rest** *more-lists* [*Function*]

mapc *function list* **&rest** *more-lists* [*Function*]

maplist *function list* **&rest** *more-lists* [*Function*]

mapl *function list* **&rest** *more-lists* [*Function*]

mapcan *function list* **&rest** *more-lists* [*Function*]

mapcon *function list* **&rest** *more-lists* [*Function*]

mapcar, maplist, mapc, mapl, mapcan, mapcon

Remarks: The *function* argument must be a function acceptable to **apply**. It cannot be a macro or a special form.

Examples:

```
> (mapcar #' car '((1 a) (2 b) (3 c)))
(1 2 3)
> (maplist #'append '(1 2 3 4) '(1 2) '(1 2 3))
((1 2 3 4 1 2 1 2 3) (2 3 4 2 2 3))
> (setq foo nil)
NIL
> (mapc #'(lambda (&rest x) (setq foo (append foo x)))
        '(1 2 3 4)
        '(a b c d e)
        '(x y. z))
(1 2 3 4)
> foo
(1 A X 2 B Y 3 C Z)
> (setq foo nil)
NIL
> (mapl #'(lambda (x) (push x foo)) '(1 2 3 4))
(1 2 3 4)
> foo
((4) (3 4) (2 3 4) (1 2 3 4))
> (mapcan #'(lambda (x y) (if (null x) nil (list x y)))
          '(nil nil nil d e)
          '(1 2 3 4 5 6))
(D 4 E 5)
> (mapcon #'list '(1 2 3 4))
((1 2 3 4) (2 3 4) (3 4) (4))
```

See Also:

map

apply

nconc

member, member-if, member-if-not

Purpose: The functions **member**, **member-if**, and **member-if-not** each search a list for a particular item or for a top-level element that satisfies some test condition or predicate. If they succeed, the tail of the list beginning with this element is returned; otherwise **nil** is returned.

Syntax:

member *item list* `&key :test :test-not :key` [*Function*]

member-if *predicate list* `&key :key` [*Function*]

member-if-not *predicate list* `&key :key` [*Function*]

Remarks: If a test argument for **member** is not specified, **eql** is used. Either the keyword **:test** or the keyword **:test-not** may be used to specify a test function other than **eql**.

The keyword **:key** may be used to specify that a part of a list element should be tested. Its argument should be a function of one argument that extracts the part to be tested from the list element.

Examples:

```
> (member 2 '(1 2 3))
(2 3)
> (member 4 '(1 2 3))
NIL
> (member 2 '((1 . 2) (3 . 4)) :test-not #'= :key #'cdr)
((3 . 4))
> (member-if #'listp '(a b nil c d))
(NIL C D)
> (member-if-not #'zerop
                 '(3 6 9 11 . 12)
                 :key #'(lambda (x) (mod x 3)))
(11 . 12)
```

See Also:

find

position

memq

memq

Purpose: The function **memq** searches a list for the first top-level element that is **eq** to a given object. If it succeeds, it returns the tail of the list starting with that element. If no such element is found, it returns **nil**.

Syntax: **memq** *object list* [*Function*]

Remarks: The function **memq** is an extension to Common Lisp.

Examples:

```
> (memq 1 '(a 1 "b"))
(1 "b")
> (memq 'a '(a 1 "b"))
(A 1 "b")
> (memq "b" '(a 1 "b"))
NIL
> (memq 2 '(1 2 . 3))
(2 . 3)
> (memq 3 '(1 2 . 3))
NIL
```

See Also:

member

member-if

member-if-not

nconc

Purpose: The function **nconc** returns a list that is the concatenation of its list arguments. The list arguments are modified rather than copied.

Syntax: **nconc** `&rest` *lists* [*Function*]

Remarks: The function **nconc** is designed to be more efficient than **append** since no new cons cells need to be allocated.

Examples:

```
> (setq lst1 '(1 2 3)
        lst2 '(a b c))
(A B C)
> (nconc lst1 lst2)
(1 2 3 A B C)
> lst1
(1 2 3 A B C)
> lst2
(A B C)
> (nconc)
NIL
```

See Also: **append**

nreconc

Purpose: The function **nreconc** reverses the order of the elements in its first list argument and appends the second list argument to the modified list. The resulting list is returned.

Syntax: **nreconc** *list1 list2* [*Function*]

Remarks: The *list1* argument is modified.

The function **nreconc** is designed to be more efficient than **revappend** since no new cons cells need to be allocated.

Examples:

```
> (setq lst1 '(1 2 3)
        lst2 '(a b c))
(A B C)
> (nreconc lst1 lst2)
(3 2 1 A B C)
> lst1
(1 A B C)
> lst2
(A B C)
> (nreconc '(1 . 2) nil)
(1)
```

See Also: **revappend**

nth

Purpose: The function **nth** returns the *n*th element of its *list* argument. The initial element of the list is considered to be the zeroth element. If the list has no such element, **nth** returns **nil**.

Syntax: **nth** *n list* [*Function*]

Remarks: The argument *n* must be a nonnegative integer.

The macro **setf** may be used with **nth** to change an element of a list. In this case, *n* must be less than the length of the list.

Examples:

```
> (nth 4 '(0 1 2 3 4))
4
> (nth 5 '(0 1 2 3 4))
NIL
> (nth 0 nil)
NIL
> (setq lst '(0 1 2 3))
(0 1 2 3)
> (setf (nth 2 lst) "two")
"two"
> lst
(0 1 "two" 3)
```

See Also: **elt**

nthcdr

Purpose: The function **nthcdr** performs the **cdr** operation *n* times on its *list* argument and returns the result.

Syntax: **nthcdr** *n list* [*Function*]

Examples:

```
> (nthcdr 0 '(0 1 2 3))
(0 1 2 3)
> (nthcdr 5 '(0 1 2 3))
NIL
> (nthcdr 0 nil)
NIL
> (nthcdr 3 '(0 1 2 3))
(3)
> (nthcdr 1 '(0 . 1))
1
```

null

Purpose: The predicate **null** is true if its argument is the empty list, which is represented by () or **nil**; otherwise it is false.

Syntax: **null** *object* [*Function*]

Remarks: The result of applying **null** to an object is the same as that of using **not**. The function **null** is intended to be used in testing for an empty list, whereas **not** is intended to be used in inverting a logical value.

Examples:

```
> (null ())
T
> (null t)
NIL
> (null 1)
NIL
```

See Also: **not**

pairlis

Purpose: The function **pairlis** takes the two lists, *keys* and *data*, and creates an association list by pairing the corresponding elements of each. If the *a-list* argument is specified, the new pairs of elements are added to the front of the given association list.

Syntax: **pairlis** *keys data* `&optional` *a-list* [*Function*]

Remarks: The *keys* and *data* lists must be the same length.

Examples:

```
> (setq keys '(1 2 3)
        data '("one" "two" "three")
        alist '((4 . "four")))
((4 . "four"))
> (pairlis keys data)
((1 . "one") (2 . "two") (3 . "three"))
> (pairlis keys data alist)
((1 . "one") (2 . "two") (3 . "three") (4 . "four"))
> alist
((4 . "four"))
```

See Also: **acons**

pop

Purpose: The macro pop takes the list that is stored in *place* and returns the car of this list as its result. The cdr of the list is stored in *place*.

Syntax: pop *place* [*Macro*]

Remarks: The *place* argument must be a generalized variable acceptable to the macro **setf**.

Examples:

```
> (setq llst '((1 2 3 4)))
((1 2 3 4))
> (pop (car llst))
1
> llst
((2 3 4))
```

See Also: **push**

push

Purpose: The macro **push** takes the list that is stored in *place*, conses the *item* argument onto the front of it, stores the resulting list in *place*, and returns this new list as its result.

Syntax: **push** *item place* [*Macro*]

Remarks: The *place* argument must be a generalized variable acceptable to the macro **setf**. The *item* argument can be any object.

Examples:

```
> (setq llst '(nil))
(NIL)
> (push 1 (car llst))
(1)
> llst
((1))
> (push 1 (car llst))
(1 1)
> llst
((1 1))
```

See Also: **pop**

pushnew

pushnew

Purpose: The macro **pushnew** tests whether its *item* argument is the same as any existing element of the list stored in *place*. If the item is not, the new item is consed onto the front of the list, and the new list is stored in *place* and returned. If such an element is found in the list, **pushnew** returns the original list.

Syntax: **pushnew** *item place* `&key :test :test-not :key` [*Macro*]

Remarks: The *place* argument must be a generalized variable acceptable to the macro **setf**. The *item* argument can be any object.

Whether an item is the same as a list element is determined by the functions specified by the keyword arguments. If a test argument is not specified, **eql** is used. Either the keyword **:test** or the keyword **:test-not** may be used to specify a test function other than **eql**.

The keyword **:key** may be used to specify that a part of an element should be tested. Its argument should be a function of one argument that extracts the part to be tested from both the *item* argument and the list element.

Examples:

```
> (setq lst '((1) (1 2) (1 2 3)))
((1) (1 2) (1 2 3))
> (pushnew '(2) lst)
((2) (1) (1 2) (1 2 3))
> (pushnew '(1) lst)
((1) (2) (1) (1 2) (1 2 3))
> (pushnew '(1) lst :test 'equal)
((1) (2) (1) (1 2) (1 2 3))
> (pushnew '(1) lst :key #'car)
((1) (2) (1) (1 2) (1 2 3))
```

See Also: **push**

adjoin

rassoc, rassoc-if, rassoc-if-not

Purpose: The functions **rassoc**, **rassoc-if**, and **rassoc-if-not** search association lists. They return the first pair whose cdr is the same as a given item or satisfies the test condition or predicate. If no such pair is found, **nil** is returned.

Syntax:

rassoc *item a-list* `&key :test :test-not :key` [*Function*]

rassoc-if *predicate a-list* [*Function*]

rassoc-if-not *predicate a-list* [*Function*]

Remarks: If **nil** appears in an association list in place of a pair, it is ignored.

If a test argument for **rassoc** is not specified, **eql** is used. Either the keyword **:test** or the keyword **:test-not** may be used to specify a test function other than **eql**.

The keyword **:key** may be used to specify that a part of an element should be tested. Its argument should be a function of one argument that extracts the part to be tested from the cdr of the association list entry.

Examples:

```
> (setq alist '((1 . "one") (2 . "two") (3 . 3)))
((1 . "one") (2 . "two") (3 . 3))
> (rassoc 3 alist)
(3 . 3)
> (rassoc "two" alist)
NIL
> (rassoc "two" alist :test 'equal)
(2 . "two")
> (rassoc 1 alist :key #'(lambda (x) (if (numberp x) (/ x 3))))
(3 . 3)
> (rassoc-if #'stringp alist)
(1 . "one")
> (rassoc-if-not #'vectorp alist)
(3 . 3)
```

See Also:

assoc

assoc-if

assoc-if-not

rest

Purpose: The function **rest** is identical to **cdr**. It returns the cdr of a list. If the list is a cons, **rest** returns the portion that follows the first element. If the list is **nil**, **rest** returns **nil**.

Syntax: **rest** *list* [*Function*]

Remarks: The function **rest** was designed for use in list operations for stylistic reasons.

The macro **setf** may be used with **rest** to change the cdr of a list.

Examples:

```
> (rest '(1 2))
(2)
> (rest '(1 . 2))
2
> (rest '(1))
NIL
> (setq cns '(1 . 2))
(1 . 2)
> (setf (rest cns) "two")
"two"
> cns
(1 . "two")
```

See Also: **cdr**

revappend

Purpose: The function **revappend** makes a copy of its first list argument; in this copy the order of elements is reversed. It appends its second list argument to that copy and returns the resulting list.

Syntax: **revappend** *list1 list2* [*Function*]

Examples:

```
> (setq lst1 '(1 2 3)
        lst2 '(a b c))
(A B C)
> (revappend lst1 lst2)
(3 2 1 A B C)
> lst1
(1 2 3)
> lst2
(A B C)
> (revappend '(1 . 2) '(a b c))
(1 A B C)
> (revappend '(1 2 3) '(a . b))
(3 2 1 A . B)
> (revappend nil '(a b c))
(A B C)
```

See Also: **nreconc**

rplaca, rplacd

Purpose: The function **rplaca** replaces the car of its *cons* argument with the specified object and returns the modified cons.

The function **rplacd** replaces the cdr of its *cons* argument with the specified object and returns the modified cons.

Syntax: **rplaca** *cons object* [*Function*]

rplacd *cons object* [*Function*]

Examples:

```
> (setq lst '(a b c))
(A B C)
> (rplaca lst "A")
("A" B C)
> lst
("A" B C)
> (rplacd '(1 2 3 . 4) lst)
(1 "A" B C)
> (rplaca '(1 . 2) lst)
(("A" B C) . 2)
```

set-difference, nset-difference

Purpose: The functions **set-difference** and **nset-difference** take two lists and return a list that contains every element that occurs in the first list argument but not in the second.

The resulting list of **set-difference** may share cells with one of the list arguments. The function **nset-difference** may modify its list arguments.

Syntax: **set-difference** *list1 list2* `&key :test :test-not :key` [*Function*]

nset-difference *list1 list2* `&key :test :test-not :key` [*Function*]

Remarks: Whether a list element is the same as another list element is determined by the functions specified by the keyword arguments. If a test argument is not specified, **eql** is used. Either the keyword **:test** or the keyword **:test-not** may be used to specify a test function other than **eql**.

The keyword **:key** may be used to specify that a part of a list element should be tested. Its argument should be a function of one argument that extracts the part to be tested from the list element.

Examples:

```
> (setq lst1 '("A" "b" "C" "d")
        lst2 '("a" "B" "C" "d"))
("a" "B" "C" "d")
> (set-difference lst1 lst2)
("d" "C" "b" "A")
> (set-difference lst1 lst2 :test 'equal)
("b" "A")
> (set-difference lst1 lst2 :test #'equalp)
NIL
> (nset-difference lst1 lst2 :test #'string=)
("A" "b")
```

set-exclusive-or, nset-exclusive-or

Purpose: The functions **set-exclusive-or** and **nset-exclusive-or** take two lists and return a list that contains every element that occurs in exactly one of the list arguments.

The result list of **set-exclusive-or** may share cells with one of the list arguments. The function **nset-exclusive-or** may modify its list arguments.

Syntax: **set-exclusive-or** *list1 list2* `&key :test :test-not :key` [*Function*]

nset-exclusive-or *list1 list2* `&key :test :test-not :key` [*Function*]

Remarks: Whether a list element is the same as another list element is determined by the functions specified by the keyword arguments. If a test argument is not specified, **eql** is used. Either the keyword **:test** or the keyword **:test-not** may be used to specify a test function other than **eql**.

The keyword **:key** may be used to specify that a part of a list element should be tested. Its argument should be a function of one argument that extracts the part to be tested from the list element.

Examples:

```
> (setq lst1 '(1 "a" "b")
        lst2 '(1 "A" "b"))
(1 "A" "b")
> (set-exclusive-or lst1 lst2)
("b" "A" "b" "a")
> (set-exclusive-or lst1 lst2 :test #'equal)
("A" "a")
> (set-exclusive-or lst1 lst2 :test 'equalp)
NIL
> (nset-exclusive-or lst1 lst2)
("a" "b" "A" "b")
```

sublis, nsublis

Purpose: The function **sublis** performs substitution operations on trees. It searches a tree for subtrees or leaves that occur as keys in the association list argument *a-list*. If the function succeeds, a new copy of the tree is returned in which each occurrence of such a subtree or leaf is replaced by the object with which it is associated. If no changes are made, the original tree is returned. The original *tree* argument is left unchanged, but the result tree may share cells with it.

The function **nsublis** is like **sublis**, but **nsublis** modifies its *tree* argument.

Syntax:

sublis *a-list tree* `&key :test :test-not :key` [*Function*]

nsublis *a-list tree* `&key :test :test-not :key` [*Function*]

Remarks: If a test argument is not specified, **eql** is used. Either the keyword **:test** or the keyword **:test-not** may be used to specify a test function other than **eql**.

The keyword **:key** may be used to specify that a part of an element should be tested. Its argument should be a function of one argument that extracts the part to be tested from the tree element.

Examples:

```
> (setq tree1 '(1 (1 2) ((1 2 3)) (((1 2 3 4)))))
(1 (1 2) ((1 2 3)) (((1 2 3 4))))
> (sublis '((3 . "three")) tree1)
(1 (1 2) ((1 2 "three")) (((1 2 "three" 4))))
> (sublis '((t . "string"))
          (sublis '((1 . "") (4 . 44)) tree1)
          :key #'stringp)
("string" ("string" 2) (("string" 2 3)) ((("string" 2 3 44))))
> tree1
(1 (1 2) ((1 2 3)) (((1 2 3 4))))
> (nsublis '((t . 'foo))
           tree1
           :key #'(lambda (x) (or (atom x) (< (list-length x) 3))))
((QUOTE FOO) (QUOTE FOO) QUOTE FOO)
```

subsetp

Purpose: The predicate **subsetp** is true if every element of the first list is the same as some element of the second list; otherwise it is false.

Syntax: **subsetp** *list1 list2* `&key :test :test-not :key` [*Function*]

Remarks: Whether a list element is the same as another list element is determined by the functions specified by the keyword arguments. If a test argument is not specified, **eql** is used. Either the keyword **:test** or the keyword **:test-not** may be used to specify a test function other than **eql**.

The keyword **:key** may be used to specify that a part of a list element should be tested. Its argument should be a function of one argument that extracts the part to be tested from the element.

Examples:

```
> (setq cosmos '(1 "a" (1 2)))
(1 "a" (1 2))
> (subsetp '(1) cosmos)
T
> (subsetp '((1 2)) cosmos)
NIL
> (subsetp '((1 2)) cosmos :test 'equal)
T
> (subsetp '(1 "A") cosmos :test #'equalp)
T
> (subsetp '((1) (2)) '((1) (2)))
NIL
> (subsetp '((1) (2)) '((1) (2)) :key #'car)
T
```

subst, subst-if, subst-if-not, nsubst, nsubst-if, nsubst-if-not

Purpose: The functions **subst**, **subst-if**, and **subst-if-not** perform substitution operations upon trees. Each searches a tree for occurrences of a particular *old* item or of an element or subexpression that satisfies some test condition or predicate. If the functions succeed, a new copy of the tree is returned in which each occurrence of such an element is replaced by the *new* element or subexpression. If no changes are made, the original tree may be returned. The original *tree* argument is left unchanged, but the result tree may share cells with it.

The functions **nsubst**, **nsubst-if**, and **nsubst-if-not** are like **subst**, **subst-if**, and **subst-if-not** respectively, except that the original tree is modified and returned as the function result.

Syntax:

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]*

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]*

Remarks: If a test argument for **subst** or **nsubst** is not specified, **eql** is used. Either the keyword **:test** or the keyword **:test-not** may be used to specify a test function other than **eql**.

The keyword **:key** may be used to specify that a part of an element should be tested. Its argument should be a function of one argument that extracts the part to be tested from the tree element.

Examples:

```
> (setq tree1 '(1 (1 2) (1 2 3) (1 2 3 4)))
(1 (1 2) (1 2 3) (1 2 3 4))
> (subst "two" 2 tree1)
(1 (1 "two") (1 "two" 3) (1 "two" 3 4))
> (subst "five" 5 tree1)
(1 (1 2) (1 2 3) (1 2 3 4))
> (eq tree1 (subst "five" 5 tree1))
T
> (subst-if 5 #'listp tree1)
5
> (subst-if-not '(x) #'consp tree1)
((X) ((X) (X) X) ((X) (X) (X) X) ((X) (X) (X) (X) X) X)
```

```
> tree1
(1 (1 2) (1 2 3) (1 2 3 4))
> (nsubst 'x 3 tree1 :key #'(lambda(y) (and (listp y) (third y))))
(1 (1 2) X X)
> tree1
(1 (1 2) X X)
```

See Also: **substitute**

nsubstitute

tailp

Purpose: The predicate **tailp** tests whether its *sublist* argument forms a tail of the given *list*. If it does, **tailp** returns true; otherwise it returns false.

Syntax: **tailp** *sublist list* [*Function*]

Examples:

```
> (tailp '(2 3) '(1 2 3))
NIL
> (tailp (cdr '(1 2 3)) '(1 2 3))
NIL
> (setq lst '(1 2 3))
(1 2 3)
> (setq tlst (cdr lst))
(2 3)
> (tailp tlst lst)
T
> (tailp nil '(1 2 3))
NIL
```

See Also: **ldiff**

tree-equal

Purpose: The predicate **tree-equal** tests whether two trees are of the same shape and have the same leaves. It returns **t** if *object1* and *object2* are both atoms and satisfy the test condition, or if they are both conses and the car of *object1* is **tree-equal** to the car of *object2* and the cdr of *object1* is **tree-equal** to the cdr of *object2*. Otherwise **tree-equal** returns false.

Syntax: **tree-equal** *object1 object2* `&key :test :test-not` [*Function*]

Remarks: If a test argument is not specified, **eql** is used. Either the keyword **:test** or the keyword **:test-not** may be used to specify a test function other than **eql**.

Examples:

```
> (setq tree1 '(1 (1 2))
        tree2 '(1 (1 2)))
(1 (1 2))
> (tree-equal tree1 tree2)
T
> (eql tree1 tree2)
NIL
> (tree-equal tree1 tree2 :test 'eq)
T
```

See Also: **equal**

union, nunion

Purpose: The functions **union** and **nunion** take two lists and return a list that contains every element that occurs in either of the list arguments.

The result list of **union** may share cells with one of the argument lists. The function **nunion** may modify its argument lists.

Syntax:

union *list1 list2* `&key :test :test-not :key` [*Function*]

nunion *list1 list2* `&key :test :test-not :key` [*Function*]

Remarks: If there is an element that is duplicated by the two lists, only one instance of it will appear in the result. If, however, one of the lists itself contains duplicate elements, there may be duplication in the result.

Whether a list element is the same as another list element is determined by the functions specified by the keyword arguments. If a test argument is not specified, **eql** is used. Either the keyword **:test** or the keyword **:test-not** may be used to specify a test function other than **eql**.

The keyword **:key** may be used to specify that a part of a list element should be tested. Its argument should be a function of one argument that extracts the part to be tested from the element.

Examples:

```
> (setq lst1 '(1 2 (1 2) "a" "b")
        lst2 '(2 3 (2 3) "B" "C"))
(2 3 (2 3) "B" "C")
> (union lst1 lst2)
("b" "a" (1 2) 1 2 3 (2 3) "B" "C")
> (union lst1 lst2 :test 'equalp)
("a" (1 2) 1 2 3 (2 3) "B" "C")
> (nunion lst1 lst2)
(1 (1 2) "a" "b" 2 3 (2 3) "B" "C")
```

Chapter 16. Arrays

Chapter 16. Arrays

About Arrays

Arrays are structured objects whose components can be directly accessed by means of index values.

An array can have many dimensions. The number of dimensions of an array is termed its ***rank***. It is possible for an array to have zero dimensions. In this case it consists of one element. The total number of elements that can be contained in an array is otherwise given by the product of its dimensions. If the length of any dimension is 0, the array has no elements. The elements of a multidimensional array are stored in row-major order.

An array is indexed by a sequence of integers called ***subscripts***. Each index value corresponds to a dimension of the array; the length of the sequence must equal the number of dimensions of the array. Array indexing is zero-origin; all index values must be nonnegative.

Arrays may be general or specialized. A ***general array*** can have elements that are members of any Common Lisp data type. A ***specialized array*** is an array whose elements must all be members of a particular data type.

Arrays can share their contents with other arrays. An array that is defined to share elements with an existing array is called a ***displaced*** array. It is said to be ***displaced to*** the existing array.

Arrays can also be created whose size and shape may be adjusted dynamically. Such arrays are called ***adjustable arrays***.

Sun Common Lisp implements arrays by using several different primitive data types. In particular, simple vectors such as single-bit, 2-bit, 4-bit, 8-bit, 16-bit, 32-bit, and single-float vectors are directly implemented using primitive data types.

Vectors

A ***vector*** is a one-dimensional array. Since the vector data type is a subtype of the sequence data type, a vector is also a sequence. A ***general vector*** can have elements that are members of any Common Lisp data type. A ***specialized vector*** is a vector whose elements must all be members of a particular data type. ***Strings*** and ***bit vectors*** are important types of specialized vectors. Strings are vectors whose elements are of the string character data type. Bit vectors are one-dimensional arrays whose elements are of the bit data type.

Fill Pointers

A one-dimensional array can have a fill pointer. A ***fill pointer*** is an index into a vector. It is a nonnegative integer whose value is less than or equal to the number of elements that the vector can contain. The elements below the fill pointer are considered to be ***active***. If the fill pointer value is 0, the vector contains no active elements. The fill pointer may be used to fill in the elements of the vector incrementally and thus to vary the length of the active portion of the vector. Generally, vector functions observe fill pointers and only operate on the active portion of a vector.

A ***simple array*** is an array that is not displaced to another array, has no fill pointer, and whose size cannot be dynamically adjusted.

A ***simple vector*** is a vector that is not displaced to another array, has no fill pointer, and whose size cannot be dynamically adjusted.

Categories of Operations

This section groups operations on arrays according to functionality.

Data Type Predicates

arrayp	**simple-vector-p**
bit-vector-p	**vectorp**
simple-bit-vector-p	

These predicates determine whether an object is a type of array.

Array Creation and Modification

adjust-array	**vector**
make-array	

These functions create arrays and modify the shape and size of arrays.

Array Access

aref	**sbit**
bit	**svref**

These functions access the elements of arrays.

Array Predicates

adjustable-array-p	**array-in-bounds-p**
array-has-fill-pointer-p	

These predicates test properties of arrays.

Array Attributes

array-dimension	**array-rank-limit**
array-dimension-limit	**array-row-major-index**
array-dimensions	**array-total-size**
array-element-type	**array-total-size-limit**
array-rank	

These constructs provide information about existing arrays and about system limitations on arrays that may be created.

Manipulating Fill Pointers

fill-pointer	**vector-push**
vector-pop	**vector-push-extend**

These functions use fill pointers to access arrays.

Logical Operations on Bit Arrays

bit-and	**bit-nor**
bit-andc1	**bit-not**
bit-andc2	**bit-orc1**
bit-eqv	**bit-orc2**
bit-ior	**bit-xor**
bit-nand	

These functions operate on bit arrays.

adjust-array

Purpose: The function **adjust-array** may be used to change the dimensions or contents of an array. The resulting array is of the same type and rank as the original array.

The original array may be modified and returned, or a new array may be created and the original array displaced to it.

The *array* argument must be an adjustable array.

The *new-dimensions* argument is a list of nonnegative integers that specify the size of each dimension of the array. The length of the *dimensions* list implicitly specifies the rank of the array and must equal the rank of the original array. The size of each dimension must be smaller than the constant **array-dimension-limit**, and the total number of elements that the array can contain (as given by the product of all dimensions) must be less than the constant **array-total-size-limit**.

The function **adjust-array** may be used with a number of keyword arguments. The use of the optional keyword arguments is discussed below.

Syntax: **adjust-array** *array new-dimensions* &key `:element-type` [*Function*]
`:initial-element`
`:initial-contents`
`:fill-pointer`
`:displaced-to`
`:displaced-index-offset`

Remarks: The **:element-type** keyword argument is used to specify the type of the elements of the array. Its value is a type specifier. If the **:element-type** argument is specified, it must be a type that is compatible with the **:element-type** specification of the original array. If the new array is a displaced array, the **:element-type** argument must be compatible with the **:element-type** specification of the array that the new array shares elements with.

The **:initial-element** keyword argument may be used to initialize each new element of the new array. Only those elements of the new array that are not within the bounds of the old array are initialized; the other elements retain their former values. If the **:initial-element** option is not specified, elements of the new array that are not within the bounds of the old array are undefined. The value of the **:initial-element** argument must agree with the type specified by the **:element-type** argument if the latter is given or with the original array if not. The **:initial-element** argument may not be specified if either the **:initial-contents** or **:displaced-to** argument is given.

The **:initial-contents** keyword argument is used to initialize each element of the new array individually. Its value is a list of nested sequences. The depth of nesting must equal the rank of the array. The top-level sequences correspond to the first dimension of the array, the second-level to the second dimension, and so on. The lowest-level sequence elements correspond to the array elements themselves. They must be of a type compatible with the **:element-type** argument. The **:initial-contents** argument may not be specified if either the **:initial-element** or the **:displaced-to** argument is given.

The **:fill-pointer** keyword argument may be specified only if the original array has a fill pointer. It is used to reset the fill pointer. If the argument value is **t**, the fill pointer is set to the length of the array. Otherwise the argument value must be a nonnegative integer that is no larger than the length of the array.

The **:displaced-to** keyword argument is used to create a displaced array. The new array shares its contents with the array that is given as the argument to the **:displaced-to** option. If the **:displaced-to** argument is defaulted or **nil**, the new array is not a displaced array. The **:initial-elements** and **:initial-contents** options must not be specified if the **:displaced-to** argument is given. The new array may not contain more elements than the array it is displaced to. The **:displaced-index-offset** option is generally used in conjunction with the **:displaced-to** option.

The **:displaced-index-offset** keyword argument is used to specify the offset of the new array from the beginning of the array that it is displaced to. The value of the argument must be a nonnegative integer; if it is not specified, it defaults to 0. The **:displaced-index-offset** argument may be used only if the **:displaced-to** argument is specified. The size of the new array plus the offset value may not exceed the size of the array that it is displaced to.

Although the original array may be a displaced array, the resulting array is not a displaced array unless the **:displaced-to** argument is specified.

Examples:

```
> (adjustable-array-p
   (setq ada
         (adjust-array (make-array '(2 3)
                                   :adjustable t
                                   :initial-contents '((a b c) (1 2 3)))
                       '(4 6))))
T
> (array-dimensions ada)
(4 6)
> (aref ada 1 1)
2
```

See Also: **adjustable-array-p**

make-array

array-dimension-limit

array-total-size-limit

adjustable-array-p

Purpose: The predicate **adjustable-array-p** is true if its *array* argument is adjustable; otherwise it is false.

Syntax: **adjustable-array-p** *array* [*Function*]

Examples:

```
> (adjustable-array-p
   (make-array 5
               :element-type 'string-char
               :adjustable t
               :fill-pointer 3))
T
> (adjustable-array-p (make-array 4))
NIL
```

aref

Purpose: The function **aref** accesses and returns the array element specified by the given subscripts.

Syntax: **aref** *array* `&rest` *subscripts* [*Function*]

Remarks: The number of subscripts given must correspond to the rank of the array. Each subscript must be in bounds for its dimension.

The function **aref** ignores fill pointers when accessing elements.

The macro **setf** may be used with **aref** to destructively replace an array element.

Examples:

```
> (aref (setq ta (make-array 4)) 3)
NIL
> (setf (aref ta 3) 'alozab)
ALOZAB
> (aref ta 3)
ALOZAB
> (aref (make-array '(2 4)
                    :element-type '(unsigned-byte 2)
                    :initial-contents '((0 1 2 3) (3 2 1 0)))
        1 2)
1
```

See Also: **bit**

char

elt

svref

array-dimension

Purpose: The function **array-dimension** returns the size of the *axis-number* dimension of the given array.

Syntax: **array-dimension** *array axis-number* [*Function*]

Remarks: The *axis-number* argument must be a nonnegative integer less than the rank of the array. Axis numbering is zero-origin.

The function **array-dimension** ignores fill pointers and returns the actual size of the given dimension.

Examples:

```
> (array-dimension (make-array 4) 0)
4
> (array-dimension (make-array '(2 3) ) 1)
3
```

See Also: **length**

array-dimension-limit

Purpose: The constant **array-dimension-limit** is an integer that defines the upper exclusive bound on each dimension of an array.

The value of **array-dimension-limit** in Sun Common Lisp is $2^{24} - 1$.

Syntax: **array-dimension-limit** *[Constant]*

Examples:

```
> array-dimension-limit
16777215
```

array-dimensions

Purpose: The function **array-dimensions** returns a list whose elements are the dimensions of the given array.

Syntax: **array-dimensions** *array* [*Function*]

Examples:

```
> (array-dimensions (make-array 4))
(4)
> (array-dimensions (make-array '(2 3)))
(2 3)
```

array-element-type

Purpose: The function **array-element-type** returns a type specifier for the set of elements that the given array can contain.

Syntax: **array-element-type** *array* [*Function*]

Remarks: The value of the type specifier may be a supertype of the type requested by the user when the array was created.

Sun Common Lisp implements arrays by using several different primitive data types. In particular, simple vectors such as single-bit, 2-bit, 4-bit, 8-bit, 16-bit, 32-bit, and single-float vectors are directly implemented using primitive data types.

Examples:

```
> (array-element-type (make-array 4))
T
> (array-element-type (make-array 12 :element-type '(unsigned-byte 8)))
(UNSIGNED-BYTE 8)
> (array-element-type (make-array 12 :element-type '(unsigned-byte 5)))
(UNSIGNED-BYTE 8)
```

See Also: **make-array**

array-has-fill-pointer-p

Purpose: The predicate **array-has-fill-pointer-p** is true if its *array* argument has a fill pointer; otherwise it is false.

Syntax: **array-has-fill-pointer-p** *array* [*Function*]

Remarks: Only one-dimensional arrays can have fill pointers.

Examples:

```
> (array-has-fill-pointer-p (make-array 4))
NIL
> (array-has-fill-pointer-p
   (make-array 8
               :fill-pointer 2
               :initial-element 'bazola))
T
```

array-in-bounds-p

Purpose: The predicate **array-in-bounds-p** checks whether the *subscripts* are all legal for the given array. It returns true if all are in bounds; otherwise it returns false.

Syntax: **array-in-bounds-p** *array* &rest *subscripts* [*Function*]

Remarks: The number of subscripts given must equal the rank of the array.

The predicate **array-in-bounds-p** ignores fill pointers.

Examples:

```
> (array-in-bounds-p
   (setq foo (make-array '(7 11)
                         :element-type 'string-char))
   0 0)
T
> (array-in-bounds-p foo 6 10)
T
> (array-in-bounds-p foo 0 -1)
NIL
> (array-in-bounds-p foo 0 11)
NIL
> (array-in-bounds-p foo 7 0)
NIL
```

array-rank

Purpose: The function **array-rank** returns the number of dimensions of the given array as a nonnegative integer.

Syntax: **array-rank** *array* [*Function*]

Examples:

```
> (array-rank (make-array nil))
0
> (array-rank (make-array 4))
1
> (array-rank (make-array '(2 3)))
2
```

array-rank-limit

Purpose: The constant **array-rank-limit** is an integer that defines the upper exclusive bound on the rank of an array.

The value of **array-rank-limit** in Sun Common Lisp is 2^8.

Syntax: **array-rank-limit** [*Constant*]

Examples:

```
> array-rank-limit
256
```

array-row-major-index

Purpose: The function **array-row-major-index** computes the position according to the row-major ordering of the array for the element that is specified by the subscript arguments. The result is a nonnegative integer value that indicates the offset of the element from the beginning of the array.

Syntax: **array-row-major-index** *array* &rest *subscripts* [*Function*]

Remarks: The function **array-row-major-index** ignores fill pointers.

The number of subscripts given must correspond to the rank of the array. Each subscript must be in bounds for its dimension.

Examples:

```
> (array-row-major-index
   (setq foo (make-array '(4 7)
                         :element-type '(unsigned-byte 8)))
   1 2)
9
> (array-row-major-index (make-array '(2 3 4)
                                     :element-type '(unsigned-byte 8)
                                     :displaced-to foo
                                     :displaced-index-offset 4)
                          0 2 1)
9
```

array-total-size

Purpose: The function **array-total-size** returns the number of elements that can be contained in the given array. The result is the product of the dimensions of the array.

Syntax: **array-total-size** *array* [*Function*]

Remarks: The function **array-total-size** ignores fill pointers.

The size of a zero-dimensional array is 1.

Examples:

```
> (array-total-size (make-array nil))
1
> (array-total-size (make-array 4))
4
> (array-total-size (make-array '(2 3)))
6
```

array-total-size-limit

Purpose: The constant **array-total-size-limit** is an integer that defines the upper exclusive bound on the number of elements that any array can contain.

The value of **array-total-size-limit** in Sun Common Lisp is $2^{24} - 1$.

Syntax: **array-total-size-limit** *[Constant]*

Examples:

```
> array-total-size-limit
16777215
```

arrayp

Purpose: The predicate **arrayp** is true if its argument is an array; otherwise it is false.

Syntax: **arrayp** *object* [*Function*]

Examples:

```
> (arrayp (make-array '(2 3 4)
                      :adjustable t))
T
> (arrayp (make-array 6))
T
> (arrayp 12)
NIL
```

bit, sbit

Purpose: The functions **bit** and **sbit** access elements of bit arrays.

The function **bit** accesses and returns the bit array element specified by the list of subscripts.

The function **sbit** is identical to **bit** but requires an array argument that is a simple bit array.

Syntax: **bit** *bit-array* **&rest** *subscripts* [*Function*]

sbit *simple-bit-array* **&rest** *subscripts* [*Function*]

Remarks: The function **bit** ignores fill pointers when accessing elements.

The number of subscripts given must correspond to the rank of the array. Each subscript must be in bounds for its dimension.

The functions **bit** and **sbit** are like **aref** except that they require their array arguments to be a bit array and a simple bit array respectively. The function **sbit** is coded in-line by the compiler; it may be significantly faster than **bit**.

The macro **setf** may be used with **bit** or **sbit** to destructively replace a bit array element.

Examples:

```
> (bit (setq ba (make-array 8
                            :element-type 'bit
                            :initial-element 1))
       3)
1
> (setf (bit ba 3) 0)
0
> (bit ba 3)
0
> (sbit ba 5)
1
> (setf (sbit ba 5) 1)
1
> (sbit ba 5)
1
```

See Also: **aref**

bit-and, bit-andc1, bit-andc2, bit-eqv, bit-ior, bit-orc1, bit-orc2, bit-nand, bit-nor, bit-xor

Purpose: The functions **bit-and**, **bit-andc1**, **bit-andc2**, **bit-eqv**, **bit-ior**, **bit-orc1**, **bit-orc2**, **bit-nand**, **bit-nor**, and **bit-xor** perform bit-wise logical operations on bit arrays and return the resulting array.

If the *result-bit-array* argument is specified, the contents of that array are replaced with the result; if it is **t**, the contents of *bit-array1* are replaced with the result; if it is **nil** or unspecified, a new array is created.

The function **bit-and** returns the *logical and* of its bit array arguments.

The function **bit-andc1** returns the *logical and* of its first argument with the *logical complement* of its second argument.

The function **bit-andc2** returns the *logical and* of its second argument with the *logical complement* of its first argument.

The function **bit-eqv** returns the *logical equivalence* of its bit array arguments.

The function **bit-ior** returns the *logical inclusive or* of its bit array arguments.

The function **bit-nand** performs the *logical and* operation on its bit array arguments and returns the *logical complement* of the result.

The function **bit-nor** performs the *logical inclusive or* operation on its bit array arguments and returns the *logical complement* of the result.

The function **bit-orc1** returns the *logical inclusive or* of its first argument with the *logical complement* of its second argument.

The function **bit-orc2** returns the *logical inclusive or* of its second argument with the *logical complement* of its first argument.

The function **bit-xor** returns the *logical exclusive or* of its bit array arguments.

Syntax:

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-orc1 *bit-array1 bit-array2* `&optional` *result-bit-array* [*Function*]

bit-orc2 *bit-array1 bit-array2* `&optional` *result-bit-array* [*Function*]

bit-xor *bit-array1 bit-array2* `&optional` *result-bit-array* [*Function*]

Remarks: The array arguments must all be of the same rank and dimensions. The result is a bit array of the same rank and dimensions as the arguments.

Examples:

```
> (setq *print-array* t)
T
> (bit-and (setq ba #*11101010) #*01101011)
#*01101010
> (setq rba (bit-andc2 ba #*00110011 t))
#*11001000
> (eq rba ba)
T
```

See Also: **bit-not**

bit-not

Purpose: The function **bit-not** inverts the bits in its *bit-array* argument and returns the resulting array.

If the *result-bit-array* argument is specified, the contents of that array are replaced with the result; if it is **t**, the contents of *bit-array* are replaced with the result; if it is **nil** or unspecified, a new array is created.

Syntax: **bit-not** *bit-array* `&optional` *result-bit-array* [*Function*]

Remarks: The resulting array is identical in rank and dimensions to the original array.

Examples:

```
> (setq *print-array* t)
T
> (bit-not (setq ba #*11101010))
#*00010101
> (setq rba (bit-not ba
                     (setq tba (make-array 8
                                           :element-type 'bit))))
#*00010101
> (equal rba tba)
T
```

bit-vector-p

Purpose: The predicate **bit-vector-p** is true if its argument is a bit vector; otherwise it is false.

Syntax: **bit-vector-p** *object* [*Function*]

Examples:

```
> (bit-vector-p (make-array 6
                            :element-type 'bit
                            :fill-pointer t))
T
> (bit-vector-p #*)
T
> (bit-vector-p (make-array 6))
NIL
```

fill-pointer

Purpose: The function **fill-pointer** returns the fill pointer of the specified vector. The *vector* argument must be a vector with a fill pointer.

Syntax: **fill-pointer** *vector* [*Function*]

Remarks: The macro **setf** may be used with **fill-pointer** to modify the fill pointer of a vector. The new fill pointer value must be a nonnegative integer less than or equal to the length of the vector.

Examples:

```
> (fill-pointer
   (setq fa (make-array 8
                        :fill-pointer 2
                        :initial-element 'bazola)))
2
> (setf (fill-pointer fa) 0)
0
> (fill-pointer fa)
0
```

make-array

Purpose: The function **make-array** creates and returns a new array.

The *dimensions* argument is a list of nonnegative integers that specify the size of each of the dimensions of the array. The length of the *dimensions* list implicitly specifies the rank of the array. The total number of dimensions of the array must be less than the constant **array-rank-limit**. The size of each dimension must be smaller than the constant **array-dimension-limit**, and the total number of elements that the array can contain (as given by the product of all dimensions) must be less than the constant **array-total-size-limit**.

If the *dimensions* argument is **nil**, a zero-dimensional array is created.

If a one-dimensional array is to be created, the *dimensions* argument may be given as a single integer rather than as a list.

The function **make-array** may be used with a number of keyword arguments. The use of the optional keyword arguments is discussed below.

Syntax: **make-array** *dimensions* &key `:element-type` [*Function*]
`:initial-element`
`:initial-contents`
`:adjustable`
`:fill-pointer`
`:displaced-to`
`:displaced-index-offset`

Remarks: The **:element-type** keyword argument is used to specify the type of the elements of the new array. Its value is a type specifier. The array that is created is of the most appropriate implementation type that can contain elements of this type. Sun Common Lisp implements arrays by using several different primitive data types. In particular, simple vectors such as single-bit, 2-bit, 4-bit, 8-bit, 16-bit, 32-bit, and single-float vectors are directly implemented using primitive data types. The **:element-type** argument defaults to **t**, the most general type.

The **:initial-element** keyword argument may be used to specify the initial value of all elements of the new array. This value must be of the type specified by the **:element-type** argument if the latter is given. If the **:initial-element** argument is not specified, the initial contents of the array elements are undefined. The **:initial-element** argument may not be specified if either the **:initial-contents** or the **:displaced-to** argument is given.

The **:initial-contents** keyword argument is used to initialize each element of the new array individually. Its value is a list of nested sequences. The depth of nesting must equal the rank of the array. The top-level sequences correspond to the first dimension of the array, the second-level to the second dimension, and so on. The lowest-level sequence elements correspond to the array elements themselves. They must be of a type compatible with the **:element-type** argument. The **:initial-contents** argument may not be specified if either the **:initial-element** or the **:displaced-to** argument is given.

The **:adjustable** keyword argument is used to specify that the size of the array may be adjusted dynamically. If this argument is specified and is non-**nil**, the array that is created is adjustable.

The **:fill-pointer** keyword argument is used to specify that the array is to have a fill pointer. Only one-dimensional arrays may have fill pointers. The fill pointer is initialized to the value of the **:fill-pointer** argument. If the argument value is **t**, the fill pointer is set to the length of the array. Otherwise it must be a nonnegative integer that is no larger than the length of the array. If the fill pointer argument is defaulted or **nil**, the array will not have a fill pointer.

The **:displaced-to** keyword argument is used to create a displaced array. The new array shares its contents with the array that is given as the argument to the **:displaced-to** option. If the **:element-type** option is also specified, it must be the same as that of the array that the new array shares elements with. If the **:displaced-to** argument is defaulted or **nil**, the new array is not a displaced array. The **:initial-elements** or **:initial-contents** option must not be specified if the **:displaced-to** argument is given. The new array may not contain more elements than the array it is displaced to. The **:displaced-index-offset** option is generally used in conjunction with the **:displaced-to** option.

The **:displaced-index-offset** keyword argument is used to specify the offset of the new array from the beginning of the array that it is displaced to. The value of the argument must be a nonnegative integer; if it is not specified, it defaults to 0. The **:displaced-index-offset** argument may be used only if the **:displaced-to** argument is specified. The size of the new array plus the offset value may not exceed the size of the array that it is displaced to.

If the **:adjustable**, **:fill-pointer**, and **:displaced-to** arguments are all defaulted or **nil**, a simple array is created.

Examples:

```
> (setq *print-array* t)
T
> (make-array nil)
#0A NIL
> (make-array 4)
#(NIL NIL NIL NIL)
> (make-array '(2 4)
              :element-type '(unsigned-byte 2)
              :initial-contents '((0 1 2 3) (3 2 1 0)))
#2A((0 1 2 3) (3 2 1 0))
> (make-array 6
              :element-type 'string-char
              :initial-element #\a
              :fill-pointer 3)
"aaa"
```

See Also:

array-dimension-limit

array-rank-limit

array-total-size-limit

simple-bit-vector-p

Purpose: The predicate **simple-bit-vector-p** is true if its argument is a simple bit vector; otherwise it is false.

Syntax: **simple-bit-vector-p** *object* [*Function*]

Examples:

```
> (simple-bit-vector-p (make-array 6))
NIL
> (simple-bit-vector-p #*)
T
```

simple-vector-p

Purpose: The predicate **simple-vector-p** is true if its argument is a simple general vector; otherwise it is false.

Syntax: **simple-vector-p** *object* [*Function*]

Examples:

```
> (simple-vector-p (make-array 6))
T
> (simple-vector-p "aaaaaa")
NIL
> (simple-vector-p (make-array 6 :fill-pointer t))
NIL
```

svref

Purpose: The function **svref** accesses a simple vector and returns the element specified by its *index* argument.

Syntax: **svref** *simple-vector index* [*Function*]

Remarks: The index value must be in bounds for the vector.

The function **svref** is like **aref** except that it requires its array argument to be a simple vector. The function **svref** is coded in-line by the compiler; it may be significantly faster than **aref**.

The macro **setf** may be used with **svref** to destructively replace an element of a simple vector.

Examples:

```
> (setq *print-array* t)
T
> (simple-vector-p (setq v (vector 1 2 'bazola)))
T
> (svref v 0)
1
> (svref v 2)
BAZOLA
> (setf (svref v 1) 'newcomer)
NEWCOMER
> v
#(1 NEWCOMER BAZOLA)
```

See Also:

aref

sbit

schar

vector

vector

Purpose: The function **vector** creates and returns a simple general vector whose size corresponds to the number of object arguments. It is initialized to contain the objects specified by the arguments.

Syntax: **vector** `&rest` *objects* [*Function*]

Examples:

```
> (arrayp (setq v (vector 1 2 'bazola)))
T
> (vectorp v)
T
> (simple-vector-p v)
T
> (length v)
3
```

See Also: **make-array**

vector-pop

Purpose: The function **vector-pop** is used to retrieve elements from vectors having fill pointers. The fill pointer is decremented by 1, and the vector element indicated by the new fill pointer is returned as the result.

Syntax: **vector-pop** *vector* [*Function*]

Remarks: The initial value of the fill pointer must be positive.

The *vector* argument must specify a vector that has a fill pointer.

Examples:

```
> (vector-push (setq frob (list 'frob))
               (setq fa (make-array 8
                                    :fill-pointer 2
                                    :initial-element 'bazola)))
2
> (fill-pointer fa)
3
> (eq (vector-pop fa) frob)
T
> (vector-pop fa)
BAZOLA
> (fill-pointer fa)
1
```

See Also: **vector-push**

vector-push, vector-push-extend

Purpose: The functions **vector-push** and **vector-push-extend** are used to store elements in vectors having fill pointers.

The function **vector-push** attempts to store the *new-element* argument in the vector location indicated by the fill pointer and then to increment the fill pointer. If it succeeds, the original value of the fill pointer is returned as the result. If the fill pointer is already too large, **vector-push** returns **nil** and does not modify the vector.

The function **vector-push-extend** is like **vector-push** except that the vector is extended when the fill pointer becomes too large. Its *vector* argument must be an adjustable vector.

Syntax: **vector-push** *new-element vector* [*Function*]

vector-push-extend *new-element vector* `&optional` *extension* [*Function*]

Remarks: The *extension* argument of **vector-push-extend** may be used to specify the minimum number of elements by which the array should be extended. It must be a positive integer.

The *vector* argument must specify a vector that has a fill pointer. The new element may be any object that is compatible with the type of the vector.

Examples:

```
> (vector-push (setq frob (list 'frob))
               (setq fa (make-array 8
                                    :fill-pointer 2
                                    :initial-element 'bazola)))
2
> (fill-pointer fa)
3
> (eq (aref fa 2) frob)
T
> (vector-push-extend #\X
                      (setq aa
                            (make-array 5
                                        :element-type 'string-char
                                        :adjustable t
                                        :fill-pointer 3)))
3
> (fill-pointer aa)
4
```

```
> (vector-push-extend #\Y aa 4)
4
> (array-total-size aa)
5
> (vector-push-extend #\Z aa 4)
5
> (array-total-size aa)
9
```

See Also: adjustable-array-p

vector-pop

vectorp

Purpose: The predicate **vectorp** is true if its argument is a vector; otherwise it is false.

Syntax: **vectorp** *object* [*Function*]

Examples:

```
> (vectorp "aaaaaa")
T
> (vectorp (make-array 6 :fill-pointer t))
T
> (vectorp (make-array '(2 3 4)))
NIL
```

Chapter 17. Strings

Chapter 17. Strings

About Strings

A ***string*** is a vector whose elements must be of the string character data type. The string type is identical to the type `(vector string-char)`.

Like other vectors and arrays, strings may have fill pointers. A ***fill pointer*** is an index into a string. It is a nonnegative integer whose value is less than or equal to the number of elements that the string can contain. The elements below the fill pointer are considered to be ***active***. If the fill pointer value is 0, the string contains no active elements. The fill pointer may be used to fill in the elements of the string incrementally and thus to vary the length of the active portion of the string. Generally, string functions observe fill pointers and only operate on the active portion of a string.

A ***simple string*** is a simple array. In particular, it has no fill pointer. Simple strings use less storage than general strings. Operations on simple strings tend to be faster than those on general strings.

Categories of Operations

This section groups operations on strings according to functionality.

Data Type Predicates

stringp	**simple-string-p**

These predicates determine whether an object is a string.

String Access

char	**schar**

These functions access a single string element.

String Comparison

string=	**string-equal**
string<	**string-lessp**
string<=	**string-not-greaterp**
string>	**string-greaterp**
string>=	**string-not-lessp**
string/=	**string-not-equal**

These functions perform lexicographic comparisons on strings. Both case-sensitive and case-insensitive forms of each operation are provided.

String Construction

make-string	**string**

These functions create new strings.

String Manipulation

string-trim	**string-capitalize**
string-left-trim	**nstring-upcase**
string-right-trim	**nstring-downcase**
string-upcase	**nstring-capitalize**
string-downcase	

These functions modify strings.

char, schar

Purpose: The function **char** accesses and returns as a character object the string element specified by *index*. The *index* value must be a nonnegative integer that is less than the length of the string.

The function **schar** is identical to **char** but requires a string argument that is a simple string.

Syntax:

char *string index* [*Function*]

schar *simple-string index* [*Function*]

Remarks: The function **char** ignores fill pointers when accessing elements.

The index is an offset value from the beginning of the string; indexing is zero-origin.

The macro **setf** may be used with **char** or **schar** to destructively replace a string element.

Examples:

```
> (setq simp-str "abcdef")
"abcdef"
> (setq un-simp-str (make-array '(6)
                                :element-type 'string-char
                                :fill-pointer 0
                                :displaced-to simp-str))
""
> (schar simp-str 2)
#\c
> (char un-simp-str 2)
#\c
> (setf (schar simp-str 2) #\C)
#\C
> simp-str
"abCdef"
```

See Also:

aref

elt

make-string

Purpose: The function **make-string** creates and returns a string of length *size*.

If the **:initial-element** argument is specified, all string elements are initialized to its value.

Syntax: **make-string** *size* `&key :initial-element` [*Function*]

Remarks: The resulting string is a simple string.

Examples:

```
> (make-string 10 :initial-element #\5)
"5555555555"
> (length (make-string 10))
10
```

simple-string-p

Purpose: The predicate **simple-string-p** is true if its argument is a simple string; otherwise it is false.

Syntax: **simple-string-p** *object* [*Function*]

Examples:

```
> (simple-string-p "aaaaaa")
T
> (simple-string-p (make-array 6
                               :element-type 'string-char
                               :fill-pointer t))
NIL
```

string

Purpose: The function **string** converts symbol and single-character arguments to strings.

If the argument of **string** is a string, it is returned; if it is a symbol, the print name of the symbol is returned as a string; if it is a string character, a one-element string containing that character is returned. If the argument is of any other type, **string** signals an error.

Syntax: **string** *x* [*Function*]

Examples:

```
> (string "already a string")
"already a string"
> (string 'foo)
"FOO"
> (string #\c)
"c"
```

See Also: **coerce**

string-char-p

string<, string<=, string>, string>=, string/=, string-lessp, string-not-greaterp, string-greaterp, string-not-lessp, string-not-equal

Purpose: These functions perform lexicographic comparisons upon their string arguments. The comparison operations may be restricted to substrings of these strings by specifying the **:start** and **:end** keyword arguments.

The functions **string<**, **string<=**, **string>**, **string>=**, and **string/=** check to see if the first of these substrings is less than, less than or equal to, greater than, greater than or equal to, or not equal to the second respectively. If so, they return the first character position at which the two substrings differ as an integer offset from the beginning of *string1*. If the two substrings are identical, **nil** is returned.

The functions **string-lessp**, **string-not-greaterp**, **string-greaterp**, **string-not-lessp**, and **string-not-equal** are identical to **string<**, **string<=**, **string>**, **string>=**, and **string/=** respectively but ignore differences in case.

Syntax:

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-lessp *string1 string2* &key :start1 :end1 :start2 :end2 [*Function*]

string-not-greaterp *string1 string2* [*Function*]
&key :start1 :end1 :start2 :end2

string-greaterp *string1 string2* &key :start1 :end1 :start2 :end2 [*Function*]

string-not-lessp *string1 string2* &key :start1 :end1 :start2 :end2 [*Function*]

string-not-equal *string1 string2* &key :start1 :end1 :start2 :end2 [*Function*]

Remarks: The **:start** and **:end** keyword arguments take integer values that specify offsets into the original strings. The **:start** arguments mark the beginning positions of the substrings; the **:end** arguments mark the positions following the last elements of the substrings. The start values default to 0; the end values default to the lengths of the strings.

The *string1* and *string2* arguments may be either symbols or strings. If a symbol is specified, the symbol's print name is used.

Examples:

```
> (string< "aaaa" "aaab")
3
> (string>= "aaaaa" "aaaa")
4
> (string-lessp "012AAAA789" "01aaab6" :start1 3 :end1 7
                                       :start2 2 :end2 6)
6
> (string-not-equal "AAAA" "aaaA")
NIL
```

See Also:

string=

string-equal

string=, string-equal

Purpose: The functions **string=** and **string-equal** perform comparisons upon strings. The comparison operations may be restricted to substrings of these strings by specifying the **:start** and **:end** keyword arguments.

The function **string=** is true if the substrings are of the same length and contain identical characters in corresponding positions; it is false if any of these conditions does not hold.

The function **string-equal** is identical to **string=** except that differences in case are ignored.

Syntax:

string= *string1 string2* `&key :start1 :end1 :start2 :end2` [*Function*]

string-equal *string1 string2* `&key :start1 :end1 :start2 :end2` [*Function*]

Remarks: The **:start** and **:end** keyword arguments take integer values that specify offsets into the original strings. The **:start** arguments mark the beginning positions of the substrings; the **:end** arguments mark the positions following the last elements of the substrings. The start values default to 0; the end values default to the lengths of the strings.

The *string1* and *string2* arguments may be either symbols or strings. If a symbol is specified, the symbol's print name is used.

Examples:

```
> (string= "abcd" "01234abcd9012" :start2 5 :end2 9)
4
> (string= "Abcd" "abcd")
NIL
> (string-equal "Abcde" "abcdE")
5
```

string-trim, string-left-trim, string-right-trim

Purpose: The function **string-trim** returns a copy of its *string* argument from which the largest prefix and suffix containing only characters from *character-bag* have been removed. The argument *character-bag* can be any sequence containing characters.

If no characters can be trimmed, a copy of the original string or the original string itself is returned.

The functions **string-left-trim** and **string-right-trim** are identical to **string-trim** except that characters are removed from only the left or right ends of the string respectively.

Syntax:

string-trim *character-bag string* [*Function*]

string-left-trim *character-bag string* [*Function*]

string-right-trim *character-bag string* [*Function*]

Remarks: The *string* argument may be either a symbol or a string. If a symbol is specified, the symbol's print name is used.

Examples:

```
> (string-trim "abc" "abcaakaaakabcaaa")
"kaaak"
> (string-right-trim " (*)" " ( *three (silly) words* ) ")
" ( *three (silly) words"
> (string-left-trim "abc" "labcabcabc")
"labcabcabc"
```

string-upcase, string-downcase, string-capitalize, nstring-upcase, nstring-downcase, nstring-capitalize

Purpose: The function **string-upcase** returns a copy of its *string* argument in which all lowercase characters have been converted to the corresponding uppercase characters. The case conversion operation may be restricted to a substring of the string by specifying the **:start** and **:end** keyword arguments.

The function **string-downcase** is like **string-upcase** except that all uppercase characters are replaced by the corresponding lowercase characters.

The function **string-capitalize** returns a copy of its *string* argument in which the first character of every word is uppercase, if possible, and all others are lowercase. A word is considered to be any consecutive subsequence of alphanumeric characters delimited by nonalphanumeric characters or the end of the string.

The functions **nstring-upcase**, **nstring-downcase**, and **nstring-capitalize** are identical to **string-upcase**, **string-downcase**, and **string-capitalize** respectively except that they modify the *string* argument.

Syntax:

string-upcase *string* `&key :start :end` [*Function*]

string-downcase *string* `&key :start :end` [*Function*]

string-capitalize *string* `&key :start :end` [*Function*]

nstring-upcase *string* `&key :start :end` [*Function*]

nstring-downcase *string* `&key :start :end` [*Function*]

nstring-capitalize *string* `&key :start :end` [*Function*]

Remarks: The **:start** and **:end** keyword arguments take integer values that specify offsets into the original strings. The **:start** argument marks the beginning position of the substring; the **:end** argument marks the position following the last element of the substring. The start value defaults to 0; the end value defaults to the length of the string.

The *string* argument of the functions **string-upcase**, **string-downcase**, and **string-capitalize** may be either a symbol or a string. If a symbol is specified, the symbol's print name is used.

Examples:

```
> (string-upcase "abcde")
"ABCDE"
> (setq str "0123ABCD890a")
"0123ABCD890a"
> (nstring-downcase str :start 5 :end 7)
"0123AbcD890a"
> str
"0123AbcD890a"
> (string-capitalize "f0o 13c arthur;bar don't")
"Foo 13c Arthur;Bar Don'T"
```

See Also:

char-upcase

char-downcase

stringp

Purpose: The predicate **stringp** is true if its argument is a string; otherwise it is false.

Syntax: **stringp** *object* [*Function*]

Examples:

```
> (stringp "aaaaaa")
T
> (stringp (make-array 6
                       :element-type 'string-char
                       :fill-pointer t))
T
> (stringp #\a)
NIL
```

Chapter 18. Hash Tables

Chapter 18. Hash Tables

About Hash Tables

Hash tables are Common Lisp objects that provide mappings between other objects. Each hash table entry is a pair of associated objects, a ***key*** and a ***value***. Hash table functions use keys to look up their associated values. Both keys and values can be any Lisp objects. Hash table keys are unique: at most one value can be associated with a key at a given time.

The size of a hash table corresponds to the maximum number of entries that it can hold. Although entries may be added and deleted, this size remains fixed until the capacity of the hash table is exceeded, at which time the hash table is automatically extended and reorganized.

Hash tables are designed so that a value associated with a key may be found quickly in situations where there is a large number of entries. This is their advantage over property lists and association lists.

Categories of Operations

This section groups operations on hash tables according to functionality.

Data Type Predicates

hash-table-p

This predicate determines whether an object is a hash table.

Hash Table Functions

clrhash	**make-hash-table**
gethash	**maphash**
hash-table-count	**remhash**

These functions create and manipulate hash tables.

Hash Functions

sxhash

This function is designed to allow the user to implement conveniently more complex hashed data structures than are provided by the hash table facility.

clrhash

Purpose: The function **clrhash** removes all entries from its *hash-table* argument and returns the empty hash table as its result.

Syntax: **clrhash** *hash-table* [*Function*]

Examples:

```
> (setq h (make-hash-table))
#<Hash-Table 3BB92B>
> (dotimes (i 100) (setf (gethash i h) i))
NIL
> (hash-table-count h)
100
> (hash-table-count (clrhash h))
0
> (hash-table-count h)
0
```

gethash

Purpose: The function **gethash** finds the hash table entry that is associated with a given key and returns its value. If such an entry does not exist, **gethash** returns *default* if this value has been specified, or **nil** if not. The function **gethash** also returns a second value; this value is true if the entry was found and false otherwise.

Syntax: **gethash** *key hash-table* `&optional` *default* [*Function*]

Remarks: The macro **setf** may be used with **gethash** to add entries to the table. If an entry with the same key already exists, that entry is replaced.

Examples:

```
> (setq h (make-hash-table))
#<Hash-Table 3BB153>
> (setf (gethash 1 h) "one")
"one"
> (setf (gethash 'nil h "ignored default") nil)
NIL
> (gethash 1 h)
"one"
T
> (gethash 'nil h "default")
NIL
T
> (gethash 2 h)
NIL
NIL
> (gethash 2 h "default")
"default"
NIL
```

hash-table-count

Purpose: The function **hash-table-count** returns the number of entries in a given hash table. If the hash table has just been created or newly cleared, the entry count is 0.

Syntax: **hash-table-count** *hash-table* [*Function*]

Examples:

```
> (setq h (make-hash-table))
#<Hash-Table 3BF7DB>
> (hash-table-count h)
0
> (setf (gethash 57 h) "57")
"57"
> (hash-table-count h)
1
> (dotimes (i 100) (setf (gethash i h) i))
NIL
> (hash-table-count h)
100
```

hash-table-p

Purpose: The predicate **hash-table-p** is true if its argument is a hash table; otherwise it is false.

Syntax: **hash-table-p** *object* [*Function*]

Examples:

```
> (setq h (make-hash-table))
#<Hash-Table 3BF7DB>
> (hash-table-p h)
T
> (hash-table-p 'h)
NIL
```

make-hash-table

Purpose: The function **make-hash-table** creates and returns a new hash table.

Syntax: **make-hash-table** `&key :test` [*Function*]

```
                  :size
                  :rehash-size
                  :rehash-threshold
```

Remarks: The meanings of the keyword arguments are as follows:

The **:test** argument specifies the predicate to be used in comparing keys. The argument value must be one of the following if given: **#'eq**, **#'eql**, **#'equal**, **eq**, **eql**, or **equal**. If a value is not specified, **eql** is used.

Hash tables that use **eq** or **eql** may use actual addresses to compute the associated hash values. These types of hash tables may need to be rehashed after a garbage collection.

The **:size** argument specifies the initial size of the hash table in terms of the number of possible entries. This value is a hint to the system: the hash table that is created may actually be larger. The **:size** argument must be a positive fixnum.

The **:rehash-size** argument specifies how much the hash table should be extended when it becomes full. This value can be a positive integer or a floating-point number greater than 1. If **:rehash-size** is specified as an integer, the table will be extended by that number of entries. If it is a floating-point value, the hash table will grow by that factor each time it is extended.

The **:rehash-threshold** argument specifies how full the hash table may become before it is extended. This value may be a positive integer less than **:rehash-size** or a floating-point value greater than 0.0 and less than or equal to 1.0. If an integer value is specified, this value will be scaled appropriately when the table is extended.

The **:size**, **:rehash-size**, and **:rehash-threshold** arguments have implementation-dependent default values.

Examples:

```
> (setq h (make-hash-table))
#<Hash-Table 3C420B>
> (setf (gethash "one" h) 1)
1
> (gethash "one" h)
NIL
NIL
> (setq h (make-hash-table :test 'equal))
#<Hash-Table 3C542B>
> (setf (gethash "one" h) 1)
1
> (gethash "one" h)
1
T
> (make-hash-table :size 100 :rehash-size 50 :rehash-threshold 75)
#<Hash-Table 3BDF4B>
> (make-hash-table :rehash-size 1.5 :rehash-threshold 0.7)
#<Hash-Table 3BE283>
```

maphash

Purpose: The function **maphash** calls its *function* argument on each entry of a given hash table.

The *function* argument must be a function of two arguments. These arguments should correspond to the key and the value of the hash table entry respectively.

The function **maphash** returns **nil** as its result.

Syntax: **maphash** *function hash-table* [*Function*]

Remarks: Adding or deleting hash table entries while a **maphash** operation is in progress may cause unpredictable results.

Examples:

```
> (setq h (make-hash-table))
#<Hash-Table 3BE59B>
> (dotimes (i 10) (setf (gethash i h) i))
NIL
> (let ((sum-of-squares 0))
    (maphash #'(lambda (key val)
                 (incf sum-of-squares (* val val)))
             h)
    sum-of-squares)
285
> (hash-table-count h)
10
> (maphash #'(lambda (key val)
               (when (oddp val) (remhash key h)))
           h)
NIL
> (hash-table-count h)
5
```

remhash

Purpose: The function **remhash** removes the hash table entry with a given key if such an entry exists. It returns true if such an entry was found, and false if not.

Syntax: **remhash** *key hash-table* [*Function*]

Examples:

```
> (setq h (make-hash-table))
#<Hash-Table 3BE59B>
> (setf (gethash 100 h) "C")
"C"
> (gethash 100 h)
"C"
T
> (remhash 100 h)
T
> (gethash 100 h)
NIL
NIL
> (remhash 100 h)
NIL
```

sxhash

Purpose: The function **sxhash** computes a nonnegative fixnum from its *object* argument. This result is intended for use as a hash value.

Syntax: **sxhash** *object* [*Function*]

Remarks: The function **sxhash** is designed to allow the user to implement conveniently more complex hashed data structures than are provided by the **make-hash-table** construct.

If two objects are the same (**equal**), **sxhash** returns the same hash value for both.

Examples:

```
> (= (sxhash (list 'list "ab")) (sxhash (list 'list "ab")))
T
```

Chapter 19. Structures

Chapter 19. Structures

About Structures

Common Lisp allows the user to create named record structures with a fixed number of named components called ***slots***. These structures are, in effect, user-defined data types. All are created with the **defstruct** macro. When these data types are defined, constructs to manipulate them are normally automatically defined as well. These constructs include type predicates and access, constructor, and copier functions.

The representation of structures may be explicitly controlled. The user may specify how a structure is to be implemented and how slots are to be allocated.

Structures can be either ***named*** or ***unnamed***. From any instance of a named structure, the user can obtain the structure name for the type.

The simple use of **defstruct** is discussed first; a description of all the **defstruct** features and options follows.

```
defstruct name-and-options [documentation] {slot-description}*

name-and-options::= structure-name | (structure-name {option}*)

structure-name::= symbol

option::= :conc-name | (:conc-name symbol) | (:conc-name nil) |
          :constructor | (:constructor symbol) | (:constructor nil) |
          (:constructor symbol boa-arglist) |
          :copier | (:copier symbol) | (:copier nil) |
          :predicate | (:predicate symbol) | (:predicate nil) |
          (:include existing-defstruct-name {slot-description}*) |
          (:print-function function) |
          (:type {list | vector | (vector type)} ) |
          :named |
          (:initial-offset non-negative-integer)

boa-arglist::= ({symbol}*
               [&optional {var | (var [initform] )}*]
               [&rest var]
               [&aux {var | (var [initform])}*])

slot-description::= slot-name | (slot-name [initform {slot-option}*] )

slot-option::= :type type | :read-only flag
```

Figure 19–1. Syntax for Defstruct

Defining Structures

The macro **defstruct** is used to define a structure. Its complete syntax is shown in Figure 19–1.

The *structure-name* argument of **defstruct** is a symbol. It becomes the name of the new type.

The components of the structure, or slots, are specified by slot descriptions.

The *slot-name* is a symbol. All the other *slot-description* arguments are optional. Of these, the *initform* argument is the most important. The *initform* argument is a form that specifies a default value for the slot. It is evaluated when a new instance of the structure is created and no slot value has been given. The other slot options are discussed later in this chapter.

If an *initform* value and other options are not specified, the *slot-name* argument may be given by itself (not in a list).

A documentation string may be attached to the structure name by specifying the optional *documentation* argument; the documentation type for this string is **structure**.

Automatically Defined Functions

In the simple case where no **defstruct** options have been specified, the following functions are automatically defined to operate on instances of the new structure.

A predicate with the name *structure-name*-**p** is defined to test membership in the structure type. The predicate (*structure-name*-p *object*) is true if an object is of this type; otherwise it is false. The function **typep** may also be used with the name of the new type to test whether an object belongs to the type. Such a function call has the form (typep *object* '*structure-name*).

Access functions are defined to access the components of the structure. For each slot name, there is a corresponding access function with the name *structure-name-slot-name*. This function accesses the contents of that slot. Each access function takes one argument, which is an instance of the structure type. The macro **setf** may be used with any of these access functions to alter the slot contents.

A constructor function with the name **make-***structure-name* is defined. This function creates and returns new instances of the structure type.

A copier function with the name **copy-***structure-name* is defined. The copier function takes an object of the structure type and creates a new object of the same type that is a copy of the first. The copier function creates a new structure with the same component entries as the original. Corresponding components of the two structure instances are **eql**.

The predicate, access function, constructor function, and copier function names are all defined in whatever package is current at the time the **defstruct** macro is processed.

Constructing New Instances of the Structure

After a new structure type has been defined, instances of that type normally can be created by using the constructor function for the type.

A call to a constructor function is of the following form:

(*constructor-function-name*
 slot-keyword-1 form1
 slot-keyword-2 form-2
 ...)

The arguments to the constructor function are all keyword arguments. Each slot keyword argument must be a keyword whose name corresponds to the name of a structure slot.

For each slot keyword, the associated form is evaluated, and the slot is initialized to its value. If a slot is not initialized in this way, it is initialized by evaluating the *initform* argument in the slot description. The *initform* argument is evaluated at the time the new structure instance is created, but in the lexical environment of the **defstruct** form in which it was defined. If no such initialization form was specified, the contents of the slot are undefined.

Defstruct Slot Options

The following keyword options are available for use in the slot descriptions. No part of these options is evaluated. For a slot option to be specified, the default value for the slot needs to have been defined by use of the *initform* argument in the slot description.

- The **:read-only** option controls whether the contents of the slot may be modified. If **:read-only** is specified with a non-**nil** argument, **setf** will not accept the access function for the slot, and the slot will always contain the default value. If **:read-only** is specified with a **nil** argument, the option has no effect.
- The **:type** option specifies the type of the slot contents. The argument to **:type** must be a type specifier. If the **:type** option is specified, the slot contents must be of the given type.

Defstruct Options

The following keyword options are available for use with **defstruct**. No part of these options is evaluated.

- The **:conc-name** option controls the naming of the access functions. When **:conc-name** is defaulted, components of the structure are accessed individually by functions whose names consist of the structure name, a hyphen, and then the name of the component. A symbol or string argument may be provided for use as an alternate prefix for the access function name. This prefix is added to each of the component names to form the names of the access functions. If a hyphen is to separate the component name, it must be included as part of the prefix given to **:conc-name**. If **:conc-name** is specified as **nil**, the names of the access functions are the same as those of the components. The names of the access functions are entered into the package that is current at the time the **defstruct** macro is processed.

- The **:constructor** option controls the naming of the constructor function. If this option is defaulted, the constructor name is **make-** followed by the structure name. The name of the constructor function is entered into the package that is current at the time the **defstruct** macro is processed. A symbol argument may be provided that specifies a different name for the constructor function. If the option is specified as **nil**, no constructor function is defined.

  It is also possible to define a constructor function that uses positional rather than keyword arguments. This is done by specifying the **:constructor** option as **(:constructor** *name arglist***)**, where *arglist* describes the arguments to the constructor function. A constructor of this form is known as a ***BOA constructor***, because it operates by order of arguments.

  In the simplest case, the elements of *arglist* are the slot names themselves (*not* keywords) in the order in which they are to occur as arguments. The keywords **&optional**, **&rest**, and **&aux** may also be used in *arglist*. Any **&optional** and **&aux** arguments for which no initialization forms have been specified in *arglist* are not set to **nil** as they would be in a lambda list. The initial value of any **&optional** argument for which no initialization form is specified in *arglist* is taken from the *initform* argument given for the slot description. The initial value of any **&aux** argument for which no initialization form is specified in *arglist* is undefined.

  The **:constructor** option may be used more than once. It is thus possible to define several different constructor functions for a given structure.

- The **:copier** option controls the naming of the copier function. If this option is defaulted, the name of the copier function is **copy-** followed by the structure name. The name of the copier function is entered into the package that is current at the time the **defstruct** form is processed. A symbol argument may be provided that specifies a different name for the copier function. If **nil** is specified, no copier function is defined.

- The **:include** option allows for creating a new structure that is an extension of an existing structure type by including the slots of the old structure. Both the access functions of the included structure and the access functions of the new structure may be applied to the included slots of the new structure.

  The **:include** option requires an argument that is the name of an existing structure. No more than one **:include** option may be specified in a **defstruct** form.

  If the **:type** option is specified for the new structure, the included structure must have been declared with the same type. If the **:type** option is not specified for the new structure, then it must not have been specified for the included structure. If the **:type** option is not specified, the structure name of the new structure becomes a data type name that is recognized by **typep**. In addition, the new type will be a subtype of the included structure.

  If it is desirable to override the default values or the slot options for the slots corresponding to those of the included structure, this can be done by specifying slot descriptions with the **:include** option. Each such slot description must bear the slot name or slot keyword of some slot of the included structure. If such a slot description has an accompanying *initform* argument, it overrides the initialization form of the included structure. If no *initform* argument is specified, the initial value of the slot is undefined in the new structure. A slot that is writable in the included structure may be made read-only in the new structure. A read-only slot of the included structure may not, however, be made writable. A type may be specified for a slot if and only if it is the same as, or a subtype of, the type specified in the included structure.

- The **:initial-offset** option is used in conjunction with the **:type** option. The argument to **:initial-offset** must be a nonnegative integer. It specifies that a certain number of slots in the representation of the structure are to be skipped before allocating the component slots.

  If the **:named** option is also specified, the slots skipped occur after the slot used by the **:named** option. If the **:include** option is also specified, the number of slots required by the included structure are skipped; then those specified by the **:initial-offset** argument are skipped. The following slots are then allocated to the *including* structure.

- The **:named** option specifies that the structure is named. If the **:type** option is not specified, the structure is always named; the structure name is part of the data type system and is therefore recognized by **typep**. In this case, the function **type-of**, when applied to an instance of this structure, returns the structure name.

  If the **:type** option is specified, the structure is not named unless the **:named** option is given. If the **:named** option is given, the first slot in the representation of the structure contains the structure name, so it will be possible to obtain the structure name from an instance of the structure. The structure name, however, will not be part of the data type system. It will not be recognized by **typep**, and **type-of** will return the type specifier for the structure.

  If the **:type** option is specified and the structure is not named, the structure name is not part of the data type system. It will not be recognized by **typep**, and **type-of** will return the type specifier for the structure.

- The **:predicate** option controls the naming of the type predicate. If the argument is defaulted, the predicate name is formed by adding the suffix **-p** to the name of the structure. The predicate name is entered into the package that is current at the time the **defstruct** macro is processed. Note that a predicate function can only be defined if the structure is named. If the argument to **:predicate** is specified as **nil**, no predicate function is defined. If the **:type** option is specified but the **:named** option is not, **:predicate** must be either unspecified or **nil**, and no predicate is defined.

- The **:print-function** option controls the printing of the structure. The argument to the print function is a function of three arguments that may be used with the **function** special form. The arguments correspond to the structure to be printed, the stream to which output is to be sent, and an integer indicating the current print level depth. The print function is expected to observe the values of the printer control variables. The **:print-function** option may only be specified if the **:type** option is not.

  Pretty-printing is not possible with a user-defined print function that calls the Common Lisp functions **write**, **prin1**, **print**, **pprint**, **princ**, **write-to-string**, **prin1-to-string**, or **princ-to-string**. These functions write the printed representation of Lisp objects to an output stream. Pretty-printing is possible with a user-defined print function that calls such Common Lisp functions as **write-char** or **write-string**.

  If neither the **:print-function** option nor the **:type** option is specified, the structure will be printed using the #S syntax. The ***print-structure*** variable of Sun Common Lisp provides the ability to print structures in a terse format rather than in the standard #S notation. If ***print-structure*** is set to **nil**, all structures are printed in this terse format. Structures printed in the terse format cannot be read back in by the Lisp reader.

- The **:type** option controls the representation of the structure. If this option is specified, the structure name will not be a type specifier recognized by **typep**. Components of the structure are stored in successive elements of the representation in the order of their specification in the **defstruct** form. The argument to **:type** must be one of the following:
  - If the argument to **:type** is specified as `vector`, the structure is represented as a simple general vector, and components are stored as successive vector elements. If the **:named** option is specified, the first component of the structure occurs as the second vector element (at offset 1 from the start of the vector); otherwise it is the first.
  - If the argument to **:type** is specified as `(vector` *element-type*`)`, the structure is represented as a vector, and components are stored as successive vector elements. If the **:named** option is specified, the first component of the structure occurs as the second vector element (at offset 1 from the start of the vector); otherwise it is the first. All components must be of a type compatible with *element-type*. The **:named** option may be specified only if a symbol may be stored in a vector of this type.
  - If the argument to **:type** is specified as `list`, the structure is represented as a list. If the **:named** option is specified, the first component of the structure occurs as the second element of the list; otherwise it is the first element.

Categories of Operations

The defstruct macro is used to define a structure.

defstruct

defstruct

Purpose: The **defstruct** macro allows the user to create and manipulate structured data types with named components. The name of the new data type is returned as a result.

Syntax: **defstruct** *name-and-options* [*documentation*] {*slot-description*}* [*Macro*]

Remarks: See the discussion on the use of the **defstruct** macro in the section "About Structures." The complete syntax for **defstruct** is shown in Figure 19–1.

Examples:

```
;;;
;;; Example 1
;;; define town structure type
;;; area, watertowers, firetrucks, population, elevation are its components
;;;
> (defstruct town
             area
             watertowers
             (firetrucks 1 :type fixnum)    ;an initialized attribute
             population
             (elevation 5128 :read-only t)) ;an attribute that can't
                                            ;be changed
TOWN
> (setq town1 (make-town :area 0 :watertowers 0))
                                            ;create a town instance
#S(TOWN AREA 0 WATERTOWERS 0 FIRETRUCKS 1 POPULATION NIL ELEVATION 5128)
> (town-p town1)                            ;town's predicate recognizes
                                            ;the new instance
T
> (town-area town1)                         ;new town's area is as
                                            ;specified by make-town
0
> (town-elevation town1)                    ;new town's elevation has
                                            ;initial value
5128
> (setf (town-population town1) 99)         ;setf recognizes access
                                            ;function
99
> (town-population town1)
99
```

```
> (setq town2 (copy-town town1))                  ;copier function makes
                                                  ;a copy of town1
#S(TOWN AREA 0 WATERTOWERS 0 FIRETRUCKS 1 POPULATION 99 ELEVATION 5128)
> (= (town-population town1) (town-population town2))
T
> (setq town3 (make-town :area 0 :watertowers 3 :elevation 1200))
#S(TOWN AREA 0 WATERTOWERS 3 FIRETRUCKS 1 POPULATION NIL ELEVATION 1200)
                                                  ;since elevation is a
                                                  ;read-only slot, its
                                                  ;value can be set only
                                                  ;when the structure is
                                                  ;created
;;;
;;; Example 2
;;; define clown structure type
;;; this structure uses a nonstandard access prefix
;;;
> (defstruct (clown (:conc-name bozo-))
             (nose-color 'red)
             frizzy-hair-p polkadots)
CLOWN
> (setq funny-clown (make-clown))
#S(CLOWN NOSE-COLOR RED FRIZZY-HAIR-P NIL POLKADOTS NIL)
> (bozo-nose-color funny-clown)                   ;use nonstandard accessor
                                                  ;name
RED
> (defstruct (clown (:constructor make-up-clown) ;redefine using other
             (:copier clone-clown)                ;customizing keywords
             (:predicate is-a-bozo-p))
             nose-color frizzy-hair-p polkadots)
CLOWN
> (fboundp 'make-up-clown)                        ;custom constructor now
                                                  ;exists
T
;;;
;;; Example 3
;;; define a vehicle structure type
;;; then define a truck structure type that includes
;;; the vehicle structure
;;;
> (defstruct vehicle name year (diesel t :read-only t))
VEHICLE
> (defstruct (truck (:include vehicle (year 79)))
             load-limit
             (axles 6))
TRUCK
```

```
> (setq x (make-truck :name 'mac :diesel t :load-limit 17))
#S(TRUCK NAME MAC YEAR 79 DIESEL T LOAD-LIMIT 17 AXLES 6)
> (vehicle-name x)                           ;vehicle accessors work
                                             ;on trucks
MAC
> (vehicle-year x)                           ;default taken from :include
                                             ;clause
79
> (defstruct (pickup (:include truck))      ;pickup type includes truck
             camper long-bed four-wheel-drive)
PICKUP
> (setq x (make-pickup :name 'king :long-bed t))
#S(PICKUP NAME KING YEAR 79 DIESEL T LOAD-LIMIT NIL AXLES 6 CAMPER NIL
LONG-BED T FOUR-WHEEL-DRIVE NIL)
> (pickup-year x)                            ;:include default inherited
79
;;;
;;; Example 4
;;; use of BOA constructors
;;;
> (defstruct (dfs-boa                        ;BOA constructors
               (:constructor make-dfs-boa (a b c))
               (:constructor create-dfs-boa
                 (a &optional b (c 'cc) &rest d &aux e (f 'ff))))
             a b c d e f)
DFS-BOA
> (setq x (make-dfs-boa 1 2 3))              ;a, b, and c set by position,
                                             ;and the rest are nil
#S(DFS-BOA A 1 B 2 C 3 D NIL E NIL F NIL)
> (dfs-boa-a x)
1
> (setq x (create-dfs-boa 1 2))              ;a and b set, c and f defaulted
#S(DFS-BOA A 1 B 2 C CC D NIL E NIL F FF)
> (dfs-boa-b x)
2
> (eq (dfs-boa-c x) 'cc)
T
> (setq x (create-dfs-boa 1 2 3 4 5 6))      ;a, b, and c set, and the rest
                                             ;are collected into d
#S(DFS-BOA A 1 B 2 C 3 D (4 5 6) E NIL F FF)
> (dfs-boa-d x)
(4 5 6)
```

Chapter 20. Streams

Chapter 20. Streams

About Streams

Streams are Common Lisp objects from which data can be read and to which data can be sent. The operations that can be performed on a stream depend on what type of stream it is. A stream may be input-only, output-only, or bidirectional. It may be a character stream or a binary stream.

There are several stream-value variables that are used by default by many Common Lisp system functions. These are known as ***standard streams***. The variables ***standard-input***, ***standard-output***, ***debug-io***, ***error-io***, ***query-io***, ***terminal-io***, and ***trace-output*** specify standard streams. A ***synonym stream*** associates a symbol with a stream. Any operation performed on the synonym stream is performed on the stream to which this symbol is bound. The streams ***standard-input***, ***standard-output***, ***debug-io***, ***error-io***, ***query-io***, and ***trace-output*** are all initially synonym streams of ***terminal-io***.

The use of streams is closely connected to the file system. Streams may also be created through the file system constructs for opening files.

The interaction between streams and the file system is discussed in the chapter "File System Interface." The chapter "Input/Output" discusses the use of streams in the context of the input/output system.

Categories of Operations

This section groups operations on streams according to functionality.

Data Type Predicates

streamp

This predicate determines whether an object is a stream.

Standard Streams

debug-io	***standard-output***
error-output	***terminal-io***
query-io	***trace-output***
standard-input	

These variables specify standard streams.

Stream Predicates

input-stream-p	**output-stream-p**

These predicates test properties of streams.

Creating New Streams

make-broadcast-stream	**make-string-output-stream**
make-concatenated-stream	**make-synonym-stream**
make-echo-stream	**make-two-way-stream**
make-string-input-stream	

These functions create new streams. File system constructs for opening files may also be used to create streams.

General Operations on Streams

close	**stream-element-type**

These functions provide operations that are common to all streams. More specific stream operations are also provided by the file system and the input/output system.

Operations on Stream Data

get-output-stream-string	**with-open-stream**
with-input-from-string	**with-output-to-string**

These functions provide operations on stream data.

close

Purpose: The function **close** closes its *stream* argument. Closing a stream means that it may no longer be used in input or output operations.

Syntax: **close** *stream* `&key :abort` [*Function*]

Remarks: Even if a stream is closed, it is still possible to perform query operations upon it.

An abnormal termination of the use of the stream may be indicated by specifying a non-**nil** value for the **:abort** argument. In this case the system tries to undo any side effects that resulted from the creation of the stream.

Examples:

```
> (setq s (make-broadcast-stream))
#<Stream COMPOSITE-STREAM 101DB3BB>
> (close s)
NIL
> (output-stream-p s)
T
```

See Also: **open**

debug-io

Purpose: The value of the variable ***debug-io*** is a bidirectional stream that is to be used for interactive debugging.

Syntax: ***debug-io*** [*Variable*]

Remarks: Frequently ***debug-io*** is bound to the same stream as ***query-io***.

Care should be exercised when redirecting ***debug-io***, since it is the stream used for handling errors.

error-output

Purpose: The value of the variable ***error-output*** is an output stream that is to be used for error messages.

Syntax: ***error-output*** [*Variable*]

Remarks: Frequently ***error-output*** is bound to the same stream as ***standard-output***.

Examples:

```
> (with-output-to-string(out)
    (let ((*error-output* out))
      (warn "this string is sent to *error-output*")))
";;; Warning: this string is sent to *error-output*
"
```

get-output-stream-string

Purpose: The function **get-output-stream-string** operates on a stream produced by **make-string-output-stream**. It returns a string containing all the characters sent to that stream since the last time **get-output-stream-string** was called on it. The string output stream is reset after each call.

Syntax: **get-output-stream-string** *string-output-stream* [*Function*]

Examples:

```
> (setq a-stream (make-string-output-stream)
        a-string "abcdefghijklm")
"abcdefghijklm"
> (write-string a-string a-stream)
"abcdefghijklm"
> (get-output-stream-string a-stream)
"abcdefghijklm"
> (get-output-stream-string a-stream)
""
```

See Also: **make-string-output-stream**

input-stream-p

Purpose: The predicate **input-stream-p** is true if its *stream* argument may be used for input operations; otherwise it is false.

Syntax: **input-stream-p** *stream* [*Function*]

Examples:

```
> (input-stream-p *standard-input*)
T
> (input-stream-p (make-broadcast-stream))
NIL
```

make-broadcast-stream

Purpose: The function **make-broadcast-stream** creates and returns an output stream.

Any output that is sent to this stream is sent to all of the argument *streams*. The result that is returned by performing any operation on new broadcast stream is the result returned by performing it on the last of the argument streams.

Syntax: **make-broadcast-stream** `&rest` *streams* [*Function*]

Remarks: Only those operations that may be performed on all of the argument streams may be performed on the broadcast stream.

If no argument streams are specified, all output is discarded.

Examples:

```
> (setq a-stream (make-string-output-stream)
        b-stream (make-string-output-stream))
#<Stream STRING-STREAM 101DC8B3>
> (format (make-broadcast-stream a-stream b-stream)
          "this will go to both streams")
NIL
> (get-output-stream-string a-stream)
"this will go to both streams"
> (get-output-stream-string b-stream)
"this will go to both streams"
```

make-concatenated-stream

Purpose: The function **make-concatenated-stream** creates and returns an input stream. Input is taken from each argument stream in turn until an end-of-file is reached on that stream.

Syntax: **make-concatenated-stream** `&rest` *streams* [*Function*]

Remarks: If no argument streams are specified, the result is an empty stream. Any attempt to read input from such a stream results in an end-of-file condition.

Examples:

```
> (read (make-concatenated-stream
         (make-string-input-stream "1")
         (make-string-input-stream "2")))
12
```

make-echo-stream

Purpose: The function **make-echo-stream** creates and returns a bidirectional stream. This stream takes its input from *input-stream* and sends its output to *output-stream*. Any input that is taken from *input-stream* is echoed to *output-stream*.

Syntax: **make-echo-stream** *input-stream output-stream* [*Function*]

Examples:

```
> (let ((out (make-string-output-stream)))
    (with-open-stream
       (s (make-echo-stream
           (make-string-input-stream "this-is-read-and-echoed")
           out))
      (read s)
      (format s " * this-is-direct-output")
      (get-output-stream-string out)))
"this-is-read-and-echoed * this-is-direct-output"
```

make-string-input-stream

Purpose: The function **make-string-input-stream** creates and returns an input stream. This stream supplies the characters in the *string* argument in the order in which they occur in the string. The characters supplied may be restricted to those contained in a substring of the *string* argument by specifying the *start* and *end* arguments.

Syntax: **make-string-input-stream** *string* `&optional` *start end* [*Function*]

Remarks: The *start* and *end* arguments take integer values that specify offsets into the original strings. The *start* argument marks the beginning position of the substring; the *end* argument marks the position following the last element of the substring. The start value defaults to 0; the end value defaults to the length of the string.

Examples:

```
> (read (make-string-input-stream "prefixtargetsuffix" 6 12))
TARGET
```

See Also: **with-input-from-string**

make-string-output-stream

Purpose: The function **make-string-output-stream** creates and returns an output stream. This stream accumulates the output sent to it for use by the function **get-output-stream-string**.

The optional *string* argument may be used to specify a string from which the output stream is to be built. If the optional string is supplied, it must be a string with a fill pointer. The output is directed to the point indicated by the fill pointer, which is increased incrementally.

Syntax: **make-string-output-stream** `&optional` *string* [*Function*]

Remarks: The optional *string* argument is an extension to Common Lisp.

Examples:

```
> (setq s (make-string-output-stream))
#<Stream STRING-STREAM 101DCA43>
> (format s "output 1~%")
NIL
> (format s "output 2~%")
NIL
> (get-output-stream-string s)
"output 1
output 2
"
```

See Also: **get-output-stream-string**

with-output-to-string

make-synonym-stream

Purpose: The function **make-synonym-stream** creates and returns a synonym stream. Any operations performed on this stream are performed on the stream that is currently the value of the variable named by *symbol*.

Syntax: **make-synonym-stream** *symbol* [*Function*]

Remarks: If the variable *symbol* is rebound, any stream operations are performed on the stream to which it is rebound.

Examples:

```
> (setq a-stream (make-string-input-stream "a-stream")      ;implemented
        b-stream (make-string-input-stream "b-stream"))     ;internally as
#<Stream BUFFERED-STREAM 101DD00B>                          ;buffered streams
> (setq s-stream (make-synonym-stream 'c-stream))
#<Stream SYNONYM-%STREAM 101DD1FB>
> (setq c-stream a-stream)
#<Stream BUFFERED-STREAM 101DCE73>
> (read s-stream)
A-STREAM
> (setq c-stream b-stream)
#<Stream BUFFERED-STREAM 101DD00B>
> (read s-stream)
B-STREAM
```

make-two-way-stream

Purpose: The function **make-two-way-stream** creates and returns a bidirectional stream that takes its input from *input-stream* and sends its output to *output-stream.*

Syntax: **make-two-way-stream** *input-stream output-stream* [*Function*]

Examples:

```
> (with-output-to-string (out)
    (with-input-from-string (in "input...")
      (let ((two (make-two-way-stream in out)))
        (format two "output...")
        (setq what-is-read (read two)))))
"output..."
> what-is-read
INPUT...
```

output-stream-p

Purpose: The predicate **output-stream-p** is true if its *stream* argument may be used for output operations; otherwise it is false.

Syntax: **output-stream-p** *stream* [*Function*]

Examples:

```
> (output-stream-p *terminal-io*)
T
> (output-stream-p (make-concatenated-stream))
NIL
```

query-io

Purpose: The value of the variable ***query-io*** is a bidirectional stream that is to be used to ask the user questions and to receive his answers.

Syntax: ***query-io*** [*Variable*]

See Also: **y-or-n-p**

yes-or-no-p

standard-input

Purpose: The value of the variable ***standard-input*** is a stream that is to be used for input. Many system functions use this stream as a default for input operations.

Syntax: ***standard-input*** [*Variable*]

Examples:

```
> (with-input-from-string (*standard-input* "1001")
    (+ 990 (read)))
1991
```

standard-output

Purpose: The value of the variable ***standard-output*** is a stream that is to be used for output. Many system functions use this stream as a default for output operations.

Syntax: ***standard-output*** *[Variable]*

Examples:

```
> (progn (setq out (with-output-to-string (*standard-output*)
                     (print "print and format t send things to")
                     (format t "*standard-output* now going to a string")))
         (values))
> out
"
\"print and format t send things to\" *standard-output* now going to a
string"
```

stream-element-type

Purpose: The function **stream-element-type** returns a type specifier that indicates the kinds of objects that may be read from or sent to the given stream.

Syntax: **stream-element-type** *stream* [*Function*]

Examples:

```
> (stream-element-type *debug-io*)
STRING-CHAR
> (stream-element-type (make-concatenated-stream))
T
> (setq s (open "tempfile.temp"
                :element-type 'bit
                :if-does-not-exist :create))
#<Stream BUFFERED-STREAM 101DD62B>
> (stream-element-type s)
(INTEGER 0 1)
```

streamp

Purpose: The predicate **streamp** is true if its argument is a stream; otherwise it is false.

Syntax: **streamp** *object* [*Function*]

Examples:

```
> (streamp *terminal-io*)
(INTERACTION-STREAM)
> (streamp 1)
NIL
```

terminal-io

Purpose: The value of the variable ***terminal-io*** is a bidirectional stream that is normally connected to the keyboard and display of the user's terminal.

Syntax: ***terminal-io*** [*Variable*]

Remarks: The streams ***standard-input***, ***standard-output***, ***debug-io***, ***error-io***, ***query-io***, and ***trace-output*** are all initially synonym streams of ***terminal-io***.

Examples:

```
> (progn (setq out (with-output-to-string (*terminal-io*)
                     (format t "you won't see")
                     (print "any of this")
                     (warn "until you")
                     (format *standard-output* "evaluate the string")
                     (format *query-io* " named  out")))
         (values))
> out
"you won't see
\"any of this\"
;;; Warning: until you
evaluate the string named  out"
```

trace-output

Purpose: The value of the variable ***trace-output*** is the stream to which the **trace** function sends its output.

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

Examples:

```
> (progn (setq out (with-output-to-string (*trace-output*)
                     (trace cons)
                     (cons 1 2)
                     (untrace)))
         (values))
> out
"1 Enter CONS 1 2
1 Exit CONS (1 . 2)
"
```

with-input-from-string

Purpose: The **with-input-from-string** macro provides a construct that creates a character input stream, performs a series of operations on it, returns a value, and then closes the stream.

The *string* argument is evaluated first, and the variable *var* is bound to a character input stream that supplies characters from the resulting string. The *form* arguments are executed in order. The results of evaluating the last form are returned as the value of executing the **with-input-from-string** macro. The stream is automatically closed on exit from **with-input-from-string**.

Syntax: **with-input-from-string** (*var string* {*keyword value*}*) {*declaration*}* {*form*}* [*Macro*]

Remarks: The **:index**, **:start**, and **:end** keyword arguments may be used with **with-input-from-string**.

The **:index** argument must specify a generalized variable acceptable to the macro **setf**. If the **with-input-from-string** macro terminates normally, this location is updated to contain the index value that indicates the first character not read in the string.

If the **:start** and **:end** keyword arguments are specified, only the substring they delimit is involved in the operation. The **:start** and **:end** keyword arguments take integer values that specify offsets into the original string. The **:start** argument marks the beginning position of the string; the **:end** argument marks the position following the last element of the string. The start value defaults to 0; the end value defaults to the length of the string.

Examples:

```
> (with-input-from-string (s "XXX1 2 3 4xxx"
                              :index ind
                              :start 3 :end 10)
     (+ (read s) (read s) (read s)))
6
> ind
9
```

See Also: **make-string-input-stream**

with-open-stream

Purpose: The **with-open-stream** macro provides a construct that takes a stream, performs a series of operations on it, returns a value, and then closes the stream.

The *stream* argument is evaluated, and the variable *var* is bound to the resulting stream. The *form* arguments are executed in order. The results of evaluating the last form are returned as the result of executing the **with-open-stream** macro. The stream is automatically closed on exit from **with-open-stream**, even if the exit is abnormal.

Syntax: **with-open-stream** (*var stream*) {*declaration*}* {*form*}* [*Macro*]

Examples:

```
> (with-open-stream (s (make-string-input-stream "1 2 3 4"))
    (+ (read s) (read s) (read s)))
6
```

See Also: **close**

with-output-to-string

Purpose: The **with-output-to-string** macro provides a construct that creates a character output stream, performs a series of operations that may send results to this stream, and then closes the stream.

The variable *var* is bound to a character output stream, and the output to this stream is saved in a string. The optional *string* argument may be used to specify a string from which the output stream is to be built. If the *string* argument is specified, it must be a string with a fill pointer. The stream output is then directed to the point indicated by the fill pointer, which is increased incrementally.

The *form* arguments are executed in order.

If no *string* argument is specified, **with-output-to-string** returns a string containing all of the accumulated stream output.

If a *string* argument was provided, **with-output-to-string** returns as its value the results of evaluating the last *form* argument.

The stream is automatically closed on exit from the **with-output-to-string** macro.

Syntax: **with-output-to-string** (*var* [*string*]) {*declaration*}* {*form*}* [*Macro*]

Remarks: If the specified string is adjustable, the appending of characters occurs as if **vector-push-extend** were used; if it is not adjustable, the effect is the same as if **vector-push** were used.

Examples:

```
> (setq fstr (make-array '(0)
                         :element-type 'string-char
                         :fill-pointer 0
                         :adjustable t))
""
> (with-output-to-string (s fstr)
    (format s "here's some output")
    (input-stream-p s))
NIL
> fstr
"here's some output"
```

See Also: **make-string-output-stream**

vector-push

vector-push-extend

Chapter 21. Input/Output

Chapter 21. Input/Output

About Input/Output

All input/output (I/O) in Common Lisp is performed with streams. Although binary input and output streams are available, most I/O is done with character streams.

The principal I/O operations read and write the printed representations of arbitrary Lisp objects. The **format** function performs complex formatting of output data.

This chapter presents the operations and constructs for I/O and tables of standard character syntax types, standard constituent character attributes, and standard # dispatching macro character syntax. It also includes a complete description and summary of the use of the format facility.

The Printed Representation of Common Lisp Objects

Common Lisp provides a ***printed representation*** for all objects. Such a representation is a text sequence that identifies the object. An object may have more than one printed representation: an integer, for instance, may have a different representation for each possible numeric base.

Output functions such as **write** transmit the characters of an object's printed representation to an output stream, and input functions such as **read** receive characters from an input stream and build the object that is specified by the printed representation.

Each Common Lisp data type has its own printed representation. Within most printed representations, variations are specified by the values of certain global variables. These variables are ***print-escape***, ***print-radix***, ***print-base***, ***print-circle***, ***print-pretty***, ***print-level***, ***print-length***, ***print-case***, ***print-gensym***, ***print-array***, and ***print-structure***.

Some of these variations cause abbreviated representations to be printed. Hence, not all printed representations can be read back in. The reading of printed representations is discussed in the section "Reading the Representations of Lisp Objects."

The printed representations of certain objects may be very obvious (those of integers, for instance), but even those objects that have no obvious printed forms have printed representations in Common Lisp. For more complex data types, such as arrays or structures, the number-sign character (#) begins a special printed representation in which the character following the # indicates the data type and the following characters describe the specific object.

The printed representations for the different data types are described below. If an object has a specific data type that is a subtype of a more general data type, the object is printed as the more specific type.

Integers

An integer is printed as a sequence of digits in the base specified by the variable ***print-base***. If the integer is negative, the sequence of digits is preceded by a minus sign. If the variable ***print-radix*** is non-**nil**, a radix indicator is also printed. For the decimal base, the radix indicator is a decimal point following the number. For other bases, the radix indicator is one of the following forms preceding the number: #o (octal), #x (hexadecimal), #b (binary), or #*nr* (other base *n*, which is printed in decimal).

Ratios

Ratios are always printed in lowest reduced form, as the numerator, a slash (/), and then the denominator. No spaces are included. In a negative ratio, the numerator is preceded by a minus sign. The numerator and denominator are printed in the base specified by ***print-base***. In addition, if ***print-radix*** is non-**nil**, the ratio begins with a radix indicator. The radix indicator is one of the following: `#10r` (decimal), `#o` (octal), `#x` (hexadecimal), `#b` (binary), or `#nr` (other base *n*, which is printed in decimal). Note that ratios are never printed with the decimal radix indicator used for integers, which is a trailing decimal point.

Floating-Point Numbers

Floating-point numbers are printed as one or more digits on each side of a decimal point, sometimes followed by an exponent. If the number is negative, it is preceded by a minus sign.

If the magnitude of a floating-point number is zero or is greater than or equal to 10^{-3} and less than 10^7, it is printed as the integer part (one to seven digits), a decimal point, and then the fractional part (one to three digits).

Nonzero magnitudes less than 10^{-3} and greater than or equal to 10^7 are printed as numbers between 1 (inclusive) and 10 (exclusive) times a power of ten. One digit is printed, then a decimal point, a fractional part of one or more digits, the exponent marker `E`, and finally the power of ten as a decimal integer.

Complex Numbers

A complex number is printed as `#C(`*r i*`)`, where *r* is the printed representation of the number's real part and *i* is the printed representation of the number's imaginary part.

Characters

If ***print-escape*** is non-**nil**, a character is printed as `#\` followed by the character, if it is a printing character, or by the name of the character, if not. If ***print-escape*** is **nil**, a character is printed as itself.

Symbols

If ***print-escape*** is non-**nil**, a symbol is printed as its print name along with any character quoting or name qualification necessary to identify the symbol uniquely. This may include backslashes (\), vertical bars (|), a colon (:) (for keywords), a package name and one or two colons (:), or a leading #: (for uninterned symbols). The use of these quoting and qualifying characters when ***print-escape*** is non-**nil** is described in the paragraphs that follow.

If the print name could be interpreted as a potential number, backslashes or vertical bars are included to prevent such an interpretation. In this determination, it is assumed that the text would be read back in with ***read-base*** set to the value that ***print-base*** has at the time of printing.

If the symbol is in the keyword package, it is printed with a leading colon. If the symbol is not accessible in the current package, it is printed with a leading package name and one or two colons, however many are needed to identify the symbol.

A leading #: is printed if the symbol is uninterned.

A symbol is printed as just its print name if ***print-escape*** is **nil**.

In either of the cases described, lowercase letters in the print name are always printed in lowercase, but the case in which uppercase letters are printed is controlled by the global variable ***print-case***. The possible values for ***print-case*** are **:upcase**, **:downcase**, and **:capitalize**.

Lists

A true list is printed as follows: first a left parenthesis, then the elements of the list in order, and finally a right parenthesis. The list elements are separated by white space (space, tab, carriage-return, or newline characters).

A dotted list is printed as follows: first a left parenthesis, then the car of the list, a dot, the cdr of the list, and finally a right parenthesis. The dot is separated from the car and the cdr of the list by white space.

Conses are printed with list notation rather than dot notation whenever possible. The global variables ***print-level*** and ***print-length*** can be used to limit the depth of printing and the number of consecutive items printed at a single level.

Arrays

If *print-array* is non-**nil**, an array is printed with the #*n*A(...) syntax. In this case, the output starts with #*n*A, where *n* is the number of dimensions of the array, and then the contents of the array are printed in row-major order with parentheses indicating the structure of the array. The length of the top-level list printed is the size of the first dimension, and the lengths of the subsequent deeper levels are the sizes of the second dimension, the third dimension, and so on.

If the array has elements that are either bits or string characters, the deepest level printed may take the form of a bit vector or string.

The global variables *print-level* and *print-length* can be used to limit the depth of printing and the number of consecutive items printed at a single level.

If *print-array* is **nil**, an array is printed with the #<...> syntax, which identifies the array without listing the values of its elements.

Vectors

If *print-array* is non-**nil**, a vector is printed as #(and) enclosing the elements of the vector, which are separated by white space. For a vector with a fill pointer, only those elements before the fill pointer are printed. The global variables *print-level* and *print-length* can be used to limit the depth of printing and the number of consecutive items printed at a single level.

If *print-array* is **nil**, a vector is printed with the #<...> syntax, which identifies the vector without listing the values of its elements.

Bit Vectors

If *print-array* is non-**nil**, a printed bit vector consists of #*, followed by the bits in the bit vector. For a bit vector with a fill pointer, only those bits before the fill pointer are printed.

If *print-array* is **nil**, a bit vector is printed with the #<...> syntax, which identifies the bit vector without listing the values of its bits.

Strings

If *print-escape* is non-**nil**, the string is preceded and followed by a double quote ("). Any double-quote or single escape character in the string is preceded by a backslash (\).

If *print-escape* is **nil**, a string is printed as just the sequence of characters that it contains.

A string with a fill pointer is printed only up to the fill pointer.

Structures

The user can completely control the format in which a structure is printed by using the **defstruct** option **:print-function** to specify a function to be called when the structure is to be printed. If this option is not used, a default printing function is supplied that prints the structure as `#S(`*name slot1 value1 slot2 value2* `...)`, where *name* is the name of the structure, *slotj* is the name of one of the structure's slots, and *valuej* is the corresponding value. The global variables *print-level* and *print-length* can be used to limit the depth of printing and the number of consecutive items printed at a single level.

If the global variable *print-structure* is **nil**, the default printing function prints a structure with the `#<...>` syntax, which identifies the structure without listing the values of its elements. The variable *print-structure* is an extension to Common Lisp.

Pathnames

A printed pathname consists of `#P` followed immediately by the pathname enclosed in double quotes.

Random States

An object of type random state is printed as a structure with the `#S` syntax. The global variables *print-level* and *print-length* can be used to limit the depth of printing and the number of consecutive items printed at a single level.

Other Data Types

Data types that do not have a syntax in which they can be printed and subsequently read back in are printed with the `#<...>` syntax. The `#<...>` syntax describes the data type and may give some indication of the particular instance (such as a memory address where it appears). The `#<...>` syntax does not allow the object to be read back in. An object that is a hash table, a readtable, a package, a stream, or a function is printed with the `#<...>` syntax.

Reading the Representations of Common Lisp Objects

The ***reader*** is the part of the Common Lisp system that reads characters in, interprets them as the printed representations of individual objects, and constructs and returns those objects. An object may be made up of various parts, such as numbers, symbol names, form indicators like parentheses, and other special characters. The object is constructed from the input text by interpreting each character according to its syntax type.

Character Syntax Types and Readtables

The ***syntax type*** of a character determines how that character is interpreted by the reader. For instance, it indicates whether the character can appear in a symbol name or whether it can appear in a number. Every character has at any given time exactly one syntax type.

It is possible to change a single character's syntax type or the entire collection of syntax types for all characters. The association between characters and syntax types is maintained in a data object known as a ***readtable***. The user can create several readtables and switch between them as needed to alter the input syntax. Common Lisp defines a ***standard syntax*** for the interpretation of characters. This syntax is embodied in the readtable that is current when Common Lisp is started up. This standard syntax is discussed below.

The possible character syntax types are *constituent*, *whitespace*, *macro*, *single escape*, *multiple escape*, and *illegal*. Figure 21–1 lists the default syntax type of each character.

- ***Constituent characters*** are those characters used in tokens. A ***token*** is a number or a symbol name. Examples of constituent characters are letters and digits.

  A constituent character has one or more ***attributes*** that define how the character can be interpreted by the reader. These are *alphabetic*, *digit*, *package marker*, *plus sign*, *minus sign*, *dot*, *decimal point*, *ratio marker*, *floating-point exponent marker*, and *illegal*. Figure 21–2 shows the standard attributes for constituent characters. Any character with the *alphadigit* attribute in that figure is considered a *digit* if ***read-base*** is greater than that character's digit value; otherwise the character is *alphabetic*. Alphabetic constituents are those characters that can appear in a symbol name. Note that any character quoted with a preceding single escape character is treated as an alphabetic constituent, regardless of its normal syntax. In particular, constituent characters with the *illegal* attribute must be quoted in order to appear in a token.

  Normally, lowercase letters in symbol names are converted to their uppercase equivalents when the name is read. This conversion can be inhibited by the use of single or multiple escape characters, as explained below.

- ***Whitespace characters*** are used to separate tokens. The space and newline characters are examples of whitespace characters.

- A ***macro character*** triggers special parsing of subsequent input characters. When a macro character is encountered by the reader, the special function assigned to that macro character is called. This function generally parses one specially formatted object from the input stream and returns the constructed object. The macro character function may also return no values to indicate that the characters scanned by the function are being ignored (as in the case of a comment). Examples of macro characters in the standard Common Lisp syntax are the backquote and single quote characters ` and ' and the parenthesis characters (and).

  A macro character is either ***terminating*** or ***nonterminating***. The difference between terminating and nonterminating macro characters lies in what happens when such characters occur in the middle of a token. In such a location, the function associated with the nonterminating macro character is not called, and the nonterminating macro character does not terminate the token's name; it becomes part of the name as if the macro character were really a constituent character. A terminating macro character, however, terminates any token, and the macro character's function is called no matter where the character appears. The only nonterminating macro character in the standard syntax is the number-sign character #.

  Macro characters are discussed in greater detail in the section "Standard Macro Characters."

- A ***single escape character*** is used to quote the next character so that it is treated as a constituent character of alphabetic attribute no matter what the character is or which attributes it has. Furthermore, the normal conversion of lowercase letters to uppercase letters in symbol names is prevented for the quoted character. Thus, a single escape character can be used to include any character in a symbol name. In the standard Common Lisp syntax, the backslash character \ is a single escape character.

- A pair of ***multiple escape characters*** is used to quote an enclosed sequence of characters, including possible macro and whitespace characters, so that they are treated as constituent characters of alphabetic attribute with letter case preserved. Any single and multiple escape characters that are to appear in the sequence must be quoted with a single escape character. Note that a symbol name is not delimited by the multiple escape character; the symbol name can continue past a multiple escape character. In the standard Common Lisp syntax, the vertical bar character | is a multiple escape character. Thus, the symbol parsed from a|B|c is the same as that parsed from ABC or abc or |ABC|, but it is not the same as that parsed from |abc|.

- If an ***illegal character*** is encountered while a Lisp object is being read, an error is signaled. However, if an illegal character is quoted with a preceding single escape character, it is treated as an alphabetic constituent instead.

Table of Standard Character Syntax Types

character	syntax type	character	syntax type
Backspace	constituent	0–9	constituent
Tab	whitespace	:	constituent
Newline	whitespace	;	terminating macro
Linefeed	whitespace	<	constituent
Page	whitespace	=	constituent
Return	whitespace	>	constituent
Space	whitespace	?	constituent*
!	constituent*	@	constituent
"	terminating macro	A–Z	constituent
#	nonterminating macro	[	constituent*
$	constituent	\	single escape
%	constituent	]	constituent*
&	constituent	^	constituent
'	terminating macro	_	constituent
(	terminating macro	`	terminating macro
)	terminating macro	a–z	constituent
*	constituent	{	constituent*
+	constituent	|	multiple escape
,	terminating macro	}	constituent*
-	constituent	~	constituent
.	constituent	**Rubout**	constituent
/	constituent		

Figure 21–1. Standard Character Syntax Types

* The characters !, ?, [,], {, and } are constituents by default but are reserved for the user. They will never be used in the names of functions and variables defined by Common Lisp.

Table of Standard Constituent Character Attributes

constituent character	attributes	constituent character	attributes
Backspace	illegal	{	alphabetic
Tab	illegal*	}	alphabetic
Newline	illegal*	+	alphabetic, plus sign
Linefeed	illegal*	-	alphabetic, minus sign
Page	illegal*	.	alphabetic, dot, decimal point
Return	illegal*	/	alphabetic, ratio marker
Space	illegal*	A, a	alphadigit
!	alphabetic	B, b	alphadigit
"	alphabetic*	C, c	alphadigit
#	alphabetic*	D, d	alphadigit, double-float exponent marker
$	alphabetic	E, e	alphadigit, float exponent marker
%	alphabetic	F, f	alphadigit, single-float exponent marker
&	alphabetic	G, g	alphadigit
'	alphabetic*	H, h	alphadigit
(	alphabetic*	I, i	alphadigit
)	alphabetic*	J, j	alphadigit
*	alphabetic	K, k	alphadigit
,	alphabetic*	L, l	alphadigit, long-float exponent marker
0-9	alphadigit	M, m	alphadigit
:	package marker	N, n	alphadigit
;	alphabetic*	O, o	alphadigit
<	alphabetic	P, p	alphadigit
=	alphabetic	Q, q	alphadigit
>	alphabetic	R, r	alphadigit
?	alphabetic	S, s	alphadigit, short-float exponent marker
@	alphabetic	T, t	alphadigit
[	alphabetic	U, u	alphadigit
\	alphabetic*	V, v	alphadigit
]	alphabetic	W, w	alphadigit
^	alphabetic	X, x	alphadigit
_	alphabetic	Y, y	alphadigit
`	alphabetic*	Z, z	alphadigit
|	alphabetic*	**Rubout**	illegal
~	alphabetic		

Figure 21–2. Standard Constituent Character Attributes

* Characters marked by asterisks are not constituent characters in the standard syntax; these attributes apply to them only if their syntax types are changed to constituent.

Standard Macro Characters

The standard Common Lisp syntax defines several macro characters. When a macro character is encountered, the macro character function associated with it is called. That function normally reads some number of characters from the input and returns a value representing the object read. The standard macro characters are discussed below.

- ((Left parenthesis)

  A left parenthesis (`(`) marks the beginning of a list or a dotted pair. Objects are read (recursively) until a right parenthesis is encountered at the same level as the left parenthesis, and a list of the objects read is returned. Whitespace characters can be used freely or omitted before and after the left and right parentheses.

  A dot may appear by itself after some element in the list, in which case there must be precisely one element and a right parenthesis following the dot. The final element is the cdr of the last pair in the list.

-) (Right parenthesis)

  A right parenthesis (`)`) ends a list or a dotted pair. A right parenthesis can occur only as part of some particular syntactic construct that uses a left parenthesis. In any other context, a right parenthesis is handled according to the setting of the global variable ***ignore-extra-right-parens***.

- ' (Single quote)

  A single quote (`'`) is used to quote a Lisp object so that it can be manipulated as a constant. The construct `'`*form* has the same meaning as `(quote` *form*`)`.

- ; (Semicolon)

  A semicolon (`;`) begins a comment, which continues through the next newline character. A comment terminates any token currently being read but is otherwise ignored.

- " (Double quote)

  A double quote (`"`) begins a simple string. All input characters that lie between the first double-quote character and a second double quote are included in the string. If a single escape character is encountered, however, the single escape character is discarded, the character following it is included in the string (no matter what that character is), and the string continues.

- ` (Backquote)

  A backquote (`` ` ``) quotes all of a form except parts directly preceded by commas. A backquote causes an object to be created from the form following the backquote. Any subform that follows a comma is evaluated, and its value takes the place of the subform in the object. The backquote construct is read as the object that results when all such evaluations have been done. This result may or may not share any list structure with the template itself. If there are no commas in the form following a backquote, the result is the same as if the backquote had been a single quote ('). In nested backquote constructs, the innermost backquote construct is processed first.

Within a backquote construct, if a comma is immediately followed by an at-sign (`,@`), the following subform is evaluated and must produce a list. The individual items in the list are inserted in the object in place of the subform. Thus, one subform in the original template can be replaced by any number of items, depending on the length of the list resulting from the evaluation of the subform.

If a comma is followed by a dot (`,.`), the following subform is evaluated and inserted in the object. The `,.` construct is the same as the `,@` construct except that the list resulting from the evaluation of the subform may be modified.

In a backquote construct, a comma can occur inside any subform that produces a cons or a simple vector. In particular, a comma can appear inside `#(` or `#'` forms but cannot appear inside `#A` or `#S` forms. A comma can also appear immediately after a backquote, provided that the comma is not followed by an at-sign or a dot. The combinations `,@` and `,.` cannot appear immediately after a backquote or immediately after the dot in a dotted pair.

The following example shows the use of the comma by itself and the `,@` and `,.` combinations:

```
> (setf a '(r s))
(R S)
> `(1 a 2 ,a 3 ,@a 4 ,(cdr a) 5 ,@(cdr a) 6 ,.(cdr a))
(1 A 2 (R S) 3 R S 4 (S) 5 S 6 S)
```

- `,` (Comma)

A comma (`,`) causes a form within a template to be evaluated. A comma can occur only within a template quoted by backquote, as described above.

- `#` (Number-sign)

The number-sign character (#) is a dispatching macro character. The effect of a dispatching macro character is determined by the dispatch-controlling character following it. The next section describes the standard syntax for the # dispatching macro character.

Dispatching Macro Characters

A ***dispatching macro character*** is a special type of macro character. Such a character dispatches to one of many possible functions, depending on the next character read. Thus, certain two-character sequences can trigger the invocation of specific functions.

The only dispatching macro character in the standard Common Lisp syntax is the number-sign character #. In the standard syntax, the dispatching macro character # is nonterminating. That is, whenever the character # occurs in the middle of a token, it is taken as a constituent character rather than as a macro character and does not terminate the token.

Forms beginning with # are used for reading in objects of particular data types. The character following the # generally specifies the data type; that character is followed by text that specifies the value, using a data-type dependent syntax. For instance, the form `#C(2 3)` represents a complex number with a real part of 2 and an imaginary part of 3. In certain cases, an unsigned decimal integer may be used between the # and the type-specifying character. The dispatch function called for a particular dispatching macro character sequence is given three arguments: the input stream, the dispatch-controlling character, and the intervening integer. If no integer is specified, the third argument is **nil**.

The standard # dispatching macro character syntax is explained below and is summarized in Figure 21–3. In constructs where the second character is a letter, the case of the letter is not significant. Although Common Lisp may be extended to include additional # constructs, the constructs beginning with `#!`, `#?`, `#[`, `#]`, `#{`, and `#}` will never be defined in the standard syntax; they are reserved for the user.

- #\ (Character object)

  A character object is represented either by #*x*, where *x* is the character, or by #*name*, where *name* is the name of the character. The names recognized after #\ are the same as the character names recognized by the function **name-char**. When #\ is followed by a single character rather than a name, that single character must be followed by a character that is not a constituent character.

  Although any character can follow #\, the use of a name after #\ is generally preferred for representing nonprinting characters in programs. If a single character follows #\, uppercase and lowercase are distinguished and are used to represent the corresponding uppercase and lowercase letters. In a character's name, however, case is not significant.

  Bits attributes can be included in characters represented with the #\ syntax. The character or its name is preceded by one or more bit names or initials, each followed by a hyphen. For example, `#\Control-Hyper-Space` and `#\c-h-space` represent the same character. When bits attributes are specified and the single-character form is used, the character itself must be quoted if it is not an alphabetic constituent in the current readtable or if it represents a lowercase letter, for instance, `#\Meta-\a`. The names of bits attributes that can be used with the `#\` syntax are `Control`, `Meta`, `Hyper`, and `Super`.

- `#'` (Function object)

  The input sequence `#'`*function* represents the form (**function** *function*), where *function* is the printed representation of any Common Lisp object.

- `#(` (Simple vector)

  A simple vector is represented by enclosing its elements in order between `#(` and `)`. An explicit length for the vector can be specified as an unsigned decimal integer between the `#` and the `(`. If an explicit length is specified, no more than that number of objects may be enclosed. If fewer objects are enclosed than is specified by an explicit length, at least one object must be enclosed; in this case, the last enclosed object is used as the value of each of the remaining elements of the vector.

- `#*` (Bit vector)

  A simple bit vector is represented by `#*` followed by the bit vector's binary digits (each is either a 0 or a 1). An explicit length for the vector can be specified as an unsigned decimal integer between the `#` and the `*`. If an explicit length is specified, no more than that number of bits may be present. If there are fewer bits present than specified by an explicit length, at least one bit value must be specified, and the last bit value specified is used as the value of each of the remaining elements of the vector.

- `#:` (Uninterned symbol)

  An uninterned symbol is represented by `#:` followed by a symbol name containing no embedded colons. A new uninterned symbol is created each time this syntax is encountered.

- `#.` (Read-time evaluation)

  The construct `#.`*form* represents the object that is obtained by evaluating the form *form*. The computation of the intended object is done at read-time. This construct is useful for representing an object that has no other convenient printed representation.

- `#,` (Load-time evaluation)

  The construct `#,`*form* represents the object that is obtained by evaluating the form *form*. The computation of the intended object is done at read-time unless it is the compiler that is doing the reading. When the compiler sees this construct, it arranges for the form to be evaluated at load-time.

  In interpreted code, `#.` and `#,` are treated the same, but in compiled code, `#.` causes *form* to be evaluated at compile-time and `#,` causes *form* to be evaluated at load-time.

- `#B` (Binary rational)

  The construct `#B`*rational* represents a rational number expressed in binary (base 2).

- `#O` (Octal rational)

  The construct `#O`*rational* represents a rational number expressed in octal (base 8).

- `#X` (Hexadecimal rational)

  The construct `#X`*rational* represents a rational number expressed in hexadecimal (base 16).

- `#`*n*`R` (Radix rational)

  The construct `#`*n*`R`*rational* represents a rational number expressed in base *n*, where *n* must be between 2 and 36 inclusive.

- `#C` (Complex number)

  The construct `#C(`*r i*`)` represents a complex number whose real part is *r* and whose imaginary part is *i*.

- `#`*n*`A` (Array)

  The construct `#`*n*`A`*object* creates an *n*-dimensional array whose initial contents are specified by *object*.

- `#S` (Structure)

  The construct `#S(`*name slot1 value1 slot2 value2* ...`)` represents a structure. Here, *name* must be the name of a defined structure that has a constructor function. The constructor function is called with the specified slot values, and the result returned by the constructor function is the result returned when the `#S` construct is read. It is not necessary to start each slot name with a colon.

- `#`*n*`=` (Object label)

  The construct `#`*n*`=`*object* is read as *object*, but it also labels that object with the unsigned decimal integer *n*. The subsequent use of the construct `#`*n*`#` with the same value of *n* represents this identical *object*. The label applies within the expression being read by the outermost call to **read** and must be unique within that expression.

- `#`*n*`#` (Label reference)

  The construct `#`*n*`#` represents the object with the label *n* in the current expression. The label must have been defined by an earlier use of `#`*n*`=` in the same expression. The `#`*n*`#` syntax is used in representing a construct that has a shared or circular element.

- #+ (Read-time conditional)

  The construct #+*feature form* causes the form *form* to be read only if *feature* specifies a true condition. If *feature* specifies a false condition, *form* is read with the global variable ***read-suppress*** bound to a non-**nil** value. This results in the form being skipped over.

  The construct *feature* must be the printed representation of either a symbol or a list. If it is a symbol, it specifies a true condition if and only if that symbol is an element of the list that is the value of the global variable ***features***. If *feature* is a list, it must be composed of the logical operators **and**, **or**, and **not** applied to other *feature* expressions; in this case, a true condition is specified if the logical combination is true.

  The #+ construct can be used in conjunction with the ***features*** list to select the portions of a program that are to be read or compiled.

- #- (Read-time negative conditional)

  The construct #-*feature form* has the same effect as #+(not *feature*) *form*.

- #| (Balanced comment)

  The construct #|... |# represents a comment and is ignored when read. The comment may contain anything, but occurrences of #| and |# must be balanced. Comments may thus be nested. The #|... |# construct can be used to disable a portion of a program by turning it into a balanced comment.

- #< #) #*whitespace* #**Backspace** (Forced error)

  If # is followed by one of the characters <,), **Backspace**, **Tab**, **Newline**, **Page**, **Return**, or **Space**, an error is signaled. These constructs prevent attempts to read back in objects with no valid printed representation or objects whose printed representation has been abbreviated.

Table of Standard # Dispatching Macro Character Syntax

character combination	purpose	character combination	purpose
#Backspace	signals error	#{	undefined*
#Tab	signals error	#}	undefined*
#Newline	signals error	#+	read-time conditional
#Linefeed	signals error	#-	read-time conditional
#Page	signals error	#.	read-time evaluation
#Return	signals error	#/	undefined
#Space	signals error	#A, #a	array
#!	undefined*	#B, #b	binary rational
#"	undefined	#C, #c	complex number
##	reference to #= label	#D, #d	undefined
#$	undefined	#E, #e	undefined
#%	undefined	#F, #f	undefined
#&	undefined	#G, #g	undefined
#'	function abbreviation	#H, #h	undefined
#(	simple vector	#I, #i	undefined
#)	signals error	#J, #j	undefined
#*	bit vector	#K, #k	undefined
#,	load-time evaluation	#L, #l	undefined
#:	uninterned symbol	#M, #m	undefined
#;	undefined	#N, #n	undefined
#<	signals error	#O, #o	octal rational
#=	labels following object	#P, #p	undefined
#>	undefined	#Q, #q	undefined
#?	undefined*	#R, #r	radix-*n* rational
#@	undefined	#S, #s	structure
#[	undefined*	#T, #t	undefined
#\	character object	#U, #u	undefined
#]	undefined*	#V, #v	undefined
#^	undefined	#W, #w	undefined
#_	undefined	#X, #x	hexadecimal rational
#`	undefined	#Y, #y	undefined
#|	balanced comment	#Z, #z	undefined
#~	undefined	**#Rubout**	undefined

Figure 21–3. Standard # Dispatching Macro Character Syntax

* The dispatching macro character pairs `#!`, `#?`, `#[`, `#]`, `#{`, and `#}` are reserved for the user and will never be defined in the standard Common Lisp syntax. The combinations `#0`, `#1`, `#2`, `#3`, `#4`, `#5`, `#6`, `#7`, `#8`, and `#9` occur only when integers are used as infix arguments. They cannot be defined.

Formatted Output

The **format** function and certain other text output functions accept as an argument a ***format control string*** that specifies formatted text to be generated. Such a string is made up of simple text and embedded directives. The simple text is written to the indicated stream (for example, the user's display); each embedded directive specifies further text output that is to appear at the corresponding point within the simple text. Directives are carried out in the order in which they appear within the format control string.

Format Control Directives

The basic format control directive consists of a tilde (`~`) followed by a directive character that specifies the type of output to be generated. In the general form, a directive may accept parameters and modifiers between the tilde and the directive character and may use arguments from the function call form in which it occurs. If any parameters are specified, they must precede any modifiers. The meanings of the parameters and modifiers vary from directive to directive and are defined in the descriptions of the syntax of individual directives. A directive must not be given more parameter values than its syntax allows.

A format control string parameter is either an integer or a character object, depending on how the parameter is used by the directive. An integer parameter is specified as an optionally signed decimal integer. A character parameter is specified as a two-character sequence consisting of a single-quote character (`'`) followed by the character that is to be the parameter. If multiple parameters are supplied to one directive, they are separated by commas. A default value is supplied for any parameter that is not specified. When parameters at the end of the parameter list are defaulted, trailing commas may also be omitted. For example, the directive `~12,,'!R` has an integer parameter followed by a defaulted parameter and then a character parameter whose value is the exclamation-point character. The fourth parameter in this example is omitted, as is the comma that would have preceded it.

The characters **V**, **v**, or **#** can be used in place of an actual integer or character value for a parameter. If **V** (or **v**) is used, the value of the next unused argument of the current function call form is taken to be the value of the parameter. This value should be an integer or character object, whichever is appropriate for the directive, or it can be **nil** to default the given parameter. If **#** is used, the number of remaining arguments of the current function call form is taken to be the value of the next parameter.

A format control string modifier is a single character, either a colon (**:**) or an at-sign (**@**). A directive can contain neither, one, or both of these modifiers. The meanings of the four combinations depend on the specific directive character. For instance, `~(`*text*`~)` causes *text* to be written in lowercase, whereas `~:@(`*text*`~)` causes *text* to be written in uppercase. A directive may only be given modifiers in a combination allowed by its syntax; the order of the two modifiers, however, does not matter.

Certain directives use arguments from the current function call form (most commonly a call to **format**). For instance, several directives cause particular printed representations of the next argument to be written. Function arguments are supplied to directives in order as needed; however, the **~*** directive can be used to select the starting argument position from which subsequent directives get their arguments. At least as many arguments as the directive requires must be supplied.

The Syntax of Format Control Directives

The individual directives that are available for use in format control strings are described below. In these descriptions, the directive is accompanied by a brief descriptive title to help the user remember what the directive does. The second line of each description presents the syntax of the general form of the directive. It is followed by an explanation of the use of the directive and its parameters and modifiers.

- **~A** (ASCII)

  ~*mincol*, *colinc*, *minpad*, *padchar*:**@A**

  The **~A** directive causes the printed representation of the next available argument of the function call form to be written with no escape characters, as if ***print-escape*** were bound to **nil**.

  - The *mincol* parameter specifies the minimum number of columns (characters) that the output is to occupy. Its default value is 0.
  - The *minpad* parameter specifies the minimum number of padding characters to be used. Its default value is 0.
  - After the first *minpad* characters have been written, padding characters are written in increments specified by the *colinc* parameter until the total output occupies at least *mincol* columns. The default value of the *colinc* parameter is 1.
  - The *padchar* parameter specifies the padding character to be used. Its default value is the space character.
  - The : modifier controls the format of any argument that is **nil**. If the argument is **nil**, it is written as `nil` unless the : modifier is used, in which case it is written as `()`. In either case, if the argument is a list or any other structured object, any null element within it is written as `nil` whether or not the : modifier is present.
  - The **@** modifier controls the placement of padding characters. Any necessary padding is normally inserted on the right of the output; it is thus left-justified. Use of the **@** modifier, however, causes all padding to be inserted on the left; the output is then right-justified.

- The : and @ modifiers may be used separately or in combination.

- **~S** (S-expression)

  ~mincol, colinc, minpad, padchar:**@S**

  The **~S** directive causes the printed representation of the next available argument of the function call form to be written with escape characters included in the output, as if ***print-escape*** were bound to **t**.

  - The *mincol* parameter specifies the minimum number of columns (characters) that the output is to occupy. Its default value is 0.
  - The *minpad* parameter specifies the minimum number of padding characters to be used. Its default value is 0.
  - After the first *minpad* characters have been written, padding characters are written in increments specified by the *colinc* parameter until the total output occupies at least *mincol* columns. The default value of the *colinc* parameter is 1.
  - The *padchar* parameter specifies the padding character to be used. Its default value is the space character.
  - The : modifier controls the format of any argument that is **nil**. If the argument is **nil**, it is written as `nil` unless the : modifier is used, in which case it is written as `()`. In either case, if the argument is a list or any other structured object, any null element within it is written as `nil` whether or not the : modifier is present.
  - The @ modifier controls the placement of padding characters. Any necessary padding is normally inserted on the right of the output; it is thus left-justified. Use of the @ modifier, however, causes all padding to be inserted on the left; the output is then right-justified.
  - The : and @ modifiers may be used separately or in combination.

- **~D** (Decimal)

  ~mincol, padchar, commachar:**@D**

  The **~D** directive causes the printed representation of the next available argument of the function call form to be written in decimal (base 10), without a trailing decimal point. If the argument is not an integer, it is written in **~A** format using the decimal base.

  - The *mincol* parameter specifies the minimum width of output in characters. Its default value is 0.
  - The *padchar* parameter specifies the padding character to be used to achieve the minimum output width. Padding characters are inserted at the left of the output. The default value of *padchar* is the space character.

- The : modifier causes the *commachar* parameter to be written between every group of three characters. The default value of *commachar* is the comma character (,).
- The @ modifier causes the integer's sign to be written. If this modifier is not specified, the sign is written only if the integer is negative.
- The : and @ modifiers may be used separately or in combination.

■ **~B** (Binary)

~mincol, padchar, commachar **:@B**

The **~B** directive is identical to the **~D** directive except that the integer argument is written in base 2 instead of in decimal.

The **~B** directive causes the printed representation of the next available argument of the function call form to be written in binary. If the argument is not an integer, it is written in binary using the **~A** format.

- The *mincol* parameter specifies the minimum width of output in characters. Its default value is 0.
- The *padchar* parameter specifies the padding character to be used to achieve the minimum output width. Padding characters are inserted at the left of the output. The default value of *padchar* is the space character.
- The : modifier causes the *commachar* parameter to be written between every group of three characters. The default value of *commachar* is the comma character (,).
- The @ modifier causes the integer's sign to be written. If this modifier is not specified, the sign is written only if the integer is negative.
- The : and @ modifiers may be used separately or in combination.

■ **~O** (Octal)

~mincol, padchar, commachar **:@O**

The **~O** directive is identical to the **~D** directive except that the integer argument is written in base 8 instead of in decimal.

The **~O** directive causes the printed representation of the next available argument of the function call form to be written in octal. If the argument is not an integer, it is written in octal using the **~A** format.

- The *mincol* parameter specifies the minimum width of output in characters. Its default value is 0.
- The *padchar* parameter specifies the padding character to be used to achieve the minimum output width. Padding characters are inserted at the left of the output. The default value of *padchar* is the space character.

- The : modifier causes the *commachar* parameter to be written between every group of three characters. The default value of *commachar* is the comma character (,).
- The @ modifier causes the integer's sign to be written. If this modifier is not specified, the sign is written only if the integer is negative.
- The : and @ modifiers may be used separately or in combination.

■ **~X** (Hexadecimal)

~mincol,padchar,commachar:@**X**

The **~X** directive is identical to the **~D** directive except that the integer argument is written in base 16 instead of in decimal.

The **~O** directive causes the printed representation of the next available argument of the function call form to be written in hexadecimal. If the argument is not an integer, it is written in hexadecimal using the **~A** format.

- The *mincol* parameter specifies the minimum width of output in characters. Its default value is 0.
- The *padchar* parameter specifies the padding character to be used to achieve the minimum output width. Padding characters are inserted at the left of the output. The default value of *padchar* is the space character.
- The : modifier causes the *commachar* parameter to be written between every group of three characters. The default value of *commachar* is the comma character (,).
- The @ modifier causes the integer's sign to be written. If this modifier is not specified, the sign is written only if the integer is negative.
- The : and @ modifiers may be used separately or in combination.

■ **~R** (Radix)

~radix,mincol,padchar,commachar:@**R**

The **~R** directive is identical to the **~D** directive except that the integer argument is written in the base specified by the *radix* parameter instead of in decimal. The value of *radix* must be between 2 and 36 inclusive. If the argument is not an integer, it is written in the base *radix* using the **~A** format.

- The *mincol* parameter specifies the minimum width of output in characters. Its default value is 0.
- The *padchar* parameter specifies the padding character to be used to achieve the minimum output width. Padding characters are inserted at the left of the output. The default value of *padchar* is the space character.

- The : modifier causes the *commachar* parameter to be written between every group of three characters. The default value of *commachar* is the comma character (,).
- The @ modifier causes the integer's sign to be written. If this modifier is not specified, the sign is written only if the integer is negative.
- The : and @ modifiers may be used separately or in combination.

However, if no *radix* parameter is specified, then ~R has a different meaning, as shown below.

- **~R** (Roman numerals)

  ~, *mincol*, *padchar* :@**R**

  If the first parameter is omitted, the directive **~R** causes the next available argument to be written either in English or in Roman numerals.

  - The *mincol* parameter specifies the minimum width of output in characters. Its default value is 0.
  - The *padchar* parameter specifies the padding character to be used to achieve the minimum output width. Padding characters are inserted at the left of the output. The default value of *padchar* is the space character.
  - The modifier combination selects one of the following four formats in which to write the integer argument:
    - `~R` causes the next available argument to be written in English as a cardinal number, such as `nine`.
    - `~:R` causes the next available argument to be written in English as an ordinal number, such as `ninth`.
    - `~@R` causes the next available argument to be written in Roman numerals, such as `IX`. Very large arguments are printed in decimal.
    - `~:@R` causes the next available argument to be written in old Roman numerals, such as `VIIII`. Very large arguments are printed in decimal.

- **~P** (Pluralize)

  :@**P**

  The **~P** directive is used to pluralize a word that has just been written. If no modifiers are specified, it writes nothing if the next available argument has the integer value 1; otherwise it writes the one character `s`.

  - If the : modifier is used, this directive first backs up to the previous argument of the function call form (thus re-using that argument) and then performs the pluralization.

- If the `@` modifier is used, this directive writes either the character y if the argument has the value 1 or the three characters `ies` if the argument has some other value.
- The : and `@` modifiers may be used separately or in combination.

■ ~C (Character)

:@C

The ~C directive causes the next available argument in the function call form to be written as a character. Printing characters are written as themselves; nonprinting characters are written by name (for example, a space is written as `Space`).

- If no modifiers are used, the name of any bits attribute of the character is abbreviated to one letter followed by a hyphen. For instance, the character `#\Meta-X` is written as `M-X`.
- If only the : modifier is used, the names of any bits attributes are written in full, for example, `Meta-X`.
- If only the `@` modifier is used, the character is written using the `#\` syntax, for example, `#\Meta-X`.
- If both modifiers are used, the names of any bits attributes are written in full, but the `#\` syntax is not used.

■ ~F (Fixed floating-point)

~*w*, *d*, *k*, *overflowchar*, *padchar*@F

The ~F directive causes the next available argument to be written as a floating-point number without an exponent field.

If the argument is a ratio or integer, it is coerced to single-float format before being written. If the argument is a complex number or not a number at all, it is written as if by the ~*w*D directive, so that the minimum width *w* is used and any rational subpart of the argument is written in decimal.

- The *w* parameter specifies the exact width of the output in characters. If *w* is not specified, no padding is used, and as many characters as necessary are used to write the number as specified by the remaining parameters.
- The *d* parameter specifies the number of digits to be used after the decimal point. If *d* is not specified, the number of digits after the decimal point is limited only by *w* and the value of the number. No trailing zeroes are written if *d* is not specified unless the value of the fractional part is zero, in which case exactly one zero appears after the decimal point.
- The *k* parameter specifies a scale factor. The argument is multiplied by 10^k before it is written. The default value of *k* is 0.
- If an *overflowchar* parameter is specified, and the number cannot fit in *w* characters, the entire output field is filled with the given overflow character. If

the *overflowchar* parameter is not specified and if the argument cannot fit in *w* characters, then as many characters as are necessary are used.

- The *padchar* parameter specifies the padding character to be used to achieve the minimum output width. Padding characters are inserted at the left of the output. The default value of *padchar* is the space character.
- The @ modifier causes the argument's sign to be written. If this modifier is not specified, the sign is written only if the integer is negative.

The output consists of exactly *w* characters. Any necessary padding is written first, followed by a minus sign if the argument is negative or a plus sign if the argument is nonnegative and the modifier @ is present. The magnitude of the argument times 10^k, rounded to *d* fractional digits, is written next. Leading zeroes are not written, but if the magnitude is less than one, a single zero digit is written before the decimal point if it fits within the specified width.

- ~E (Exponential floating-point)

  ~*w*, *d*, *e*, *k*, *overflowchar*, *padchar*, *exponentchar*@E

  The ~E directive causes the next available argument to be written as a floating-point number with an exponent field.

  If the argument is a ratio or integer, it is coerced to single-float format before being written. If the argument is a complex number or is not a number at all, it is written as if by the ~*w*D directive, so that the minimum width *w* is used and any rational subpart of the argument is written in decimal.

  - The *w* parameter specifies the exact width of the output in characters. If *w* is not specified, no padding is used, and as many characters as necessary are used to print the number as specified by the remaining parameters.
  - The *d* parameter specifies the number of digits to be written after the decimal point. If *d* is not specified, the number of digits after the decimal point is limited only by *w* and the value of the number. No trailing zeroes are written if *d* is not specified unless the value of the fractional part is zero, in which case exactly one zero appears after the decimal point.
  - The *e* parameter specifies the number of exponent digits written. If *e* is not specified, the minimum number of digits necessary for the exponent is used.
  - The *k* parameter is the number of significant digits written before the decimal point. If *k* is zero or negative, the first significant digit occurs after the decimal point and after $-k$ zeroes. The default value of *k* is 1.
  - If an *overflowchar* parameter is specified, and the number cannot fit in *w* characters, the entire output field is filled with the given overflow character. If the *overflowchar* parameter is not specified and if the argument cannot fit in *w* characters, then as many characters as necessary are used.

- The *padchar* parameter specifies the padding character to be used to achieve the minimum output width. Padding characters are inserted at the left of the output. The default value of *padchar* is the space character.
- The *exponentchar* parameter specifies the character written before the signed decimal exponent. The default value of *exponentchar* is E.
- The @ modifier causes the argument's sign to be written. If this modifier is not specified, the sign is written only if the integer is negative.

The output consists of exactly *w* characters. Any necessary padding is written first, followed by a minus sign if the argument is negative or by a plus sign if the argument is nonnegative and the modifier @ is present. It is followed by a digit sequence containing a decimal point in which *k* specifies the position of the first significant digit. If *k* is zero or negative, this sequence contains *d* digits after the decimal point and a single zero digit before the decimal point if the width allows. No other leading zeroes are written. If *k* is positive, it must be less than $d + 2$, and $d - k + 1$ digits are written after the decimal point. The number is rounded.

The exponent field is written next. It consists of an exponent character followed by a plus sign or a minus sign, and then *e* exponent digits. The signed exponent is the power of ten by which the number represented by the digit sequence must be multiplied to get the rounded value of the original argument. The exponent character written is *exponentchar* if that parameter is specified; otherwise the exponent marker is `E`.

- ~G (General floating-point)

  ~*w*, *d*, *e*, *k*, *overflowchar*, *padchar*, *exponentchar*@G

  The ~G directive causes the next available argument to be written as a floating-point number in either fixed or exponential format. Fixed format is used if the absolute value of the argument is either 0 or greater than or equal to 1 and if the integer part of that absolute value can be represented in *d* digits. Otherwise exponential format is used.

  If *d* is omitted, the value used for it here is `(max` *q* `(min` *n* `7))`, where *n* is the number of digits in the integer part of the absolute value of the argument, and *q* is the number of digits needed to represent the argument without loss of information and without leading or trailing zeroes.

  - If fixed format is used, the argument is written as if the following pair of directives (the first of which writes the number and the second writes some spaces) were used:

    ~*ww*, *dd*, , *overflowchar*, *padchar*@F~*ee*@T

    Here, the parameters *ww*, *dd*, and *ee* are related to the original parameters according to these prescriptions: *ee* is $e+2$ or 4 if *e* is omitted; *ww* is $w - ee$ or `nil` if *w* is omitted; and *dd* is $d - n$, with *n* (and *d*, if omitted) as defined above.

- If exponential format is used, the argument is written as if the following directive were used:

  ~*w*, *d*, *e*, *k*, *overflowchar*, *padchar*, *exponentchar*@E

  Here, all the original parameters of ~G are used in the ~E directive.

- In each of these cases, the modifier @ is supplied to the ~F or ~E directive if and only if the modifier @ was supplied to the ~G directive.

■ **~$** (Dollars floating-point)

  ~*d*, *n*, *w*, *padchar*:@$

  The **~$** directive causes the next available argument to be written as a floating-point number in fixed format. This directive can be used for writing a number in dollars and cents.

  If the argument is a complex number or not a number at all, it is written as if by the ~*w*D directive, so that the minimum width *w* is used.

  - The *d* parameter specifies the number of digits to be written after the decimal point. The default value of *d* is 2.
  - The *n* parameter specifies the minimum number of digits written before the decimal point. The default value of *n* is 1.
  - The *w* parameter specifies the minimum width of output in characters. Its default value is 0.
  - The *padchar* parameter specifies the padding character to be used to achieve the minimum output width. Padding characters are inserted at the left of the output. The default value of *padchar* is the space character.
  - The @ modifier causes the argument's sign to be written. If this modifier is not specified, the sign is written only if the integer is negative.
  - The : modifier controls whether the sign or padding is written first. Any necessary padding is written first unless the : modifier is present, in which case the sign, if any, is written before any padding.
  - The : and @ modifiers may be used separately or in combination.

  The sign and padding are written first, and then the absolute value of the argument is written as *n* digits of integer part, including leading zeroes if necessary, followed by a decimal point, and finally *d* digits of fraction. The magnitude that is written represents the rounded value of the argument.

- ~% (New line)

  ~*n*%

  The ~% directive causes a newline character to be written. Using this directive instead of inserting newline characters in the format control string may make a program easier for the user to read.

  – If the parameter *n* is specified, it must be a nonnegative integer, in which case *n* newline characters are written. The default value of *n* is 1.

- ~& (Fresh line)

  ~*n*&

  The ~& directive is used to ensure that any output that follows occurs at the beginning of a line. If the output stream is not already at the beginning of a line, a newline character is written.

  – If the parameter *n* is specified, it must be a nonnegative integer, in which case $n - 1$ (additional) newline characters are written. If *n* is 0, this directive has no effect.

- ~| (New page)

  ~*n*|

  The ~| directive causes a page character to be written.

  – If the parameter *n* is specified, it must be a nonnegative integer, in which case *n* new page characters are written. The default value of *n* is 1.

- ~~ (Tilde)

  ~*n*~

  The ~~ directive causes a tilde (~) character to be written.

  – If the parameter *n* is specified, it must be a nonnegative integer, in which case *n* tilde characters are written. The default value of *n* is 1.

- **~Newline** (Suppress newline character)

  ~:@**Newline**

  The ~**Newline** directive causes the newline character and any following whitespace characters other than the newline character to be ignored.

  – If only the : modifier is used, the newline character is ignored, but any whitespace characters other than newline characters are written.

  – If only the @ modifier is used, the newline character is written, but any following whitespace characters other than the newline character are ignored. This directive is generally useful for breaking a long format control string into multiple lines to make it easier to read, without having to insert newlines in the output.

- The : and @ modifiers are mutually exclusive.

- **~T** (Tabulate)

  ~colnum, colinc@**T**

  The **~T** directive causes any output that follows to be positioned at or beyond a given column.

  - The *colnum* parameter specifies the number of the column at which future output is to be positioned. If output has not reached column *colnum*, then enough space characters are written to reach that column. If output is already at or beyond column *colnum*, this directive writes the minimum number of spaces to reach a column that is a multiple of *colinc* columns beyond column *colnum*; but if *colinc* is 0 when output is already at or beyond column *colnum*, no spaces are written. The default value of both *colnum* and *colinc* is 1.

  - If the @ modifier is used, this directive does relative positioning by first writing *colnum* spaces and then by writing zero or more spaces to reach the nearest column that is a multiple of *colinc*.

- **~*** (Skip arguments)

  ~*n*:@*

  The **~*** directive causes the next argument of the current function call to be ignored.

  - If the parameter *n* is specified, it must be a nonnegative integer, in which case *n* arguments are ignored. The default value for *n* is 1.

  - If the : modifier is used, the directive backs up *n* arguments instead, thus allowing those arguments to be re-used. The default value of *n* here is 1.

  - If the @ modifer is used, the directive selects the *n*th argument as the point in the argument sequence at which directives continue using arguments. If *n* is 0, the first argument available to the format string is selected. The default value for *n* is 0. If **~*n*@*** is used inside a **~{** directive, it selects the *n*th argument within the argument list being processed by the iteration.

  - The : and @ modifiers are mutually exclusive.

- **~?** (Indirection)

  ~@?

  The directive **~?** causes the next available argument to be interpreted as a format control string. The argument must be a string, and the argument that follows it must be a list of the arguments to that format control string. There must be enough arguments in the list to satisfy the directives in the string; any extra arguments in the list beyond that required number are ignored. After this additional format control string has been processed, interpretation of the format control string that contains the `~?` directive resumes.

- If the modifier `@` is used, the directive takes the next available argument as a format control string, but the arguments used for that string are the next available arguments in the function call form that contains the `~@?` directive. The additional format control string thus effectively takes the place of the `~@?` directive.

- `~(` and `~)` (Case conversion)

  `~:@(`*str*`~)`

  The `~(` and `~)` directives are used to enclose an embedded format control string, *str*. The output produced by this string undergoes case conversion depending on the modifiers used with the `~(` directive.

  - If no modifiers are used, all uppercase letters contained in the embedded string are converted to lowercase.
  - If the `:` modifier is used, all words contained in the embedded string are capitalized. For this purpose, a word is considered to be any consecutive subsequence of alphanumeric characters delimited by nonalphanumeric characters or by the end of the string.
  - If the `@` modifier is used, the first word contained in the embedded string is capitalized, and the others are converted to lowercase.
  - If the `:` and `@` modifiers are used in combination, all lowercase letters contained in the embedded string are converted to uppercase.

- `~[` and `~]` with `~;` (Conditional)

  `~`*n*`:@[`*str0*`~;`*str1*`~;`...*strn*`~]`

  The directive `~[` introduces a sequence of embedded format control strings, one of which may be selected for processing. The strings are separated by `~;` and the sequence is ended by `~]`. After any selected string has been processed, the interpretation of the original format control string resumes.

  There are three possible variations of this directive, depending on the modifier combination used.

  - If no modifiers are used, `~`*j*`[`*str0*`~;`*str1*`~;`...*strn*`~]` selects the *j*th embedded string, where the first string is considered to be string 0. If the parameter *j* is omitted, the value of the next available argument is used as the value of *j* in selecting the string. If no such string exists, the directive has no effect. However, a default string can be designated to be selected if no other string is chosen by number. A default string must be the last in the sequence. It is designated by preceding it with `~:;` instead of with `~;`.
  - If the modifier `:` is used, the directive `~:[`*false*`~;`*true*`~]` selects one of the embedded format control strings *false* or *true*, depending on the value of the next available

argument in the function call form. If the argument is **nil**, the *false* string is selected; otherwise the *true* string is selected.

- If the modifier **@** is used, the directive ~**@**[*str*~] processes the format control string *str* if and only if the next available argument in the function call form is non-**nil**. In that case, the argument is made available for re-use (as if by ~:*) . If the argument is **nil**, then *str* is not processed and the argument is not re-used. Thus the string *str* is normally expected to use precisely one argument, which is non-**nil**.
- The : and **@** modifiers are mutually exclusive.

■ ~{ and ~} (Iteration)

~*max*:**@**{*str*~}

The directives ~{ and ~} enclose an embedded format control string that is to be processed repeatedly. The next available argument in the function call form must be a list. Any arguments needed by the embedded format control string are taken from this list.

- The parameter *max* specifies the maximum number of times the string is to be processed, but the repetition also terminates when there are no more unprocessed arguments left at the beginning of any iteration. The directive ~^ can be used to stop the iteration at any time. If the enclosed string ends with ~:} instead of just ~}, the string is processed at least once. However, if *max* is 0, the string is not processed at all.
- If an embedded string *str* is not specified, the next available argument of the function call form is used as the string, and as many of the arguments that follow it as are needed are used as the arguments to the string.
- If the modifier : is used, the next available argument must be a list of sublists. Each sublist in turn is used as the list of arguments for one iteration. The repetition terminates when there is no remaining sublist for the next iteration or when *max* iterations have been completed.
- If the modifier **@** is used, the remaining available arguments are treated as a list. Any arguments needed by the embedded format control string are taken from this list. Repetition terminates when no arguments remain for the next iteration or when *max* iterations have been completed. If arguments remain after *max* iterations have been completed, they are made available to any directives that follow.
- If both the : and **@** modifiers are used, the iteration is performed using as many arguments as necessary from the function call form. Each such argument must be a list of arguments to be used during that iteration. Repetition terminates when no argument is available for the next iteration or when *max* iterations have been completed.

- ~< and ~> with ~; (Justification)

  ~mincol, colinc, minpad, padchar :@<*str*~>

  The directives ~< and ~> enclose a format control string whose output is to be justified by insertion of padding characters.

  The text is padded only on the left or only on the right unless the string *str* is broken up into segments with the ~; directive; in that case padding is evenly applied to all such breaks.

  - The *mincol* parameter specifies the minimum width of output in characters. Its default value is 0.
  - If the output cannot fit into *mincol* characters, the least amount of padding is used such that the total output width is *mincol* characters plus a multiple of *colinc*. The default value of *colinc* is 1.
  - The *minpad* parameter specifies the minimum number of padding characters to be used at any point where padding is allowed. Its default value is 0.
  - The *padchar* parameter specifies the padding character to be used. Its default value is the space character.
  - If neither modifier is used, the first segment is left justified and the last segment is right justified. If there is only one segment, it is right justified.
  - If the modifier : is used, padding is added before the first segment.
  - If the modifier @ is used, padding is added after the last segment.
  - If the first segment of the string ends with the ~:; directive instead of the ~; directive, that segment is not justified. It is written only if the remaining justified segments do not fit on the current output line. This first segment should contain a newline character. All the segments are processed to generate the formatted text before any output is done, so that any arguments referenced by the first segment are used whether or not that segment is written.
  - If the first segment ends with a parameter *spare*, as in ~*spare*:;, the justified segments must fit on the current output line with *spare* columns to spare, or else the first segment is written. If the first segment ends with a second parameter *lwidth*, as in ~,*lwidth*:;, that parameter is used as the line width for the output stream. If the width is not given here and the line width of the output stream can be determined, it is used; otherwise a width of 72 is used.

  The ~^ directive can be used to terminate the processing of the string prematurely. In such a case, only those segments that have been completely processed are justified and written.

- `~^` (Up and out)

  `~`*zero*, *equal*, *ordered*`:^`

  The `~^` directive prematurely terminates the innermost iteration (`~{...~}`), justification (`~<...~>`), or indirection (`~?`) directive, or the entire format control string if no such directive is in progress. Termination occurs if the `~^` is encountered when there are no arguments left.

  - If there are any parameters to the `~^` directive, however, termination depends on their values instead of the absence of more arguments. If one parameter is supplied, termination occurs if that parameter is 0. If two parameters are supplied, termination occurs if they are equal. If three parameters are supplied, termination occurs if the first is less than or equal to the second, and the second less than or equal to the third. For this arithmetic termination test to be useful, instead of all parameters being constant in the arithmetic test, one or more of the parameters should use the `V` or `#` form to compute a variable value.
  - Within an iteration construct (`~{...~}`), the directive `~^` terminates only the current iteration, but with the modifier `:`, the directive `~:^` terminates the entire iteration directive.
  - Within a justification construct (`~<...~>`), the directive `~^` terminates the processing of segments and discards the current segment; any completely processed segments are properly justified and written.
  - Within a format control string specified by indirection (`~?`), the directive `~^` terminates that format control string. Processing continues immediately after the `~?` itself.

Summary of Format Directives

~A (ASCII)

parameters:	*mincol*, *colinc*, *minpad*, *padchar*	
defaults:	0 , 1 , 0 , '␣	
modifiers:	`~:A`	print () if argument is **nil**
	`~@A`	right justify
	`~:@A`	combine : and @

~S (S-expression)

parameters:	*mincol*, *colinc*, *minpad*, *padchar*	
defaults:	0 , 1 , 0 , '␣	
modifiers:	`~:S`	print () if argument is **nil**
	`~@S`	right justify
	`~:@S`	combine : and @

~D (Decimal)

parameters:	*mincol*, *padchar*, *commachar*	
defaults:	0 , '␣ , ',	
modifiers:	`~:D`	insert *commachar* every 3rd digit
	`~@D`	always print sign
	`~:@D`	combine : and @

~B (Binary)

parameters:	*mincol*, *padchar*, *commachar*	
defaults:	0 , '␣ , ',	
modifiers:	`~:B`	insert *commachar* every 3rd digit
	`~@B`	always print sign
	`~:@B`	combine : and @

~O (Octal)

parameters:	*mincol*, *padchar*, *commachar*	
defaults:	0 , '␣ , ',	
modifiers:	`~:O`	insert *commachar* every 3rd digit
	`~@O`	always print sign
	`~:@O`	combine : and @

~X (Hexadecimal)

parameters:	*mincol*, *padchar*, *commachar*	
defaults:	0 , '␣ , ',	
modifiers:	`~:X`	insert *commachar* every 3rd digit
	`~@X`	always print sign
	`~:@X`	combine : and @

~R (Radix)

parameters: *radix*, *mincol*, *padchar*, *commachar*
defaults: , 0 , '␣ , ',
modifiers:
`~:nR` insert *commachar* every 3rd digit
`~@nR` always print sign
`~:@nR` combine `:` and `@`

~R (Roman numerals)

parameters: , *mincol*, *padchar*, *commachar*
defaults: , 0 , ␣ ,
modifiers:
`~R` print argument as a cardinal English number, *e.g.*, `nine`
`~:R` print argument as an ordinal English number, *e.g.*, `ninth`
`~@R` print argument as a Roman numeral, *e.g.*, `IX`
`~:@R` print argument as an old Roman numeral, *e.g.*, `VIIII`

~P (Pluralize)

modifiers:
`~:P` back up to previous argument first
`~@P` print y if argument = 1, print `ies` otherwise
`~:@P` combine `:` and `@`

~C (Character)

modifiers:
`~:C` spell out control bits, *e.g.*, `Control-Z`
`~@C` print character for the Lisp reader, *e.g.*, `#\Control-Z`
`~:@C` spell out control bits and explain special shift keys, if any

~F (Fixed floating-point)

parameters: *w*, *d*, *k*, *overflowchar*, *padchar*
defaults: , , 0, , '␣
modifiers:
`~@F` always print sign

~E (Exponential floating-point)

parameters: *w*, *d*, *e*, *k*, *overflowchar*, *padchar*, *exponentchar*
defaults: , , , 1, , '␣ , 'E
modifiers:
`~@E` always print sign

~G (General floating-point)

parameters: *w*, *d*, *e*, *k*, *overflowchar*, *padchar*, *exponentchar*
defaults: , , , , , '␣ ,
modifiers:
`~@G` always print sign

~$ (Dollars floating-point)

parameters: *d*, *n*, *w*, *padchar*
defaults: 2, 1, 0, '␣
modifiers: `~:$` print sign before padding
`~@$` always print sign
`~:@$` combine : and @

~% (Newline)

parameters: *n*
defaults: 1

~& (Fresh line)

parameters: *n*
defaults: 1

~| (New page)

parameters: *n*
defaults: 1

~~ (Tilde)

parameters: *n*
defaults: 1

~newline (Suppress newline)

modifiers: ~newline ignore newline and whitespace
~:newline ignore newline, preserve whitespace
~@newline preserve newline, ignore whitespace

~T (Tabulate)

parameters: *colnum*, *colinc*
defaults: 1 , 1
modifiers: `~@T` tab *colnum* spaces, then to nearest $k \cdot colinc$ column

~* (Skip arguments)

parameters: *n*
defaults: 1
modifiers: ~*n*:* skip backwards *n* arguments
~*n*@* go to argument *n* (or argument 0 if *n* is omitted)

~? (Indirection)

modifiers: `~?` use argument as a control string, use next argument as new arguments
`~@?` use argument as a control string, use remaining arguments

~((Case conversion)

modifiers:	`~(`*str*`~)`	convert *str* to lowercase
	`~:(`*str*`~)`	capitalize all words in *str*
	`~@(`*str*`~)`	capitalize the first word of *str*; convert the rest to lowercase
	`~:@(`*str*`~)`	convert *str* to uppercase

~[(Conditional)

modifiers:	`~[`*str0*`~;...~]`	use clause given by argument (or *default* if present)
	`~nth[`*str0*`~;...~]`	use *n*th clause (or *default*, if present)
	`...~:;`*default*`~]`	the last string is the default string
	`~:[`*false*`~;`*true*`~]`	use *false* if argument is **nil**; otherwise use *true*
	`~@[`*str*`~]`	if argument is non-**nil**, use *str* and re-use argument; otherwise do nothing

~{ (Iteration)

modifiers:	`~{...}`	use argument as new argument list
	`~:{...}`	use sublists of argument as new argument list
	`~@{...}`	use required numbers of remaining arguments as list
	`~:@{...}`	use sublists of remaining arguments

~< (Justification)

parameters:	*mincol*, *colinc*, *minpad*, *padchar*	
defaults:	0 , 1 , 0 , '␣	
modifiers:	`~<`*str*`~>`	only one element, right justify
	`~<`*str0*`~;...~>`	leftmost text is left justified; rightmost text is right justified
	`~:<`*str0*`~;...~>`	put space before first text segment
	`~@<`*str0*`~;...~>`	put space after last text segment
	`~:@<`*str0*`~;...~>`	put space before and after text

~^ (Up and out)

parameters:	*zero*, *equal*, *ordered*	
defaults:	, ,	
modifiers:	`~:^`	terminate entire iteration process

Categories of Operations

This section groups input/output operations according to functionality.

Data Type Predicates

readtablep

This predicate determines whether an object is a readtable.

Character Input Control

read-base	**make-dispatch-macro-character**
read-suppress	**set-dispatch-macro-character**
readtable	**get-dispatch-macro-character**
copy-readtable	**set-syntax-from-char**
set-macro-character	***ignore-extra-right-parens***
get-macro-character	

These constructs control the operation of character input functions.

Character Output Control

print-array	***print-length***
print-base	***print-level***
print-case	***print-pretty***
print-circle	***print-radix***
print-escape	***print-structure***
print-gensym	***pp-line-length***

These variables control the operation of character output functions.

Character Stream Input

read	**read-preserving-whitespace**
read-char	**unread-char**
read-char-no-hang	**peek-char**
read-default-float-format	**listen**
read-delimited-list	**clear-input**
read-from-string	**parse-integer**
read-line	

These constructs are used to read and parse input characters.

Character Stream Output

write	**write-char**
prin1	**write-string**
print	**write-line**
pprint	**terpri**
princ	**fresh-line**
write-to-string	**finish-output**
prin1-to-string	**force-output**
princ-to-string	**clear-output**

These constructs are used to write output characters.

Binary Stream Input

read-byte

This function is used to read a byte from a binary input stream.

Binary Stream Output

write-byte

This function is used to write a byte into a binary output stream.

Formatted Character Stream Output

format

This function can be used to generate complex formatted output.

Querying the User

y-or-n-p	**yes-or-no-p**

These functions are used to ask yes-or-no questions of the user.

clear-input

Purpose: The function **clear-input** clears any available input from an input stream.

This function has no effect on any stream that is not associated with a keyboard. Its main use is to clear a keyboard stream of type-ahead characters when an error is encountered.

The function **clear-input** returns **nil**.

Syntax: **clear-input** `&optional` *input-stream* [*Function*]

Remarks: If the *input-stream* argument is not specified or is **nil**, the stream that is the current value of ***standard-input*** is used. If *input-stream* is **t**, the stream that is the value of ***terminal-io*** is used.

Examples:

```
> (progn (print (read)) (print (read)) (values)) 1 2

1
2
> (progn (print (read)) (clear-input) (print (read)) (values)) 1 2

1 this-must-now-be-typed-in

THIS-MUST-NOW-BE-TYPED-IN
> (with-input-from-string (is "1 2 3")
    (format t "~S " (read is))
    (clear-input is)
    (format t "~S " (read is)))
1 2
NIL
```

clear-output

Purpose: The function **clear-output** is used to exercise control over the internal handling of buffered stream output. It causes as much of the output data as possible to be discarded instead of being sent to its original destination.

The function **clear-output** returns **nil**.

Syntax: **clear-output** `&optional` *output-stream* [*Function*]

Remarks: If the *output-stream* argument is not specified or is **nil**, the stream that is the value of the variable ***standard-output*** is used. If *output-stream* is **t**, the stream that is the value of ***terminal-io*** is used.

Examples:

```
> (progn (print "am i seen?") (clear-output))

NIL
```

copy-readtable

Purpose: The function **copy-readtable** is used to copy readtables.

If the *from-readtable* argument is not specified, the readtable that is the current value of the variable ***readtable*** is copied. If *from-readtable* is **nil**, the standard Common Lisp readtable is copied.

If the *to-readtable* argument is not specified or is **nil**, a new readtable is created and returned. Otherwise the readtable specified by the *to-readtable* argument is modified and returned.

Syntax: **copy-readtable** `&optional` *from-readtable to-readtable* [*Function*]

Examples:

```
> (setq zvar 123)
123
> (set-syntax-from-char #\z #\' (setq table2 (copy-readtable)))
T
> zvar
123
> (copy-readtable table2 *readtable*)
#<Readtable 42A11B>
> zvar
VAR
> (setq *readtable* (copy-readtable))
#<Readtable 42AF33>
> zvar
VAR
> (setq *readtable* (copy-readtable nil))
#<Readtable 42B4B3>
> zvar
123
```

finish-output, force-output

Purpose: The functions **finish-output** and **force-output** are used to exercise control over the internal handling of buffered stream output.

The functions **finish-output** and **force-output** cause output buffers to be forced out to their final destination. Both of these functions return **nil**, but **finish-output** does so only after waiting to make sure that any buffered output has reached its target.

Syntax:

finish-output `&optional` *output-stream* [*Function*]

force-output `&optional` *output-stream* [*Function*]

Remarks: If the *output-stream* argument is not specified or is **nil**, the stream that is the value of the variable ***standard-output*** is used. If *output-stream* is **t**, the stream that is the value of ***terminal-io*** is used.

Examples:

```
> (progn (print "am i seen?") (force-output) (clear-output))

"am i seen?"
NIL


> (progn (print "am i seen?") (finish-output) (clear-output))

"am i seen?"
NIL
```

format

Purpose: The function **format** produces formatted text. The formatting is controlled by a format control string, which is made up of simple text and embedded directives. The simple text is written as is; each embedded directive specifies further text output that is to appear at the corresponding point within the simple text. All directives begin with a tilde (~) character.

If the *destination* argument is **nil**, **format** creates and returns a string containing the output from *format-control-string*. If *destination* is non-**nil**, **format** sends the output to the specified destination and returns **nil**. In this case, the value of *destination* must be a string with a fill pointer, a stream, or **t**. If *destination* is a string with a fill pointer, the output is added to the end of the string. If *destination* is a stream, the output is sent to that stream. If *destination* is **t**, output is sent to the stream that is the value of the variable ***standard-output***.

Syntax: **format** *destination format-control-string* &rest *arguments* [*Function*]

Remarks: The section "Formatted Output" explains how the format control string is interpreted and how the elements of *arguments* are processed.

Examples:

```
> (format t "no args")
no args
NIL
> (format nil "some ~A returned ~% as a ~S" 'args 'string)
"some ARGS returned
 as a STRING"
> (format *standard-output* "~{~S~%~}" '(1 2 3))
1
2
3
NIL
> (format t "~s ~:* ~d ~:* ~b ~:* ~o ~:* ~x ~:* ~r ~:* ~35r ~:* ~:@r" 99)
99  99  1100011  143  63  ninety-nine  2T  LXXXXVIIII
NIL
> (format t "~R pupp~:@p" 8)
eight puppies
NIL
```

get-dispatch-macro-character

Purpose: The function **get-dispatch-macro-character** returns the dispatch function associated with a particular dispatching macro character pair in a readtable.

The argument *disp-char* must be a dispatching macro character in the indicated readtable. The value returned is the dispatch function for the subcharacter *sub-char* associated with the macro character *disp-char*. If the subcharacter has no dispatch function, **get-dispatch-macro-character** returns **nil**.

Syntax: **get-dispatch-macro-character** *disp-char sub-char* `&optional` *readtable* [*Function*]

Remarks: If the *readtable* argument is not specified, the readtable that is the current value of the variable ***readtable*** is used.

If *sub-char* is a lowercase letter, it is converted to its uppercase equivalent. If *sub-char* is a decimal digit, **get-dispatch-macro-character** returns **nil**.

Examples:

```
> (null (get-dispatch-macro-character #\# #\{))
T
> (null (get-dispatch-macro-character #\# #\x))
NIL
```

See Also: **set-dispatch-macro-character**

get-macro-character

Purpose: The function **get-macro-character** returns the function associated with a specified macro character in a readtable. If there is no such function, **get-macro-character** returns **nil**.

A second value is returned that indicates whether the character is a nonterminating macro character. If the character is a nonterminating macro character, this value is true; otherwise it is false.

Syntax: **get-macro-character** *char* `&optional` *readtable* [*Function*]

Remarks: If the *readtable* argument is not specified, the readtable that is the current value of the variable ***readtable*** is used.

Examples:

```
> (null (get-macro-character #\{))
T
> (null (get-macro-character #\;))
NIL
```

See Also: **set-macro-character**

ignore-extra-right-parens

Purpose: The variable ***ignore-extra-right-parens*** is used to control the action of the reader when excess right parentheses are encountered in the input stream. If ***ignore-extra-right-parens*** is **t**, excess right parentheses in the input stream are ignored; if it is **:just-warn**, a warning message is generated; if it is **nil**, a continuable error is signaled.

Syntax: ***ignore-extra-right-parens*** [*Variable*]

Remarks: The initial value of ***ignore-extra-right-parens*** is **:just-warn**.

The variable ***ignore-extra-right-parens*** is an extension to Common Lisp.

Examples:

```
> *ignore-extra-right-parens*
:JUST-WARN
> (read-from-string ")1")
;;; Warning: Ignoring an unmatched right parenthesis.
1
2
> (let((*ignore-extra-right-parens* t))
      (declare (special *ignore-extra-right-parens*))
    (read-from-string ")1"))
1
2
```

listen

Purpose: The predicate **listen** is true if a character can be read from a given input stream; otherwise it is false.

Syntax: **listen** `&optional` *input-stream* [*Function*]

Remarks: If the *input-stream* argument is not specified or is **nil**, the stream that is the value of the variable ***standard-input*** is used. If *input-stream* is **t**, the stream that is the value of ***terminal-io*** is used.

If an end-of-file is encountered, **listen** returns **nil**.

This function is designed to allow a program to avoid waiting for input. It is often used with a stream associated with a keyboard to determine if the user has typed a character.

Examples:

```
> (listen) 1
T
>
1
> (progn (clear-input) (listen))
NIL
```

See Also: **read-char-no-hang**

make-dispatch-macro-character

Purpose: The function **make-dispatch-macro-character** makes the character *char* a dispatching macro character in a given readtable. The function **make-dispatch-macro-character** returns **t**.

Syntax: **make-dispatch-macro-character** *char* `&optional` *non-terminating-p readtable* [*Function*]

Remarks: A dispatching macro character has an associated table that specifies the function to be called for each character that can be read following the dispatching macro character. This dispatch table is initialized by **make-dispatch-macro-character**, so that every such character has an associated function that signals an error. The function **set-dispatch-macro-character** is used to specify the dispatch functions for characters that follow a dispatching macro character.

If the argument *non-terminating-p* is non-**nil**, the dispatching macro character is made a nonterminating macro character; otherwise it is made a terminating macro character. The default value for *non-terminating-p* is **nil**.

If the *readtable* argument is not specified, the readtable that is the current value of the variable ***readtable*** is used.

Examples:

```
> (get-macro-character #\{)
NIL
> (make-dispatch-macro-character #\{)
T
> (null (get-macro-character #\{))
NIL
```

See Also: **set-dispatch-macro-character**

parse-integer

Purpose: The function **parse-integer** reads an integer from a given string, using a specified radix. White space before and after the integer is ignored.

The parsing operation may be restricted to a substring of the string by specifying the **:start** and **:end** keyword arguments.

The function **parse-integer** returns two values. The first value is the integer parsed. If no integer is found and **:junk-allowed** is true, the first value is **nil**. The second value specifies the index within the string of the character that caused the parse to terminate (or one character beyond the end of the substring if the parse reached the end of the substring).

Syntax: **parse-integer** *string* `&key :start :end :radix :junk-allowed` [*Function*]

Remarks: The **:start** and **:end** keyword arguments take integer values that specify offsets into the string. The **:start** argument marks the beginning position of the substring; the **:end** argument marks the position following the last element of the substring. The start value defaults to 0; the end value defaults to the length of the string.

The **:radix** keyword argument specifies the base in which the number is to be read. It must be an integer from 2 to 36 inclusive. If **:radix** is not specified, base 10 is used.

The **:junk-allowed** keyword argument specifies whether the given substring is permitted to contain anything besides the integer and whitespace characters. If **:junk-allowed** is **nil**, the substring must contain precisely one integer, optional leading and trailing whitespace characters, and nothing else. If **:junk-allowed** is non-**nil**, the substring can contain arbitrary text following the integer.

Integers parsed by **parse-integer** must consist of an optional sign and one or more digits in the indicated radix.

Examples:

```
> (parse-integer "123")
123
3
> (parse-integer "123" :start 1 :radix 5)
13
3
> (parse-integer "foo" :junk-allowed t)
NIL
0
```

peek-char

Purpose: The function **peek-char** returns the next character in an input stream without actually reading it, thus leaving the character to be read at a later time. It can also be used to skip over and discard intervening characters in the input stream until a particular character is found.

Syntax: **peek-char** `&optional` *peek-type input-stream eof-error-p eof-value recursive-p* [*Function*]

Remarks: If the *peek-type* argument is **nil**, **peek-char** simply looks at the next character in the input stream and returns it without reading it out of the stream.

If *peek-type* is **t**, **peek-char** reads and discards any whitespace characters at the front of the input stream and returns the first nonwhitespace character in the stream without actually reading it. Note that comments are not discarded in this process.

If *peek-type* is a character, **peek-char** discards characters from the front of the input stream until encountering a character that is the same as *peek-type* (**char=**). That character is returned without being read out of the stream.

If an end-of-file occurs before such a character can be read, an error is signaled if *eof-error-p* is true. If an end-of-file occurs and *eof-error-p* is **nil**, no error is signaled and *eof-value* is returned. The default value of *eof-error-p* is true. The default value of *eof-value* is **nil**.

The argument *recursive-p* should be true if this call is embedded in a higher-level call to **read** or a similar function.

The *input-stream* argument specifies the stream to be used. If it is not specified or is **nil**, the stream that is the value of the variable ***standard-input*** is used. If *input-stream* is **t**, the stream that is the value of ***terminal-io*** is used.

Examples:

```
> (with-input-from-string (is "    1 2 3 4 5")
    (format t "~S ~S ~S"
            (peek-char t is)
            (peek-char #\4 is)
            (peek-char nil is)))
#\1 #\4 #\4
NIL
```

print-array

Purpose: The variable ***print-array*** controls the format in which arrays are printed.

If the value of ***print-array*** is non-**nil**, arrays are printed in their entirety with the `#(`, `#*`, or `#`*n*`A` syntax. If ***print-array*** is **nil**, just enough is printed, using the `#<...>` syntax, to identify the array.

Syntax: ***print-array*** [*Variable*]

Remarks: The initial value of ***print-array*** is **nil**.

Examples:

```
> *print-array*
NIL
> (setq a (make-array '(2 3)))
#<Simple-Array T (2 3) 4789D3>
> (let ((*print-array* t)) (format t "~S" a))
#2A((NIL NIL NIL) (NIL NIL NIL))
NIL
```

print-base, *print-radix*

Purpose: The variables ***print-base*** and ***print-radix*** control the printing of rational numbers.

The value of the variable ***print-base*** is the numerical base in which integers and ratios are printed.

The value of the variable ***print-radix*** determines whether a radix indicator is included with each integer or ratio printed. If the value of ***print-radix*** is non-**nil**, a radix indicator is printed.

Syntax:

print-base [*Variable*]

print-radix [*Variable*]

Remarks: The initial value of ***print-base*** is 10.

The initial value of ***print-radix*** is **nil**.

The value of ***print-base*** must be an integer value between 2 and 36 inclusive. When the value of ***print-base*** is greater than 10, capital letters are used for digits greater than 9, starting with A for 10, B for 11, and so on.

When the value of ***print-radix*** is non-**nil**, a decimal base is indicated for integers by a decimal point following the number; for ratios, a leading **#10r** is used. For a base of 2, 8, or 16, a leading **#b**, **#o**, or **#x** is used respectively. For other values of ***print-base***, a leading **#*n*r** radix indicator is used, with the base *n* itself printed in decimal.

Examples:

```
> *print-base*
10
> (dotimes (i 35)
    (let ((*print-base* (+ i 2)))          ;print the decimal number 40
      (write 40)                           ;in each base from 2 to 36
      (if (zerop (mod i 10)) (terpri) (format t " "))))
101000
1111 220 130 104 55 50 44 40 37 34
31 2C 2A 28 26 24 22 20 1J 1I
1H 1G 1F 1E 1D 1C 1B 1A 19 18
17 16 15 14
NIL
```

```
> *print-radix*
NIL
> (dolist (pb '(2 3 8 10 16))
    (let ((*print-radix* t)                    ;print the integer 10 and
          (*print-base* pb))                   ;the ratio 1/10 in bases 2,
     (format t "~S ~S   " 10 1/10)))           ;3, 8, 10, 16
#b1010 #b1/1010   #3r101 #3r1/101   #o12 #o1/12   10. #10r1/10   #xA #x1/A
NIL
```

print-case

Purpose: The variable ***print-case*** determines the case used in printing the names of symbols.

Normally, symbol names are stored internally with uppercase letters and are printed with uppercase letters. The value of ***print-case*** specifies the case in which uppercase letters in symbol names are printed. Lowercase letters in symbol names are always printed in lowercase.

The value of ***print-case*** must be either **:upcase**, **:downcase**, or **:capitalize**. Corresponding to these three possible values, the printing of uppercase letters of symbols is in uppercase, in lowercase, or in a combination of cases in which words are capitalized.

Syntax: ***print-case*** [*Variable*]

Remarks: The initial value of ***print-case*** is **:upcase**.

For purposes of capitalization, a word is considered to be any consecutive sequence of alphanumeric characters that is preceded and followed by nonalphanumeric characters or the end of the symbol name.

Examples:

```
> *print-case*
:UPCASE
> (dolist (pc '(:upcase :downcase :capitalize))
    (let ((*print-case* pc)) (format t "~S " 'foo-bar)))
FOO-BAR foo-bar Foo-bar
NIL
```

print-circle

Purpose: The variable ***print-circle*** controls the attempt to detect circularity in an object being printed.

If ***print-circle*** is non-**nil** and a circular object is detected, the **#*n*=** and **#*n*#** constructs are used to denote the circular structure.

If ***print-circle*** is **nil**, an attempt to print a circular object may cause Common Lisp to loop indefinitely.

Syntax: ***print-circle*** [*Variable*]

Remarks: The initial value of ***print-circle*** is **nil**.

Examples:

```
> *print-circle*
NIL
> (progn (setq a '(1 2 3))
         (setf (cdddr a) a)                       ;create a circular list
         (values))
> (let ((*print-circle* t)) (write a) (values))  ;print it
#1=(1 2 3 . #1#)
```

print-escape

Purpose: The variable ***print-escape*** controls the printing of escape characters. If ***print-escape*** is **nil**, the printing of escape characters is suppressed.

Syntax: ***print-escape*** [*Variable*]

Remarks: The initial value of ***print-escape*** is **t**.

Examples:

```
> *print-escape*
T
> (write #\a)
#\a
#\a
> (let ((*print-escape* nil)) (write #\a))
a
#\a
```

See Also: **princ**

prin1

print-gensym

Purpose: The variable ***print-gensym*** controls the printing of the names of uninterned symbols. If ***print-gensym*** is non-**nil**, the prefix `#:` is printed before the name of any uninterned symbol.

Syntax: ***print-gensym*** [*Variable*]

Remarks: The initial value of ***print-gensym*** is **t**.

Examples:

```
> *print-gensym*
T
> (format t "~S" (gensym))
#:G39
NIL
> (let ((*print-gensym* nil)) (format t "~S" (gensym)))
G40
NIL
```

print-level, *print-length*

Purpose: The variables ***print-level*** and ***print-length*** are used to limit the amount of output when an object is printed. These two variables affect the printing of any object with a listlike syntax, including lists, vectors, and arrays.

If ***print-level*** is set to an integer value, the printing depth of an object is limited to that value. The object itself is considered to be at level 0. Any portion at or below the level of ***print-level*** is printed as just # if that portion contains components. If ***print-level*** is **nil**, no limit is imposed on the printing depth.

If ***print-length*** is set to an integer value, the maximum number of consecutive elements printed at any level is limited to that value. An ellipsis (...) is used to represent further objects at that level. If ***print-length*** is **nil**, no limit is imposed on the number of elements printed.

Syntax:

print-level [*Variable*]

print-length [*Variable*]

Remarks: The initial value of both ***print-level*** and ***print-length*** is **nil**.

Examples:

```
> *print-level*
NIL
> (setq a '(1 (2 (3 (4 (5 (6)))))))
(1 (2 (3 (4 (5 (6))))))
> (dotimes (i 3) (let ((*print-level* (* i 3))) (format t "~S~%" a)))
#
(1 (2 (3 #)))
(1 (2 (3 (4 (5 (6))))))
NIL
> *print-length*
NIL
> (setq a '(1 2 3 4 5 6))
(1 2 3 4 5 6)
> (dotimes (i 3) (let ((*print-length* (* i 3))) (format t "~S~%" a)))
( ...)
(1 2 3 ...)
(1 2 3 4 5 6)
NIL
```

print-pretty, *pp-line-length*

Purpose: The variables ***print-pretty*** and ***pp-line-length** are used to control pretty-printing.

The value of the variable ***print-pretty*** controls the use of whitespace characters. If the value of ***print-pretty*** is non-**nil**, additional whitespace characters are written in order to make printed expressions easier to read. If ***print-pretty*** is **nil**, a minimal amount of white space is used.

The value of the variable ***pp-line-length*** is an integer that specifies the output line length to be used for pretty-printing (for example, when ***print-pretty*** is true).

Syntax:

print-pretty [*Variable*]

pp-line-length [*Variable*]

Remarks: The initial value of ***print-pretty*** is **nil**.

The initial value of ***pp-line-length*** is 80.

The variable ***pp-line-length*** is an extension to Common Lisp.

Examples:

```
> *print-pretty*
NIL
> (progn (write '(let((a 1)(b 2)(c 3))(+ a b c))) (values))
(LET ((A 1) (B 2) (C 3)) (+ A B C))
> (let ((*print-pretty* t))
    (progn (write '(let((a 1)(b 2)(c 3))(+ a b c))) (values)))
(LET ((A 1)
      (B 2)
      (C 3))
  (+ A B C))
```

See Also: **grindef**

print-structure

Purpose: The variable ***print-structure*** controls the printing of structures.

If the value of ***print-structure*** is non-**nil**, structures are printed in detail, using the `#S` syntax. If ***print-structure*** is **nil**, they are printed with the abbreviated `#<...>` syntax.

Syntax: ***print-structure*** [*Variable*]

Remarks: The initial value of ***print-structure*** is **t**.

The variable ***print-structure*** is an extension to Common Lisp.

Examples:

```
> *print-structure*
T
> (defstruct family mom dad brother sister dog)
FAMILY
> (setq jones (make-family :mom 'simone :dad 'sam :brother 'basket-ball
                           :sister 'sally :dog 'bowser))
#S(FAMILY MOM SIMONE DAD SAM BROTHER BASKET-BALL SISTER SALLY DOG BOWSER)
> (let ((*print-structure* nil)) (print jones) (values))

#<Structure FAMILY 428F3B>
```

read, read-preserving-whitespace

Purpose: The function **read** reads the printed representation of an object from an input stream. The object itself is constructed from its printed representation and returned as the value of **read**.

The function **read-preserving-whitespace** is like **read** but preserves any whitespace character that delimits the printed representation of the object.

Syntax:

read `&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*]

Remarks: If the *input-stream* argument is not specified or is **nil**, the stream that is the value of the variable ***standard-input*** is used. If *input-stream* is **t**, the stream that is the value of ***terminal-io*** is used.

If an end-of-file occurs before an object can be read, an error is signaled if *eof-error-p* is true. An error is always signaled if an end-of-file occurs in the middle of an incomplete object, such as before the right parenthesis that ends a list. If *eof-error-p* is **nil** and an end-of-file occurs anywhere else, no error is signaled and *eof-value* is returned. The default value of *eof-error-p* is true. The default value of *eof-value* is **nil**.

The argument *recursive-p* should be true if the call to **read** is from within some function that itself has been called from **read** or from a similar input function, rather than from the top level. For instance, a macro character function that has to read from the input stream beyond the macro character should specify *recursive-p* as true. The reasons for this are as follows. First of all, the scoping of the constructs **#*n*=** and **#*n*#** occurs within a top-level call, so calls to **read** from macro character functions must specify *recursive-p* as true to ensure that these constructs are interpreted correctly. Second, for white space to be preserved correctly by low-level calls to **read** occurring within a call to **read-preserving-whitespace**, the *recursive-p* argument must be true. Otherwise a low-level call to **read** does not know that it needs to preserve white space for the higher-level call.

A macro character function should not rely on any side effects it has on the reader's global variables, such as ***readtable***, unless such effects are made only for a top-level call to **read**. The reader caches certain variables during the entry to **read** at the top level, where *recursive-p* is **nil**, and thus may not notice changes to those variables below the top level.

Examples:

```
> (read)
'a
(QUOTE A)
> (with-input-from-string (is " ") (read is nil 'the-end))
THE-END
> (defun skip-then-read-char (s c n)
    (if (char= c #\{) (read s) (read-preserving-whitespace s))
    (read-char-no-hang s))
SKIP-THEN-READ-CHAR
> (let ((*readtable* (copy-readtable nil)))
    (set-dispatch-macro-character #\# #\{ #'skip-then-read-char)
    (set-dispatch-macro-character #\# #\} #'skip-then-read-char)
    (with-input-from-string (is "#{123 x #}123 y")
      (format t "~S ~S" (read is) (read is))))
#\x #\Space
NIL
```

read-base

Purpose: The value of the variable ***read-base*** is the numerical base used for reading integers and ratios.

Floating-point numbers are always read as decimal numbers regardless of the value of ***read-base***. Any number whose base is specified explicitly, such as a number that contains a decimal point or that starts with `#O`, `#X`, `#B`, or `#nR`, is also unaffected by ***read-base***.

Syntax: ***read-base*** [*Variable*]

Remarks: The value of ***read-base*** can be any integer from 2 to 36 inclusive. The initial value of ***read-base*** is 10.

When ***read-base*** is greater than 10, ambiguity can arise over a symbol name composed of letters, all of which are digits in the current base; such a symbol may be read as a number.

The use of a read base other than decimal is not recommended except for reading data files. Nondecimal numbers within programs should be notated with `#O`, `#X`, `#B`, or `#nR`.

Examples:

```
> *read-base*
10
> (setq dad 'pop)
POP
> 16
16
> dad
POP
> (setq *read-base* 16)
16
> 16
22
> dad
3501
```

read-byte

Purpose: The function **read-byte** reads a single byte from a specified binary input stream. The byte is returned as an integer.

Syntax: **read-byte** *binary-input-stream* `&optional` *eof-error-p eof-value* [*Function*]

Remarks: The size of the byte read depends on the **:element-type** argument given in the **open** or **with-open-file** construct that created the stream *binary-input-stream*. Unless the byte size of that element type is one, two, or four bits, each call to **read-byte** uses up an integral number of 8-bit bytes, namely the minimum number necessary to hold the number of bits indicated by the given element type. If the byte size of the element type is one, two, or four bits, then as many elements as possible (eight, four, or two respectively) are unpacked from each 8-bit byte.

If an end-of-file occurs before a byte can be read, an error is signaled if *eof-error-p* is true. If an end-of-file occurs and *eof-error-p* is **nil**, no error is signaled and *eof-value* is returned. The default value of *eof-error-p* is true. The default value of *eof-value* is **nil**.

Examples:

```
> (with-open-file (s "temp-bytes"
                     :direction :output
                     :element-type 'unsigned-byte)
    (write-byte 101 s))
101
> (with-open-file (s "temp-bytes" :element-type 'unsigned-byte)
    (format t "~S ~S" (read-byte s) (read-byte s nil 'eof)))
101 EOF
NIL
```

See Also: **write-byte**

read-char

Purpose: The function **read-char** reads a character from an input stream. The character that is read is returned as the result of **read-char**.

Syntax: **read-char** `&optional` *input-stream eof-error-p eof-value recursive-p* [*Function*]

Remarks: If the *input-stream* argument is not specified or is **nil**, the stream that is the value of the variable ***standard-input*** is used. If *input-stream* is **t**, the stream that is the value of ***terminal-io*** is used.

If an end-of-file occurs before a character can be read, an error is signaled if *eof-error-p* is true. If an end-of-file occurs and *eof-error-p* is **nil**, no error is signaled and *eof-value* is returned. The default value of *eof-error-p* is true. The default value of *eof-value* is **nil**.

The argument *recursive-p* should be true if this call is embedded in a higher-level call to **read** or a similar function.

The function **read-char** does not perform case conversion on alphabetic characters.

Examples:

```
> (with-input-from-string (is "0123")
    (do ((c (read-char is) (read-char is nil 'the-end)))
        ((not (characterp c)))
     (format t "~S" c)))
#\0#\1#\2#\3
NIL
```

See Also: **read**

read-char-no-hang

Purpose: The function **read-char-no-hang** reads and returns a character from the input stream if such a character is available. If no character is available, **read-char-no-hang** returns **nil**.

Syntax: **read-char-no-hang** `&optional` *input-stream eof-error-p eof-value recursive-p* [*Function*]

Remarks: The *input-stream* argument specifies the stream to be used. If it is not specified or is **nil**, the stream that is the value of the variable ***standard-input*** is used. If *input-stream* is `t`, the stream that is the value of ***terminal-io*** is used.

If an end-of-file occurs, an error is signaled if *eof-error-p* is true. If an end-of-file occurs and *eof-error-p* is **nil**, no error is signaled and *eof-value* is returned. The default value of *eof-error-p* is true. The default value of *eof-value* is **nil**.

The argument *recursive-p* should be true if this call is embedded in a higher-level call to **read** or a similar function.

This function is designed to allow a program to avoid waiting for input.

Examples:

```
> (format t "~S ~S ~S" (read-char-no-hang)
                       (read-char-no-hang)
                       (read-char-no-hang))a
#\a #\Newline NIL
NIL
```

See Also: **listen**

read-default-float-format

Purpose: The variable ***read-default-float-format*** specifies the floating-point format that is to be used when reading a floating-point number that contains no explicit format indicator.

Syntax: ***read-default-float-format*** [*Variable*]

Remarks: The initial value of ***read-default-float-format*** is **single-float**.

In Sun Common Lisp, all floating-point numbers are represented in single-float format, and the value of ***read-default-float-format*** has no effect.

Examples:

```
> *read-default-float-format*
SINGLE-FLOAT
```

read-delimited-list

Purpose: The function **read-delimited-list** reads objects from an input stream until a specified delimiting character is found. A list of the objects that have been read up to that point is returned.

Syntax: **read-delimited-list** *char* `&optional` *input-stream recursive-p* [*Function*]

Remarks: The argument *char* must not be a whitespace character in the current readtable, because whitespace characters are ignored by **read-delimited-list**. A terminating macro character is usually chosen as the delimiting character so that it can follow the last object to be read without any intervening white space.

The *input-stream* argument specifies the stream to be used. If it is not specified or is **nil**, the stream that is the value of the variable ***standard-input*** is used. If *input-stream* is **t**, the stream that is the value of ***terminal-io*** is used.

The argument *recursive-p* should be true if this call is embedded in a higher-level call to **read** or a similar function.

An error is signaled if an end-of-file is encountered during **read-delimited-list**.

Examples:

```
> (read-delimited-list #\])
1 2 3
4 5 6 ]
(1 2 3 4 5 6)
```

See Also: **read**

read-from-string

Purpose: The function **read-from-string** reads an object's printed representation from a specified string instead of from an input stream. The object is constructed from its printed representation and returned as the value of **read-from-string**. A second value is returned that specifies the index within the string of the character just beyond the last character read.

The operation may be restricted to a substring of the string by specifying the **:start** and **:end** keyword arguments.

Syntax: **read-from-string** *string* `&optional` *eof-error-p eof-value* [*Function*]
`&key :start :end :preserve-whitespace`

Remarks: The **:start** and **:end** keyword arguments take integer values that specify offsets into the string. The **:start** argument marks the beginning position of the substring; the **:end** argument marks the position following the last element of the substring. The start value defaults to 0, the end value to the length of the string.

If the **:preserve-whitespace** keyword argument is non-**nil**, the read operation preserves white space; otherwise it does not. The default value of **:preserve-whitespace** is **nil**.

If the end of the specified substring occurs before an object can be read, an error is signaled if *eof-error-p* is true. An error is always signaled if the end of the substring occurs in the middle of an incomplete object. If *eof-error-p* is **nil** and if the end of the substring occurs anywhere else, no error is signaled and *eof-value* is returned. The default value of *eof-error-p* is true. The default value of *eof-value* is **nil**.

If any keyword arguments are supplied to **read-from-string**, both of the optional arguments must also be specified. Otherwise the first keyword and its value are taken as the optional arguments.

Examples:

```
> (read-from-string " 1 3 5" t nil :start 2)
3
5
```

See Also: **read**

read-preserving-whitespace

read-line

Purpose: The function **read-line** reads a line of text from an input stream. The characters up to but not including the newline character that ends the line are returned as a string. A second value is also returned; it is **nil** if the line was terminated normally and non-**nil** if a nonempty line was terminated by an end-of-file.

Syntax: **read-line** `&optional` *input-stream eof-error-p eof-value recursive-p* [*Function*]

Remarks: If the *input-stream* argument is not specified or is **nil**, the stream that is the value of the variable ***standard-input*** is used. If *input-stream* is **t**, the stream that is the value of ***terminal-io*** is used.

If an end-of-file occurs before any characters are read in the line, an error is signaled if *eof-error-p* is true. If an end-of-file occurs and *eof-error-p* is **nil**, no error is signaled and *eof-value* is returned. The default value of *eof-error-p* is true. The default value of *eof-value* is **nil**.

The argument *recursive-p* should be true if this call is embedded in a higher-level call to **read** or a similar function.

Examples:

```
> (setq a "line 1
line2")
"line 1
line2"
> (read-line (setq is (make-string-input-stream a)))
"line 1"
NIL
> (read-line is)
"line2"
T
> (read-line is nil 'empty)
EMPTY
T
```

See Also: **read**

read-suppress

Purpose: The variable ***read-suppress*** can be used to suppress many of the operations normally performed by the reader.

If the value of ***read-suppress*** is non-**nil**, much of the interpretation that is usually carried out when expressions are read is suppressed. Suppression of interpretation is needed by the conditional-read constructs `#+` and `#-`, whose principal use is to make a single program work under other Lisp systems that have slight differences in syntax.

When ***read-suppress*** is **nil**, normal read operations take place.

Syntax: ***read-suppress*** *[Variable]*

Remarks: When ***read-suppress*** is non-**nil**, the reader skips over certain printed constructs that may not be entirely valid. The effects of a non-**nil** value of ***read-suppress*** are listed below. Constructions other than those listed continue to be interpreted normally.

- Extended tokens are not interpreted but are discarded and treated as if they were **nil**. For instance, potential numbers and symbols qualified with package markers are not checked for valid syntax.
- Standard `#` dispatching macro character constructs ignore normal restrictions on the presence, absence, or value of a numeric argument, such as that in `#`*n*`R`.
- The construct `#\` reads a following character or character name but generates **nil** in all cases. Unknown character names do not cause errors.
- The constructs `#B`, `#O`, `#X`, and `#`*n*`R` read the next token and generate **nil**. No errors are signaled, even if the token does not have numeric syntax.
- The construct `#*` reads the next token and generates **nil**. No errors are signaled, even if the token contains characters other than 0 or 1.
- The constructs `#.` and `#,` read the following form without evaluating it and then generate **nil**.
- The constructs `#A`, `#S`, and `#:` read the following form without interpreting it and without requiring it to be a list (for `#S`) or a symbol (for `#:`). The value **nil** is generated.
- The construct `#`*n*`=` (where *n* is an integer) is completely ignored, generates no object, and is treated as white space.
- The construct `#`*n*`#` (where *n* is an integer) generates **nil**.

Examples:

```
> *read-suppress*
NIL
> (let ((*read-suppress* t))
    (format t "~%input here> ")
    (format t "evaluated as: ~S~%" (eval (read)))
    (format t "~%input here> ")
    (format t "evaluated as: ~S~%" (eval (read)))
    (format t "~%input here> ")
    (format t "evaluated as: ~S~%" (eval (read)))
    (format t "~%input here> ")
    (format t "evaluated as: ~S~%" (eval (read)))
    (values))

input here> 101
evaluated as: NIL

input here> #\a
evaluated as: NIL

input here> :test
evaluated as: NIL

input here> (list 1 2 3)
evaluated as: NIL
> 101
101
```

See Also: read

readtable

Purpose: The variable ***readtable*** specifies the current readtable.

Syntax: ***readtable*** [*Variable*]

Remarks: The initial value of this variable is a readtable that provides the standard Common Lisp syntax.

Examples:

```
> (readtablep *readtable*)
T
> (setq zvar 123)
123
> (set-syntax-from-char #\z #\' (setq table2 (copy-readtable)))
T
> zvar
123
> (setq *readtable* table2)
#<Readtable 429B13>
> zvar
VAR
> (setq *readtable* (copy-readtable nil))
#<Readtable 42A11B>
> zvar
123
```

readtablep

Purpose: The predicate **readtablep** is true if its argument is a readtable; otherwise it is false.

Syntax: **readtablep** *object* [*Function*]

Examples:

```
> (readtablep *readtable*)
T
> (readtablep (copy-readtable))
T
> (readtablep '*readtable*)
NIL
```

set-dispatch-macro-character

Purpose: The function **set-dispatch-macro-character** installs a dispatch function to be called when a particular dispatching macro character pair is read.

The function *function* is installed as the dispatch function to be called when the readtable *readtable* is in use and when the character *disp-char* is followed by the character *sub-char*. The argument *disp-char* must be a dispatching macro character in the indicated readtable.

The function **set-dispatch-macro-character** returns **t**.

Syntax: **set-dispatch-macro-character** *disp-char sub-char function* `&optional` *readtable* [*Function*]

Remarks: Whenever the indicated character sequence is read, the dispatch function *function* is called with three arguments: the current input stream, *sub-char*, and the nonnegative decimal number that was read between *disp-char* and *sub-char*. If no such number has been read, the third argument is **nil**.

If the *readtable* argument is not specified, the readtable that is the current value of the variable ***readtable*** is used.

The argument *sub-char* must not be a decimal digit. If *sub-char* is a lowercase letter, it is converted to its uppercase equivalent. Thus case is not significant in a dispatching macro subcharacter.

Examples:

```
> (get-dispatch-macro-character #\# #\{)
NIL
> (set-dispatch-macro-character #\# #\{        ;dispatch on #{
    #'(lambda(s c n)
        (let ((list (read s nil (values) t)))  ;list is object after #n{
          (when (consp list)                   ;return nth element of list
            (unless (and n (< 0 n (length list))) (setq n 0))
            (setq list (nth n list)))
         list)))
T
> #{(1 2 3 4)
1
> #3{(0 1 2 3)
3
> #{123
123
```

See Also: **get-dispatch-macro-character**

set-macro-character

Purpose: The function **set-macro-character** makes the specified readtable character a macro character and installs a function to be called whenever that character is read and the given readtable is in use.

The function **set-macro-character** always returns **t**.

Syntax: **set-macro-character** *char function* [*Function*]
`&optional` *non-terminating-p readtable*

Remarks: The character is made a macro character in the readtable *readtable*. If the *readtable* argument is not specified, then *char* is made a macro character in the readtable that is the current value of the variable ***readtable***.

When the character *char* is read and the specified readtable is current, the function *function* is called. It is passed two arguments: the input stream from which characters are being read and the macro character that caused it to be invoked. Normally, such a function returns a Common Lisp object that it reads from the input stream; this is the object whose printed representation starts with *char*. However, the function may return no values to indicate that no object has been read, as may be the case when a comment is scanned.

If the *non-terminating-p* argument is specified and is non-**nil**, the character becomes a nonterminating macro character; otherwise it becomes a terminating macro character. A nonterminating macro character that appears in the middle of an extended token is treated like a constituent character, and the macro character's function is not called in that case. A terminating macro character always terminates any token it appears in, and the terminating macro character's function is always called. The *non-terminating-p* argument defaults to **nil**.

Examples:

```
> (set-macro-character #\{                    ;makes { read an object
    #'(lambda (s c)                          ;and return a string
        (with-output-to-string (os)          ;of its ~S format output
          (format os "~S" (read s nil (values) t)))))
T
> {123
"123"
```

set-syntax-from-char

Purpose: The function **set-syntax-from-char** sets the syntax of one readtable character from the syntax of another readtable character.

The syntax type of the character *to-char* in the readtable *to-readtable* is set to the syntax type of the character *from-char* in the readtable *from-readtable*.

Syntax: **set-syntax-from-char** *to-char from-char* [*Function*]
`&optional` *to-readtable from-readtable*

Remarks: If the *to-readtable* argument is not specified, the readtable that is the current value of the variable ***readtable*** is used.

If the *from-readtable* argument is not specified or is **nil**, the standard Common Lisp readtable is used.

If the character is a macro character, the function associated with the character is also copied. If the character is a dispatching macro character, its entire dispatch table of functions is copied. The constituent character attributes, however, are not copied.

Examples:

```
> (set-syntax-from-char #\7 #\;)
T
> 123579
1235
```

terpri, fresh-line

Purpose: The functions **terpri** and **fresh-line** ensure that subsequent output begins on a new line.

The function **terpri** writes a newline character and returns **nil**. The function **fresh-line** writes a newline character and returns **t** if the output stream is not already at the beginning of a line; otherwise **fresh-line** does nothing and returns **nil**.

Syntax: **terpri** `&optional` *output-stream* [*Function*]

fresh-line `&optional` *output-stream* [*Function*]

Remarks: If the *output-stream* argument is not specified or is **nil**, the stream that is the current value of the variable ***standard-output*** is used. If *output-stream* is **t**, the stream that is the value of ***terminal-io*** is used.

Examples:

```
> (with-output-to-string (s)
    (format s "not an ")
    (format s "empty line")
    (terpri s)
    (terpri s)
    (format s "aftermath"))
"not an empty line

aftermath"
> (with-output-to-string (s)
    (format s "not an ")
    (format s "empty line")
    (fresh-line s)
    (fresh-line s)
    (format s "aftermath"))
"not an empty line
aftermath"
```

unread-char

Purpose: The function **unread-char** returns the specified character to the front of an input stream so that the character will be read again as the next character in that stream.

Syntax: **unread-char** *character* `&optional` *input-stream* [*Function*]

Remarks: The *character* argument must be the last character that was read from the given input stream.

The *input-stream* argument specifies the stream to be used. If it is not specified or is **nil**, the stream that is the value of the variable ***standard-input*** is used. If *input-stream* is **t**, the stream that is the value of ***terminal-io*** is used.

Examples:

```
> (with-input-from-string (is "0123")
    (dotimes (i 6)
      (let ((c (read-char is)))
        (if (evenp i) (format t "~S ~S~%" i c) (unread-char c is)))))
0 #\0
2 #\1
4 #\2
NIL
```

write, prin1, princ, print, pprint

Purpose: The functions **write**, **prin1**, **princ**, **print**, and **pprint** write the printed representation of an object to an output stream.

The function **write** is the general output function. It has the ability to specify all the parameters applicable to the printing of an object.

The functions **prin1**, **princ**, **print**, and **pprint** implicitly set certain print parameters to particular values. The remaining parameter values are taken from the global variables ***print-escape***, ***print-radix***, ***print-base***, ***print-circle***, ***print-pretty***, ***print-level***, ***print-length***, ***print-case***, ***print-gensym***, ***print-array***, and ***print-structure***.

Each of the functions **write**, **prin1**, **print**, and **princ** returns *object* as its value. The function **pprint** returns no values.

Syntax:

write *object* `&key :stream :escape :radix :base :circle :pretty :level :length :case :gensym :array :structure` [*Function*]

prin1 *object* `&optional` *output-stream* [*Function*]

princ *object* `&optional` *output-stream* [*Function*]

print *object* `&optional` *output-stream* [*Function*]

pprint *object* `&optional` *output-stream* [*Function*]

Remarks: The keyword argument **:stream** of **write** and the optional *output-stream* arguments of **prin1**, **princ**, **print**, and **pprint** specify the stream to which output is to be sent. If the argument is not specified or is **nil**, the stream that is the value of the variable ***standard-output*** is used. If **:stream** or *output-stream* is **t**, the stream that is the value of ***terminal-io*** is used.

The other keyword arguments of **write** are described below. If any keyword argument is not specified, its value is taken from the corresponding global variable, namely ***print-escape***, ***print-radix***, ***print-base***, ***print-circle***, ***print-pretty***, ***print-level***, ***print-length***, ***print-case***, ***print-gensym***, ***print-array***, or ***print-structure***.

- The **:escape** keyword argument controls the printing of escape characters. If the value of **:escape** is **nil**, the printing of escape characters is suppressed.
- The value of the **:radix** keyword argument determines whether a radix indicator is included with each integer or ratio printed. If the value of **:radix** is non-**nil**, a radix indicator is printed.

- The value of the **:base** keyword argument is the numerical base (radix) in which integers and ratios are printed.
- The **:circle** keyword argument controls the attempt to detect circularity in an object being printed. If **:circle** is non-**nil** and a circular object is detected, the #*n*= and #*n*# constructs are used to denote the circular structure. If **:circle** is **nil**, an attempt to print a circular object may cause Common Lisp to loop indefinitely.
- The **:pretty** keyword argument is used to control pretty-printing. If the value of **:pretty** is non-**nil**, additional whitespace characters are written in order to make printed expressions easier to read.
- The **:level** keyword argument is used to limit the amount of text output when an expression is printed. If an integer value is specified, the printing depth of the expression is limited to that value. A value of **nil** means that no print limit is imposed.
- The **:length** keyword argument is used to limit the amount of text output when an expression is printed. An integer value specifies the maximum number of consecutive elements printed at one level. An ellipsis (...) is used to represent further objects at that level. A value of **nil** means that no print limit is imposed.
- The **:case** keyword argument determines the case used in printing the names of symbols. The value of **:case** must be either **:upcase**, **:downcase**, or **:capitalize**. Corresponding to these three possible values, uppercase letters in symbol names are printed in uppercase, in lowercase, or in a combination of cases in which words are capitalized.
- The **:gensym** keyword argument controls the printing of the names of uninterned symbols. If **:gensym** is non-**nil**, the prefix #: is printed before the name of any uninterned symbol.
- The **:array** keyword argument controls the format in which arrays are printed. If the value of **:array** is non-**nil**, arrays are printed in their entirety with the #(, #*, or #*n*A syntax. If **:array** is **nil**, just enough is printed to identify the array, using the #<...> syntax.
- The **:structure** keyword argument controls the printing of structures. If the value of **:structure** is non-**nil**, structures are printed in detail, using the #S syntax. If **:structure** is **nil**, they are printed with the abbreviated #<...> syntax. The **:structure** keyword argument of **write** is an extension to Common Lisp.

The function **prin1** acts like **write** with **:escape t**, that is, escape characters are written where appropriate. This tends to make it possible to use **read** to read back the output of **prin1**.

The function **princ** acts like **write** with **:escape nil**. Thus no escape characters are written. This function is generally used when the output is to be read by humans, not by Common Lisp.

The function **print** acts like **write** with **:escape t**, but in addition it causes the output to begin with a newline character and to end with a space.

The function **pprint** acts like **write** with **:escape t :pretty t** but also causes the output to begin with a newline character.

Examples:

```
> (write #\a)
#\a
#\a
> (prin1 #\a)
#\a
#\a
> (print #\a)

#\a
#\a
> (princ #\a)
a
#\a
> (write '(let((a 1)(b 2))(+ a b)))
(LET ((A 1) (B 2)) (+ A B))
(LET ((A 1) (B 2)) (+ A B))
> (pprint '(let((a 1)(b 2))(+ a b)))

(LET ((A 1)
      (B 2))
  (+ A B))
> (write '(let((a 1)(b 2))(+ a b)) :pretty t)
(LET ((A 1)
      (B 2))
  (+ A B))
(LET ((A 1) (B 2)) (+ A B))
> (with-output-to-string (s)
    (write 'write :stream s)
    (prin1 'prin1 s))
"WRITEPRIN1"
```

See Also: ***print-escape***

print-radix

print-base

print-circle

print-pretty

print-level

print-length

print-case

print-gensym

print-array

print-structure

write-byte

Purpose: The function **write-byte** writes a single byte to a specified binary output stream.

The *integer* argument is written as a byte to the output stream specified by *binary-output-stream*. The *integer* argument must be of the type **:element-type** that was specified in the call to **open** or to **with-open-file** that created the stream.

Syntax: **write-byte** *integer binary-output-stream* [*Function*]

Remarks: The size of the byte written depends on the **:element-type** argument from **open** or **with-open-file**. Unless the byte size of that element type is one, two, or four bits, each call to **write-byte** generates an integral number of 8-bit bytes, namely the minimum number necessary to hold the number of bits indicated by the given element type. If the byte size of the element type is one, two, or four bits, then as many elements as possible (eight, four, or two respectively) are packed into each 8-bit byte.

Examples:

```
> (with-open-file (s "temp-bytes"
                     :direction :output
                     :element-type 'unsigned-byte)
    (write-byte 101 s))
101
> (with-open-file (s "temp-bytes" :element-type 'unsigned-byte)
    (format t "~S ~S" (read-byte s) (read-byte s nil 'eof)))
101 EOF
NIL
```

See Also: **read-byte**

write-char

Purpose: The function **write-char** writes a character to an output stream and returns the given character as its result.

Syntax: **write-char** *character* `&optional` *output-stream* [*Function*]

Remarks: If the *output-stream* argument is not specified or is **nil**, the stream that is the current value of the variable ***standard-output*** is used. If *output-stream* is **t**, the stream that is the value of ***terminal-io*** is used.

Examples:

```
> (write-char #\a)
a
#\a
> (with-output-to-string (s) (write-char #\b s))
"b"
```

write-line, write-string

Purpose: The functions **write-line** and **write-string** write a string to an output stream. The function **write-line** writes a newline character after the string, whereas **write-string** does not.

The output operation may be restricted to a substring of the original string by specifying the **:start** and **:end** keyword arguments.

Both **write-line** and **write-string** return the original string as a result.

Syntax:

write-string *string* `&optional` *output-stream* `&key :start :end` [*Function*]

write-line *string* `&optional` *output-stream* `&key :start :end` [*Function*]

Remarks: The **:start** and **:end** keyword arguments take integer values that specify offsets into the string. The **:start** argument marks the beginning position of the substring; the **:end** argument marks the position following the last element of the substring. The start value defaults to 0; the end value defaults to the length of the string.

If the *output-stream* argument is not specified or is **nil**, the stream that is the value of the variable ***standard-output*** is used. If *output-stream* is **t**, the stream that is the value of ***terminal-io*** is used.

If any keyword arguments are supplied to **write-line** or **write-string**, the optional argument must also be specified. Otherwise the first keyword is taken as the optional argument.

Examples:

```
> (write-string "beans")
beans
"beans"
> (write-line "limas" *standard-output* :end 4)
lima
"limas"
```

write-to-string, prin1-to-string, princ-to-string

Purpose: The functions **write-to-string**, **prin1-to-string**, and **princ-to-string** are used to create a string consisting of the printed representation of an object.

The function **write-to-string** is the general output function. It has the ability to specify all the parameters applicable to the printing of an object.

The functions **prin1-to-string** and **princ-to-string** implicitly set certain print parameters to particular values. The remaining parameter values are taken from the global variables ***print-escape***, ***print-radix***, ***print-base***, ***print-circle***, ***print-pretty***, ***print-level***, ***print-length***, ***print-case***, ***print-gensym***, ***print-array***, and ***print-structure***.

Each of these functions returns the created string as its result.

Syntax:

write-to-string *object* &key :escape :radix :base :circle :pretty :level :length :case :gensym :array :structure [*Function*]

prin1-to-string *object* [*Function*]

princ-to-string *object* [*Function*]

Remarks: The keyword arguments of **write-to-string** are described below. If any keyword argument is not specified, its value is taken from the corresponding global variable, namely ***print-escape***, ***print-radix***, ***print-base***, ***print-circle***, ***print-pretty***, ***print-level***, ***print-length***, ***print-case***, ***print-gensym***, ***print-array***, or ***print-structure***.

- The **:escape** keyword argument controls the printing of escape characters. If the value of **:escape** is **nil**, the printing of escape characters is suppressed.
- The value of the **:radix** keyword argument determines whether a radix indicator is included with each integer or ratio printed. If the value of **:radix** is non-**nil**, a radix indicator is printed.
- The value of the **:base** keyword argument is the numerical base (radix) in which integers and ratios are printed.
- The **:circle** keyword argument controls the attempt to detect circularity in an object being printed. If **:circle** is non-**nil** and a circular object is detected, the **#*n*=** and **#*n*#** constructs are used to denote the circular structure. If **:circle** is **nil**, an attempt to print a circular object may cause Common Lisp to loop indefinitely.

- The **:pretty** keyword argument is used to control pretty-printing. If the value of **:pretty** is non-**nil**, additional whitespace characters are written to make printed expressions easier to read.
- The **:level** keyword argument is used to limit the amount of text output when an expression is printed. If an integer value is specified, the printing depth of the expression is limited to that value. A value of **nil** means that no print limit is imposed.
- The **:length** keyword argument is used to limit the amount of text output when an expression is printed. An integer value specifies the maximum number of consecutive elements printed at one level. An ellipsis (...) is used to represent further objects at that level. A value of **nil** means that no print limit is imposed.
- The **:case** keyword argument determines the case used in printing the names of symbols. The value of **:case** must be either **:upcase**, **:downcase**, or **:capitalize**. Corresponding to these three possible values, uppercase letters in symbol names are printed in uppercase, in lowercase, or in a combination of cases in which words are capitalized.
- The **:gensym** keyword argument controls the printing of the names of uninterned symbols. If **:gensym** is non-**nil**, the prefix `#:` is printed before the name of any uninterned symbol.
- The **:array** keyword argument controls the format in which arrays are printed. If the value of **:array** is non-**nil**, arrays are printed in their entirety with the `#(`, `#*`, or `#nA` syntax. If **:array** is **nil**, just enough is printed to identify the array, using the `#<...>` syntax.
- The **:structure** keyword argument controls the printing of structures. If the value of **:structure** is non-**nil**, structures are printed in detail, using the `#S` syntax. If **:structure** is **nil**, they are printed with the abbreviated `#<...>` syntax. The **:structure** keyword argument of **write-to-string** is an extension to Common Lisp.

The function **prin1-to-string** acts like **write-to-string** with **:escape t**, that is, escape characters are written where appropriate.

The function **princ-to-string** acts like **write-to-string** with **:escape nil**. Thus no escape characters are written.

Examples:

```
> (prin1-to-string "abc")
"\"abc\""
> (princ-to-string "abc")
"abc"
```

See Also:

print-escape

print-radix

print-base

print-circle

print-pretty

print-level

print-length

print-case

print-gensym

print-array

print-structure

write

prin1

princ

y-or-n-p, yes-or-no-p

Purpose: The functions **y-or-n-p** and **yes-or-no-p** are used to ask questions of the user; they return **t** if the answer was "yes" and **nil** if the answer was "no."

The function **y-or-n-p** allows the user to answer the question with a `Y` or an `N`. It should be used if the question is anticipated or if the resulting decision is not of major impact.

The function **yes-or-no-p** requires the user to answer with either `YES` or `NO`. This function should be used for unexpected questions or for questions with possibly serious impact.

If the *format-control-string* argument is specified and is non-**nil**, the function **fresh-line** is called. Its output is followed by the output of the text specified by the format control string and by a list of response choices.

Syntax:

y-or-n-p `&optional` *format-control-string* `&rest` *arguments* [*Function*]

yes-or-no-p `&optional` *format-control-string* `&rest` *arguments* [*Function*]

Remarks: The stream that is the value of the variable ***query-io*** is used for all input and output operations.

Both **y-or-n-p** and **yes-or-no-p** ignore the alphabetic case in the user's answer.

The section "Formatted Output" explains how the format control string is interpreted and how the elements of *arguments* are processed.

Examples:

```
> (y-or-n-p "(t or nil) given by")
(t or nil) given by (Y or N) Y
T
> (yes-or-no-p "a ~S message" 'frightening)
a FRIGHTENING message (Yes or No) no
NIL
```

Chapter 22. File System Interface

Chapter 22. File System Interface

About the File System Interface

File systems are among the least standardized of the major features of computing environments. Since Common Lisp is designed for use under a variety of operating systems and file systems, it uses its own method for designating files, a method called ***pathnames***. A pathname consists of six fields: host, device, directory, name, type, and version. In many of the functions that require a pathname argument, the argument may be specified as a pathname, a string, or a stream associated with the file. When necessary, a pathname is converted into the description that is appropriate to the outside file system; that description is a string called a ***namestring***.

The six parts of a pathname are as follows:

- The ***host*** field specifies the name of the file system that contains the file. When a pathname is specified as a string, if a colon occurs in the string, anything preceding the colon is considered to be the name of the host. Although the host field may be specified as part of a pathname, this information is not used in Sun Common Lisp.
- The ***device*** field specifies the name of the physical or logical device that contains the file. Although the device field may be specified as part of a pathname, this information is not used in Sun Common Lisp.
- The ***directory*** field specifies the location of the file in the file system in terms of its directory structure.
- The ***name*** field specifies a particular set of related files. The source version and the compiled version of a Lisp program generally have the same name.
- The ***type*** field specifies the kind or format of the file. This is often the file extension.
- The ***version*** field specifies the file version. When a pathname is specified as a string and the character #, followed by a number, ends that string, the number is the version number of the corresponding file.

The keyword **:wild** may be used in a pathname argument in functions that permit it. This keyword indicates that the pathname component may match anything.

The keyword **:newest** may be used in a pathname to specify the most recent version of a file.

The function **equal** should be used when testing pathnames for equality.

Many functions described in this chapter, such as **delete-file**, **open**, and **rename-file**, are implemented using the functionality provided by the underlying operating system. Thus, their behavior may mirror that of the underlying primitives.

Categories of Operations

This section groups file system operations according to functionality.

Data Type Predicates

pathnamep

This predicate determines whether an object is a pathname.

Operations on Names

pathname	**pathname-type**
truename	**pathname-version**
make-pathname	**user-homedir-pathname**
merge-pathnames	**namestring**
rename-file	**file-namestring**
default-pathname-defaults	**directory-namestring**
pathname-host	**host-namestring**
pathname-device	**enough-namestring**
pathname-directory	**parse-namestring**
pathname-name	

These constructs manipulate pathnames and namestrings.

Opening Files

open	**with-open-file**

These functions open files.

Deleting Files

delete-file

This function deletes files.

Loading Files

load	***load-verbose***

These constructs are used to load files.

File Attributes

file-author	**file-write-date**
file-length	**probe-file**
file-position	

These functions provide information about files.

Directory Functions

directory

This function examines directories.

default-pathname-defaults

Purpose: The value of the variable ***default-pathname-defaults*** is a pathname. This pathname is used whenever a function needs a default pathname and one is not supplied.

Syntax: ***default-pathname-defaults*** [*Variable*]

Remarks: The default value of ***default-pathname-defaults*** is the working directory that was current when Lisp was started up.

Examples:

```
> *default-pathname-defaults*    ;current working directory is set to /etc
#P"/etc/"
> (setq *default-pathname-defaults* "/usr/bin/")
"/usr/bin/"
> (setq q "calendar")
"calendar"
> (merge-pathnames (make-pathname :name q))
#P"/usr/bin/calendar"
```

delete-file

Purpose: The function **delete-file** deletes the specified file. It returns a non-**nil** value if it succeeds.

Syntax: **delete-file** *file* [*Function*]

Remarks: The *file* argument is a pathname, a string, or a stream. If the *file* argument is an open stream associated with a file, that stream is closed and the file is deleted.

The pathname may not contain a **:wild** component.

The function **delete-file** is implemented by using operations provided by the operating system.

Examples:

```
> (with-open-file (s "/tmp/delete-test" :if-does-not-exist :create)
    (setq p (merge-pathnames s)))
#P"/tmp/delete-test"
> (probe-file p)
#P"/tmp/delete-test"
> (delete-file "/tmp/delete-test")
#P"/tmp/delete-test"
> (probe-file p)
NIL
> (setq p (merge-pathnames
            (setq s (open "/tmp/delete-test" :direction :output))))
#P"/tmp/delete-test"
> (probe-file p)
#P"/tmp/delete-test"
> (probe-file s)
#P"/tmp/delete-test"
> (delete-file s)
#P"/tmp/delete-test"
> (probe-file p)
NIL
> (probe-file s)
NIL
```

directory

Purpose: The function **directory** is used to examine a file system directory. It returns a list of the pathnames of all files in the system that correspond to the *pathname* argument.

Syntax: **directory** *pathname* [*Function*]

Remarks: The *pathname* argument is a pathname, a string, or a stream associated with a file.

Use of the keyword **:wild** is permitted in the pathname. It indicates that the corresponding pathname component may match anything.

Examples:

```
;;; assume that there is a subdirectory under /tmp that contains
;;; three files called a, b, and c

> (directory "/tmp/sub")
(#P"/tmp/sub")
> (directory "/tmp/sub/")
(#P"/tmp/sub/c" #P"/tmp/sub/b" #P"/tmp/sub/a")
```

enough-namestring

Purpose: The function **enough-namestring** converts its *pathname* argument into a namestring. It returns a shortened version of that namestring that is sufficient to identify the file when merged with the pathname *defaults.*

Syntax: **enough-namestring** *pathname* `&optional` *defaults* [*Function*]

Remarks: If the *defaults* argument is not specified, the value of the variable ***default-pathname-defaults*** is used.

The *pathname* argument is a pathname, a string, a symbol, or a stream that is or was associated with a file.

If *pathname* is a stream, the namestring is the name used to open the file. This name may be different from the true name of the file.

Examples:

```
> *default-pathname-defaults*      ;current working directory is set
#P"/etc/"                          ;to /etc
> (enough-namestring "passwd")
"passwd"
> (setq q (make-pathname :name "passwd"
                         :directory
                           (pathname-directory *default-pathname-defaults*)))
#P"/etc/passwd"
> (enough-namestring q *default-pathname-defaults*)
"passwd"
> (enough-namestring q "/etc/")
"passwd"
> (enough-namestring q)
"passwd"
```

See Also: **merge-pathnames**

namestring

file-author

Purpose: The function **file-author** returns the name of the author or last writer of the file as a string. If the name cannot be determined by the operating system, **file-author** returns **nil**.

Syntax: **file-author** *file* [*Function*]

Remarks: The *file* argument is a pathname, a string, or a stream that is open to a file.

Examples:

```
> *default-pathname-defaults*
#P"/etc/"
> (setq s (open (merge-pathnames "passwd")))
#<Stream BUFFERED-STREAM 101DB7CB>
> (file-author s)
"root"
> (file-author "/etc/passwd")
"root"
```

file-length

Purpose: The function **file-length** returns the length of the specified file as a nonnegative integer. If the length cannot be determined by the operating system, **file-length** returns **nil**.

Syntax: **file-length** *file-stream* [*Function*]

Remarks: The *file-stream* argument must be a stream that is open to a file.

The unit of length for a binary file is that specified by the **:element-type** argument used in the **open** command that created the stream.

Examples:

```
> (setq s (open "/tmp/file-len-test"
                :direction :output
                :if-exists :supersede))
#<Stream BUFFERED-STREAM 101DC673>
> (format s "0123456789")
NIL
> (file-length s)
10
```

See Also: **open**

file-position

Purpose: The function **file-position** returns the current position within a file or sets that position. Its action depends on whether the *position* argument is specified.

If the *position* argument is omitted, **file-position** returns a nonnegative integer that indicates the current position in the file. If this position cannot be determined by the operating system, **file-position** returns **nil**.

If the *position* argument is specified, **file-position** sets the current position in the file to that value. The position may be specified by a nonnegative integer, by the keyword **:start**, or by the keyword **:end**. If *position* is too large, an error is signaled. The function **file-position** returns a non-nil value if the operation succeeds; otherwise it returns **nil**.

Syntax: **file-position** *file-stream* &optional *position* [*Function*]

Remarks: The *file-stream* argument must be a stream that is open to a random-access file.

The position is given in units corresponding to the element type argument specified when the file was opened.

The value of the starting position is 0.

Examples:

```
> (setq s (open "/tmp/file-pos-test"
                :direction :output
                :if-exists :supersede))
#<Stream BUFFERED-STREAM 101DCF3B>
> (file-position s)
0
> (format s "abcdefghijklmnopqrstuvwxyzzyxwvutsrqponmlkjihgfedcba")
NIL
> (file-position s)
52
```

See Also: **open**

file-length

file-write-date

Purpose: The function **file-write-date** returns in universal time format the time at which the file was last written (or created). This value is an integer. If the time cannot be determined by the operating system, **file-write-date** returns **nil**.

Syntax: **file-write-date** *file* [*Function*]

Remarks: The *file* argument is a pathname, a string, or a stream that is open to a file.

load, *load-verbose*

Purpose: The function **load** reads the file specified by the *filename* argument and evaluates each form in that file. A non-**nil** value is returned if the operation is successful.

The variable ***load-verbose*** provides a default value for the **:verbose** argument of **load**. The initial value of ***load-verbose*** is **nil**.

Syntax: load *filename* `&key :verbose :print :if-does-not-exist` [*Function*]

load-verbose [*Variable*]

Remarks: The *filename* argument is a pathname, stream, string, or symbol. If the *filename* argument is not fully specified, default values are taken from the value of the variable ***default-pathname-defaults*** by using the function **merge-pathnames**.

If the file type is not specified and both source and binary versions of the file exist, **load** will use the binary version if it is more recent. If the source version is more recent, **load** asks the user to specify the action to be taken.

If the *filename* argument specifies a stream, **load** determines the type of the stream and loads directly from the stream.

The keyword arguments control details of the performance of the function **load**.

If the **:verbose** argument is non-**nil**, **load** prints information about its progress on the standard output. The default value of this argument is **nil**.

If the **:print** argument is non-**nil**, **load** prints the value of each expression that is loaded on the standard output. The default value of this argument is **nil**.

The **:if-does-not-exist** argument controls what happens if the specified file does not exist. The function **load** calls the function **open** with the **:if-does-not-exist** argument bound to this value. This value can be either **:error** (signal an error), **:create** (create an empty file), or **nil** (return **nil** from the function **load**). It defaults to **:error**.

Examples:

```
;;; assuming the file /test/load-test-file.lisp contains
;;;
;;; 1
;;; (setq a 888)
;;;
;;; then...

> (load "/test/load-test-file")
#P"/test/load-test-file.lisp"
> a
888
> (load (setq p (merge-pathnames "/test/load-test-file")) :verbose t)
;;; Loading source file "/test/load-test-file.lisp"
#P"/test/load-test-file.lisp"
> (load p :print t)
1
888
#P"/test/load-test-file.lisp"
```

See Also:

merge-pathnames

error

make-pathname

Purpose: The function **make-pathname** constructs a pathname from the specified keyword arguments and returns the result. Any fields of the pathname that are left unspecified are filled with the corresponding values from the **:defaults** argument.

Syntax: **make-pathname** `&key :host :device :directory :name :type :version :defaults` *[Function]*

Remarks: If the **:defaults** argument is not specified, the value of ***default-pathname-defaults*** is used by **merge-pathnames** to supply the host component, if it is missing. Any other missing components will be **nil**.

The **:directory** argument is a list of directories leading to the desired directory.

The **:version** argument is an integer.

The other keyword arguments are strings.

Although the host and device fields may be specified as part of a pathname, this information is not used in Sun Common Lisp.

Examples:

```
> *default-pathname-defaults*
#P"/etc/"
> (setq q (make-pathname :name "getty"
                         :directory
                            (pathname-directory *default-pathname-defaults*)))
#P"/etc/getty"
> (pathnamep q)
T
> (make-pathname :directory (list "dev") :name "ttya")
#P"/dev/ttya"
> (make-pathname :directory (list "usr" "bin") :name "calendar")
#P"/usr/bin/calendar"
> (make-pathname :host "edsel"
                 :directory (pathname-directory *default-pathname-defaults*)
                 :name "getty")
#P"edsel:/etc/getty"
> (make-pathname :directory (pathname-directory *default-pathname-defaults*)
                 :name "getty" :version 1)
#P"/etc/getty#1"
> (make-pathname :directory (list "usr" "include")
                 :name "stdio" :type "h")
#P"/usr/include/stdio.h"
```

See Also: **merge-pathnames**

merge-pathnames

Purpose: The function **merge-pathnames** constructs a pathname from the *pathname* argument by filling in any unspecified components with the corresponding values from the *defaults* and *default-version* arguments. It returns the resulting pathname.

Syntax: **merge-pathnames** *pathname* `&optional` *defaults default-version* [*Function*]

Remarks: The *pathname* and *defaults* arguments are each a pathname, a string, a symbol, or a stream.

If the *defaults* argument is not specified, the value of ***default-pathname-defaults*** is used.

If *pathname* specifies a host but not a device, the value for the device is taken from the *defaults* argument only if the two hosts agree. Otherwise the device is set to the default device for that host.

If *pathname* specifies a name but not a version, the version is copied from *default-version* rather than from *defaults*. If the *pathname* argument does not specify a name component, the version is copied from *defaults*, if *defaults* specifies a version. If *defaults* does not have a version, the version is copied from *default-version*. If the *default-version* argument is also not specified, the newest version is used. If *default-version* is **nil**, the version component will remain unspecified.

Examples:

```
> *default-pathname-defaults*                ;current working directory is set
#P"/etc/"                                    ;to /etc
> (merge-pathnames "")
#P"/etc/"
> (merge-pathnames "subdir")
#P"/etc/subdir"
> (setq q (merge-pathnames "subdir/"))
#P"subdir/"
> (truename q)
#P"/etc/subdir/"
> (merge-pathnames "/subdir/")
#P"/subdir/"
> (setq q (merge-pathnames "subdir/isfile"))
#P"subdir/isfile"
> (truename q)                               ;file /etc/subdir/isfile exists
#P"/etc/subdir/isfile"
```

```
> (setq q (merge-pathnames "subdir/" "isfile"))
#P"subdir/isfile"
> (truename q)
#P"/etc/subdir/isfile"
```

See Also: ***default-pathname-defaults***

namestring, file-namestring, directory-namestring, host-namestring

Purpose: These functions convert a *pathname* argument into a namestring and return the result.

The function **namestring** returns the namestring that corresponds to the specified pathname components. If the value of *pathname* is a string, that string is returned.

The function **file-namestring** returns only that part of the namestring that corresponds to the name, type, and version components of the pathname.

The function **directory-namestring** returns only that part of the namestring that corresponds to the directory component.

The function **host-namestring** returns only that part of the namestring that corresponds to the host component.

Syntax:

namestring *pathname* [*Function*]

file-namestring *pathname* [*Function*]

directory-namestring *pathname* [*Function*]

host-namestring *pathname* [*Function*]

Remarks: The *pathname* argument is a pathname, a string, a symbol, or a stream.

If *pathname* is a stream, the namestring is the name used to open the file. This name may be different from the true name of the file.

Examples:

```
> (namestring "getty")          ;current working directory is /etc
"getty"
> (setq q (make-pathname :host "edsel"
                         :directory
                           (pathname-directory *default-pathname-defaults*)
                         :name "getty"))
#P"edsel:/etc/getty"
> (file-namestring q)
"getty"
> (directory-namestring q)
"/etc/"
> (host-namestring q)
"edsel"
```

See Also: **truename**

open

Purpose: The function **open** opens the specified file and returns a stream that is connected to it.

Syntax: **open** *filename* `&key :direction :element-type :if-exists :if-does-not-exist` [*Function*]

Remarks: The *filename* argument may be a pathname, a string, or a stream.

The function **open** may be used with the following keyword arguments:

The **:direction** argument determines the direction of the stream. Its value is one of the following keywords:

- The keyword **:input** specifies an input stream; this is the default.
- The keyword **:output** specifies an output stream.
- The keyword **:io** specifies a bidirectional stream.
- The keyword **:probe** checks to see if the file exists.

The **:element-type** argument determines the basic unit for the stream. It must be a type-specifier for a bounded subrange of integer or character. If the keyword **:default** is specified or if no **:element-type** argument is given, a default character type is used.

The **:if-exists** argument is relevant if the stream is open for output and the specified file already exists. Its value is one of the following keywords:

- The keyword **:error** causes an error to be signaled if the file already exists. It is the default if the version is not **:newest**.
- The keyword **:new-version** causes the creation of a new file with the same name but a larger version number. It is the default if the version is **:newest**.
- The keyword **:rename** causes the old file to be renamed and a new file with the specified name to be created.
- The keyword **:rename-and-delete** causes the old file to be renamed and then deleted. A new file is created with the specified name.
- The keyword **:overwrite** uses the existing file. The file pointer is set to the beginning of the file. Output to the stream modifies the file.
- The keyword **:append** uses the existing file. Output to the stream modifies the file. The file pointer is set to the end of the file.

- The keyword **:supersede** replaces the existing file. It does not use a new version number.
- The keyword **nil** causes **open** to fail, returning **nil**.

The **:if-does-not-exist** argument specifies the action to be taken if the file does not exist. Its value is one of the following:

- The keyword **:error** signals an error if the file does not exist. This is the default if the direction is **:input** or if the **:if-exists** argument is **:append** or **:overwrite**.
- The keyword **:create** causes the creation of an empty file with the specified name. This is the default if the direction is **:output** or **:io**, or if the **:if-exists** argument is not **:append** or **:overwrite**.
- The value **nil** causes **open** to fail, returning **nil**. This is the default if the **:direction** argument is **:probe**.

Examples:

```
> (open "/dev/ttya" :direction :probe)
#<Stream %STREAM 101DB6B3>
> (setq q (merge-pathnames (user-homedir-pathname) "mbox"))
#P"/u/foo/mbox"                                  ;home directory is /u/foo
> (open q :direction :output :if-exists :append)
#<Stream BUFFERED-STREAM 101DCA23>
> (open "/tmp/bar" :if-does-not-exist :create)   ;file bar does not
#<Stream BUFFERED-STREAM 101DCA33>               ;exist in /tmp
> (setq s (open "/tmp/bar" :direction :probe))
#<Stream %STREAM 101DCD43>
> (truename s)
#P"/tmp/bar"
> (open s :direction :output :if-exists nil)
NIL
```

See Also: **with-open-file**

close

parse-namestring

Purpose: The function **parse-namestring** converts its *thing* argument into a pathname and returns the result.

Syntax: **parse-namestring** *thing* `&optional` *host defaults* `&key :start :end :junk-allowed` [*Function*]

Remarks: The *thing* argument may be a pathname, a string, a symbol, or a stream. If a symbol is specified, the print name of the symbol is used.

If the *host* argument is specified, the *thing* argument must not specify a host unless the two are the same. If neither *thing* nor *hosts* specifies a host, a host field is taken from *defaults*. This argument defaults to ***default-pathname-defaults***.

If a string or symbol (print name) is specified, the **:start** and **:end** keyword arguments may be used to restrict the parsing operation to a substring of the string. The keyword arguments **:start** and **:end** take integer values that specify offsets into the original string. The **:start** argument marks the beginning position of the substring; the **:end** argument marks the position following the last element of the substring. The start value defaults to 0; the end value defaults to the length of the string.

If **:junk-allowed** is **nil**, the entire substring is parsed, and the resulting pathname is returned. An error is signaled if the substring contains anything besides a valid pathname. If **:junk-allowed** is non-**nil**, the pathname that is parsed is returned. If a syntactically correct pathname is not found, **nil** is returned. The default value for **:junk-allowed** is **nil**.

A second value is also returned; this value is the index into the string that corresponds to the position following the last one that was part of the pathname.

If the *thing* argument is not a string or a symbol, the second value is the start value.

Note that if the keyword arguments are to be used, the optional arguments must also be given.

Examples:

```
> (setq q (parse-namestring "/etc/getty"))
#P"/etc/getty"
> (pathnamep q)
T
> (parse-namestring "getty")
#P"getty"
5
> (setq s (open "/usr/include/sys/types.h"))
#<Stream BUFFERED-STREAM 101DE06B>
> (parse-namestring s)
#P"/usr/include/sys/types.h"
0
> (parse-namestring "/usr/include/sys/types.h" nil nil :start 6 :end 15 )
#P"nclude/sy"
15
> (parse-namestring s nil nil :start 5 :end 12 )
#P"/usr/include/sys/types.h"
5
```

pathname

Purpose: The function **pathname** converts its argument into a pathname and returns the result.

Syntax: **pathname** *pathname* [*Function*]

Remarks: The *pathname* argument is a pathname, a string, a symbol, or a stream.

Examples:

```
> (pathname "/etc/getty")
#P"/etc/getty"
> (setq p (pathname '/etc/getty))
#P"/ETC/GETTY"
> (pathnamep p)
T
> (eq p (pathname p))                                  ;no conversion
T                                                      ;required
> (eq (pathname '/etc/getty) (pathname '/etc/getty))   ;creates new
NIL                                                    ;pathname structures
> (setq s (open "/etc/getty"))
#<Stream BUFFERED-STREAM 101E1503>
> (pathname s)
#P"/etc/getty"
```

pathname-host, pathname-device, pathname-directory, pathname-name, pathname-type, pathname-version

Purpose: These functions return the corresponding components of the *pathname* argument.

Syntax:

pathname-host *pathname* [*Function*]

pathname-device *pathname* [*Function*]

pathname-directory *pathname* [*Function*]

pathname-name *pathname* [*Function*]

pathname-type *pathname* [*Function*]

pathname-version *pathname* [*Function*]

Remarks: The *pathname* argument is a pathname, a string, a symbol, or a stream.

The device component is not used in Sun Common Lisp. The function **pathname-device** returns **nil**.

The version component is returned as an integer or **nil**.

The directory component is returned as a list of strings or symbols or as **nil**.

All other specified components are returned as strings or **nil**.

Examples:

```
> *default-pathname-defaults*      ;current working directory is
#P"/usr/lucid/src/"                ;/usr/lucid/src
> (pathname-directory "chase.c")
(:RELATIVE)
> (pathname-version "chase.c")
NIL
> (setq q (make-pathname :host "edsel"
                         :directory (list "usr" "lucid" "src")
                         :name "chase" :type "c"))
#P"edsel:/usr/lucid/src/chase.c"
> (pathname-host q)
"edsel"
> (pathname-directory q)
("usr" "lucid" "src")
> (pathname-name q)
"chase"
> (pathname-type q)
"c"
```

pathnamep

Purpose: The predicate **pathnamep** is true if its argument is a pathname; otherwise it is false.

Syntax: **pathnamep** *object* [*Function*]

Examples:

```
> (setq q "/etc/getty")
"/etc/getty"
> (pathnamep q)
NIL
> (setq q (pathname "/etc/getty"))
#P"/etc/getty"
> (pathnamep q)
T
```

probe-file

Purpose: The function **probe-file** tests whether a file exists. It returns **nil** if the specified file does not exist. Otherwise it returns the true name of the file.

Syntax: **probe-file** *file* [*Function*]

Remarks: The *file* argument is a pathname, a string, or a stream that is open to a file.

Examples:

```
> *default-pathname-defaults*
#P"/etc/"
> (probe-file (make-pathname
                 :directory (pathname-directory *default-pathname-defaults*)
                 :name "passwd"))
#P"/etc/passwd"
> (probe-file "/dev/ttya")
#P"/dev/ttya"
> (probe-file "/dev/nofile")
NIL
```

See Also: **truename**

open

rename-file

Purpose: The function **rename-file** modifies the file system in such a way that the file indicated by *file* is renamed to *new-name*.

The function **rename-file** returns three values. The first is *new-name* with any missing components supplied from *file*; the second and third values are the true names of the file, before and after renaming respectively.

Syntax: **rename-file** *file new-name* [*Function*]

Remarks: The *file* argument is a pathname, stream, or string. The *new-name* argument is a pathname, string, or symbol.

The function **rename-file** is implemented by using operations provided by the operating system.

Examples:

```
> (with-open-file (s "/tmp/rename-test.a" :if-does-not-exist :create)
    (setq p (merge-pathnames s)))
#P"/tmp/rename-test.a"
> (rename-file "/tmp/rename-test.a" "rename-test.b")
#P"/tmp/rename-test.b"
#P"/tmp/rename-test.a"
#P"/tmp/rename-test.b"
> (probe-file p)
NIL
```

See Also: **truename**

truename

Purpose: The function **truename** tries to find the file indicated by its *pathname* argument. If it finds that file, it returns the true name of that file, that is, it returns the name that the operating system considers to be the primary name for the file. This name is returned as a pathname. If the file cannot be found, **truename** signals an error.

Syntax: **truename** *pathname* [*Function*]

Remarks: The *pathname* argument is a pathname, a string, a symbol, or a stream.

Examples:

```
> *default-pathname-defaults*        ;current working directory is
#P"/usr/"                            ;set to /usr
> (setq s (open "/etc/passwd"))
#<Stream BUFFERED-STREAM 101DB763>
> (truename s)
#P"/etc/passwd"
> (truename "bin")
#P"/usr/bin"
```

user-homedir-pathname

Purpose: The function **user-homedir-pathname** returns a pathname that corresponds to the user's home directory on the machine *host*. The name, type, and version components that are returned are always **nil**. The function **user-homedir-pathname** returns **nil** if it cannot determine what the home directory is.

Syntax: **user-homedir-pathname** `&optional` *host* [*Function*]

Remarks: If the *host* argument is not specified, **user-homedir-pathname** always succeeds.

Examples:

```
> (pathnamep (user-homedir-pathname))
T
> (file-namestring (user-homedir-pathname))
""
```

with-open-file

Purpose: The macro **with-open-file** uses **open** to generate a stream that reads or writes the specified file. The macro then performs a specified series of actions on the open file.

The *options* arguments are passed as arguments to **open**. The stream that **open** returns is bound to the variable *stream*. The *form* arguments are then executed in order.

Syntax: **with-open-file** (*stream filename* {*options*}*){*declaration*}* {*form*}* [*Macro*]

Remarks: The *filename* argument specifies the file that is to be opened; *filename* is a pathname, a string, or a stream.

The file is closed automatically when **with-open-file** terminates or aborts.

Examples:

```
> (setq p (merge-pathnames "/tmp/w-open-fl.tmp"))
#P"/tmp/w-open-fl.tmp"
> (with-open-file (s p :direction :output :if-exists :supersede)
    (format s "Here are a couple~%of test data lines~%"))
NIL
> (with-open-file (s p)
    (do ((l (read-line s) (read-line s nil 'eof)))
        ((eq l 'eof) "Reached end-of-file")
     (format t "==>~A~%" l)))
==>Here are a couple
==>of test data lines
"Reached end-of-file"
```

See Also: **open**

close

Chapter 23. Errors

Chapter 23. Errors

About Errors

Common Lisp provides a variety of facilities for signaling errors.

An ***error*** causes Common Lisp to stop whatever it is doing and enter the debugger. The debugger then displays some information about what caused the error and the courses of action available to the user. Errors may be either continuable or fatal. A ***continuable error*** allows the program to regain control and to provide options for repairing the cause of the error. A ***fatal error*** is one that forces the current computation to end.

A ***warning*** causes a warning message to be issued. The user may specify whether a warning is to cause the debugger to be entered.

A ***break*** causes the debugger to be entered. It is possible to continue from a break. The break facility is intended for use in debugging.

The user is referred to the *Sun Common Lisp User's Guide* for a more detailed discussion of debugging facilities.

Categories of Operations

This section groups error-signaling operations according to functionality.

General Error-Signaling Facilities

error	**warn**
cerror	***break-on-warnings***
break	

These constructs are used to signal errors.

Specialized Error Checking

assert	**check-type**

These functions perform specific tests and may signal continuable errors.

assert

Purpose: The macro **assert** tests whether a given form evaluates to **nil**. If the specified test form evaluates to **nil**, **assert** signals a continuable error and causes the debugger to be entered. If the value of the test form is non-**nil**, **assert** returns **nil**.

If the user continues from such an error, **assert** prompts for new values for the generalized variables specified by the *place* arguments. It then re-evaluates the *test-form* argument. If *test-form* again evaluates to **nil**, **assert** once again signals a continuable error. When *test-form* evaluates to non-**nil**, **assert** returns **nil**.

Syntax: **assert** *test-form* [({*place*}*) [*format-string* {*arg*}*]] [*Macro*]

Remarks: Each *place* argument must be a generalized variable acceptable to the macro **setf**.

The function **format** is used to produce a string from the *format-string* and *arg* arguments. That string is passed on to the debugger for use as an error message. The *format-string* and *arg* arguments are evaluated only if an error is signaled.

Examples:

```
> (defun assert-example (x) (assert (numberp x) (x)) (+ x x))
ASSERT-EXAMPLE
> (compile 'assert-example)
;;; Compiling function ASSERT-EXAMPLE...assembling...emitting...done.
ASSERT-EXAMPLE
> (assert-example t)
>>Error: Assert form (NUMBERP X) failed.

ASSERT-RUNTIME:
   Required arg 0 (PLACES-LIST): (X)
   Required arg 1 (FORM): (NUMBERP X)
   Required arg 2 (STRING): NIL
   Required arg 3 (ARGS): NIL

:A    Abort to Lisp Top Level
:C    Replace some values and test the assertion again
-> :c
Replace some values and test the assertion again
Give a new value for X?  (Y or N) y
Form to evaluate and store as the value of X: 2
4
```

See Also: **format**

break

Purpose: The function **break** causes the debugger to be entered. It is possible to continue from a break.

When the user continues from the breakpoint, **break** returns **nil**.

Syntax: **break** `&optional` *format-string* `&rest` *args* [*Function*]

Remarks: The function **break** is intended for use in debugging; it allows the user to examine values and then continue the computation.

The function **format** is used to generate a string from the arguments to **break**. That string is passed on to the debugger for use as a break message.

Examples:

```
> (break "a break message with ~S" 'arguments)
>>Break: a break message with ARGUMENTS

EVAL:
   Required arg 0 (EXPRESSION): (BREAK "a break message with ~S" (QUOTE
ARGUMENTS))

:A    Abort to Lisp Top Level
:C    Return from break
-> :c
Return from break
NIL
```

See Also: **format**

cerror

Purpose: The function **cerror** signals a continuable error and causes the debugger to be entered.

If the user continues from the error, **cerror** returns **nil**.

Syntax: **cerror** *continue-format-string error-format-string* &rest *args* [*Function*]

Remarks: The function **format** is used to generate strings from the arguments to **cerror**. Those strings are passed on to the debugger. The *error-format-string* argument is used to produce an error message when the debugger is entered. The *continue-format-string* is used to describe directions for or the effect of continuing from the error.

Examples:

```
> (defun foo (a)
    (loop (when (numberp a) (return (1+ a)))
          (cerror "enter new value" "~s is not a number" a)
          (format t " -> ")
          (setq a (read))))
FOO
> (compile 'foo)
;;; Compiling function FOO...assembling...emitting...done.
FOO
> (foo 1)
2
> (foo t)
>>Error: T is not a number

FOO:
   Required arg 0 (A): T

:A    Abort to Lisp Top Level
:C    enter new value
-> :c
enter new value -> 2
3
```

See Also: **error**

format

check-type

Purpose: The macro **check-type** tests the type of the value of a generalized variable. It signals a continuable error if the value in *place* is not of type *typespec*. If the value is of the specified type, **check-type** returns **nil**.

If the user continues from such an error, **check-type** prompts for a new value for the generalized variable specified by the *place* argument. It then tests the type of the new value of the variable. If the value is not of the specified type, **check-type** once again signals a continuable error. When the value in *place* is of the specified type, **check-type** returns **nil**.

Syntax: **check-type** *place typespec* `&optional` *string* [*Macro*]

Remarks: The *place* argument must be a generalized variable acceptable to the macro **setf**.

The *typespec* argument is a type specifier; it is not evaluated.

The *string* argument should describe the desired type; it is used to provide an error message. If *string* is not present, a string is generated automatically from *typespec*.

Examples:

```
> (defun foo (a) (check-type a integer) (+ a a))
FOO
> (compile 'foo)
;;; Compiling function FOO...assembling...emitting...done.
FOO
> (foo 1)
2
> (foo t)
>>Error: T should be of type INTEGER

FOO:
   Required arg 0 (A): T

:A    Abort to Lisp Top Level
:C    Supply a new value
-> :c
Supply a new value
Enter a form to be evaluated: (+ 3 4)
Value is 7, OK? (Y or N) y
14
```

error

Purpose: The function **error** signals a fatal error and causes the debugger to be entered.

Syntax: **error** *format-string* &rest *args* [*Function*]

Remarks: The function **format** is used to generate a string from the arguments to **error**. That string is passed on to the debugger for use as an error message.

Examples:

```
> (error "Uncontinuable problem")
>>Error: Uncontinuable problem

EVAL:
   Required arg 0 (EXPRESSION): (ERROR "Uncontinuable problem")

:A    Abort to Lisp Top Level
-> :a
Back to Lisp Top Level

> (setq a 'foo)
FOO
> (if (numberp a)
      (1+ a)
      (error "~S is not a number" a))
>>Error: FOO is not a number

EVAL:
   Required arg 0 (EXPRESSION): (ERROR "~S is not a number" A)

:A    Abort to Lisp Top Level
-> :a
Back to Lisp Top Level
```

See Also: **cerror**

format

warn, *break-on-warnings*

Purpose: The function **warn** prints an error message and may cause the debugger to be entered.

The variable ***break-on-warnings*** controls the behavior of **warn**. If ***break-on-warnings*** is **nil**, **warn** prints an error message and returns **nil**. If the value of ***break-on-warnings*** is non-**nil**, **warn** also causes the debugger to be entered. When the user continues from the debugger, **warn** returns **nil**.

Syntax:

warn *format-string* `&rest` *args* [*Function*]

break-on-warnings [*Variable*]

Remarks: The initial value of ***break-on-warnings*** is **nil**.

The function **format** is used to generate a string from the arguments to **warn**. These arguments should provide the error message.

Examples:

```
> *break-on-warnings*
NIL
> (warn "caveat emptor")
;;; Warning: caveat emptor
NIL
> (let ((*break-on-warnings* t)) (warn "caveat emptor"))
>>Break on warning: caveat emptor

WARN:
   Required arg 0 (FORMAT-STRING): "caveat emptor"
   Rest arg (FORMAT-ARGS): NIL

:A    Abort to Lisp Top Level
:C    Return from break on warning
-> :c
Return from break on warning
NIL
```

See Also:

break

format

Chapter 24. Environmental Features

Chapter 24. Environmental Features

About Environmental Features

Common Lisp supplies various tools that allow the user to interact with the programming environment. These facilities include a compiler, debugging facilities, an editor, time functions, and functions for obtaining information about the current system.

The user is referred to the *Sun Common Lisp User's Guide* for a more detailed presentation of the compiler, debugging facilities, and the editor.

The Compiler

The ***compiler*** transforms Common Lisp code into a form that is more efficient to execute than interpreted code. Generally, compiled code behaves like its interpreted counterpart but does not do as much checking for errors.

Another important difference between the interpreter and the compiler lies in the treatment of declarations. Those 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.

The compiler is discussed in the chapter "Compiling Lisp Programs" of the *Sun Common Lisp User's Guide.*

Debugging Facilities

Sun Common Lisp provides extensive facilities for debugging programs. When an error or interrupt occurs, an interactive ***debugger*** is entered that allows the dynamic status of the program to be examined. The debugger is described in the chapter "Debugging Lisp Programs" of the *Sun Common Lisp User's Guide.*

The ***trace*** facility is a tool for debugging. It allows one or more functions to be traced and provides the ability to perform certain actions at the time a function is called or at the time it exits. The trace facility is described in the chapter "Tracing Functions" of the *Sun Common Lisp User's Guide.*

The ***step*** facility allows the user to examine program behavior by stepping through the evaluation of forms and functions. The **step** macro is intended for use on interpreted functions. The step facility is described in the chapter "Stepping Through an Evaluation" of the *Sun Common Lisp User's Guide.*

The ***inspector*** facility allows the user to inspect data structures. It displays the components of the selected objects and allows them to be modified. The inspector facility is described in the chapter "Inspecting Data Structures" of the *Sun Common Lisp User's Guide.*

Time

Common Lisp uses three formats to represent time:

In the ***Universal Time*** format, time is measured in seconds. The representation of an interval of time is the nonnegative integer that specifies the number of seconds in the interval. The representation of a particular time is the nonnegative integer that specifies the number of seconds from midnight Janunary 1, 1900 GMT until the particular time. Note that times prior to January 1, 1900 GMT cannot be represented.

In the ***Internal Time*** format, time is measured in implementation-dependent units. The representation of an interval is the nonnegative integer that specifies the number of units in the interval. The representation of a particular time is the number of units from an arbitrary time (for example, when the machine was booted) until that particular time.

The ***Decoded Time*** format is used only for a particular time. This format has the following nine fields:

- The second is an integer in the range $[0,60)$.
- The minute is an integer in the range $[0,60)$.
- The hour is an integer in the range $[0,24)$.
- The date is an integer in the range $[1,31]$; the actual upper bound of the interval depends on the month and year of the particular date.
- The month is an integer in the range $[1,12]$.
- The year is a nonnegative integer. If the integer is less than 100, it indicates the year in the range $[\text{current year} - 50, \text{current year} + 50)$ with those last two digits.
- The day of the week is an integer in the range $[0,6]$; 0 means Monday, 1 means Tuesday, and so on.
- The daylight-saving-time flag, if non-**nil**, indicates that daylight saving time is in effect.
- The time zone is an integer in the range $[0,24)$; it is the number of hours from GMT west to the particular time zone.

Categories of Operations

This section groups environmental features according to functionality.

Compilation

compile	**disassemble**
compile-file	

These functions are used to compile and disassemble code.

Debugging Facilities

trace	**apropos-list**
untrace	**arglist**
step	**describe**
inspect	**dribble**
apropos	

These functions are used in debugging.

Documentation

documentation

This function is used to add documentation to programs.

Editor

ed

This function invokes the editor.

Time Functions

get-decoded-time	**get-internal-run-time**
get-universal-time	**get-internal-real-time**
decode-universal-time	**time**
encode-universal-time	**sleep**
internal-time-units-per-second	

These functions provide timing facilities and information about time.

Other Environmental Functions

lisp-implementation-type	**software-version**
lisp-implementation-version	**short-site-name**
machine-type	**long-site-name**
machine-version	***features***
machine-instance	**room**
software-type	

These constructs provide information about the current implementation and the system on which Common Lisp is running.

Quitting Lisp

quit	**abort**

These functions terminate the current invocation of Lisp.

abort

Purpose: The function **abort** is used to exit from Lisp. It terminates the Lisp environment. It immediately returns the user to the operating system environment.

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

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

The function **abort** is an extension to Common Lisp.

See Also: **quit**

apropos, apropos-list

Purpose: The functions **apropos** and **apropos-list** are used to find all the symbols in the current environment whose print names contain a specified string.

The function **apropos** prints the names of the symbols that were found and information about the function definition and the value of those symbols on the standard output. It returns no values.

The function **apropos-list** returns a list of the symbols that were found.

Syntax: **apropos** *string* `&optional` *package* [*Function*]

apropos-list *string* `&optional` *package* [*Function*]

Remarks: The *string* argument may be either a string or a symbol. If it is a symbol, the symbol's print name is used.

If the *package* argument is specified, only symbols in that package will be found by either function.

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

arglist

Purpose: The function **arglist** returns a list that describes the arguments to a function.

Syntax: **arglist** *function* [*Function*]

Remarks: The *function* argument may be a function object or a symbol. If the argument is a function or a symbol that has a function definition, a list that describes the arguments of the function is returned. Otherwise, an error is signaled.

The function **arglist** is an extension to Common Lisp.

compile

Purpose: The function **compile** compiles an interpreted function in the current Lisp environment.

The function **compile** produces a compiled code object from a lambda expression. The lambda expression is specified by the *definition* argument if it is present; otherwise the function definition associated with the symbol *name* is used.

If the *name* argument is non-**nil**, **compile** sets the function definition associated with the specified symbol to the compiled code object and returns that symbol. Otherwise if *name* is **nil**, **compile** returns the compiled code object.

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

Remarks: Use of the compiler is discussed further in the *Sun Common Lisp User's Guide.*

Examples:

```
> (defun foo () "bar")
FOO
> (compiled-function-p #'foo)
NIL
> (compile 'foo)
;;; Compiling function FOO...assembling...emitting...done.
FOO
> (compiled-function-p #'foo)
T
> (setf (symbol-function 'foo)
        (compile nil #'(lambda () "replaced")))
;;; Compiling function...assembling...emitting...done.
#<Compiled-Function 4BEE9F>
> (foo)
"replaced"
```

See Also: **compile-file**

compile-file

Purpose: The function **compile-file** produces binary files from Lisp source files. It converts the file specified by the *input-pathname* argument into compiled code. The **:output-file** argument, if present, specifies where to put the compiled code.

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

Remarks: The binary file produced by **compile-file** overwrites any file with the same name.

If the **:output-file** option is specified, the corresponding argument should be a pathname or a string describing a valid filename. The binary file that is produced is given that name. If this option is not specified or if it is bound to **nil**, the binary file in question is named in the following way. If the source file has the extension `.lisp`, that extension is changed to `.lbin` if the value of the **:target** option is the default or to `.2bin` if the value of the **:target** option is 68020. Otherwise the extension `.lbin` or `.2bin` is concatenated to the end of the source filename.

The following keyword options are extensions to Common Lisp.

The keyword argument **:messages** controls 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 output.

The keyword argument **:warnings** 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 causes the warnings to be sent to the stream that is the value of ***error-output***.

If the **:fast-entry** keyword argument 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 of **:fast-entry** is **nil**.

If the **:tail-merge** keyword argument 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** keyword argument has a non-**nil** value, the compiler behaves as if all functions have been declared **notinline**. The default value of **:notinline** is **nil**.

compile-file

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 use of **compile-file** is discussed further in the *Sun Common Lisp User's Guide*.

See Also: **compile**

declare

decode-universal-time

Purpose: The function **decode-universal-time** converts a time from Universal Time format to Decoded Time format. It returns the time as nine values. These values correspond to the second, minute, hour, day, month, year, day of the week, a flag indicating whether the time is a daylight saving time value, and the time zone respectively.

Syntax: **decode-universal-time** *universal-time* `&optional` *time-zone* [*Function*]

Remarks: If the *time-zone* argument is not specified, the current time zone is used.

Examples:

```
> (decode-universal-time 0 0)
0
0
0
1
1
1900
0
NIL
0
```

See Also: **encode-universal-time**

get-universal-time

describe

Purpose: The function **describe** prints information about a given object on the standard output. The function **describe** returns no values.

Syntax: **describe** *object* [*Function*]

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

Examples:

```
> (describe '-)
#<Symbol 364FF5>

[0: NAME] "-"
[1: VALUE] (DESCRIBE (QUOTE -))
[2: FUNCTION] #<Compiled-Function - 27780F>
[3: PLIST] NIL
[4: PACKAGE] #<Package "LISP" 2E0003>
> (describe :test)
#<Symbol 362DBD>

[0: NAME] "TEST"
[1: VALUE] :TEST
[2: FUNCTION] Undefined
[3: PLIST] NIL
[4: PACKAGE] #<Package "KEYWORD" 2FE143>
> (describe "abc")
"abc"
```

See Also: **inspect**

disassemble

Purpose: The function **disassemble** disassembles a compiled function and prints the resulting code on the standard output. It returns **nil**.

Syntax: **disassemble** ***name-or-compiled-function*** [*Function*]

Remarks: If the argument is the name of an interpreted function, that function is first compiled and then disassembled. The function definition that is attached to the name as the content of the symbol's function cell is not changed.

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

documentation

Purpose: The function **documentation** returns the documentation string of type *doc-type* for a given symbol. If no documentation string is associated with the symbol, **documentation** returns **nil**.

Syntax: **documentation** *symbol doc-type* [*Function*]

Remarks: The *doc-type* argument is a symbol. It can be one of the following types: **variable**, **function**, **structure**, **type**, and **setf**.

The macro **setf** may be used with **documentation** to update the documentation for a symbol.

Examples:

```
> (defvar grz 0 "grz variable documentation")
GRZ
> (documentation 'grz 'variable)
"grz variable documentation"
```

dribble

Purpose: The function **dribble** is used to produce a record of input and output.

If a *pathname* argument is specified, **dribble** records input and output in the file indicated by the pathname.

If no argument is given, **dribble** terminates the recording of input and output and closes the file it has been using.

Syntax: **dribble** `&optional` *pathname* [*Function*]

Remarks: Only one dribble file may be in use at a time.

The *pathname* argument may be a pathname, string, stream, or symbol.

Examples:

```
> (dribble "/test/dribble-test")
;;; Dribble file #P"/test/dribble-test" started
T
> 'this-will-be-on-file
THIS-WILL-BE-ON-FILE
> (dribble)
;;; Dribble transcript to #P"/test/dribble-test" ended
T
```

ed

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

If no arguments are specified or the optional argument is **nil**, the editor is entered in the same state in which the user last left it.

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

Remarks: The optional argument *x* may be specified as a pathname, a string, or a symbol. If either a pathname or a string is specified, **ed** allows the user to edit the contents of the corresponding file. If a symbol argument that represents the name of an interpreted function is specified, **ed** pretty-prints the corresponding function into a buffer that becomes the current buffer. The user may then edit the text of the function definition. Any interpreted function that is edited must be reevaluated in order for the changes to become 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. Buffers and window configurations that were established earlier in the current Lisp session are restored.

If the **:windows** keyword argument is specified, it can 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 made to the window configuration in previous editing sessions are not retained. If no window environment is available, an error is signaled.

If the editor is started up as a terminal editor, it cannot be used in the window environment in subsequent editing sessions. Similarly, if the editor starts up in the window environment, subsequent editing sessions must remain in the window environment.

The user can also specify any 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**, the user can determine if a window environment is available.

The editor, the Window Tool Kit, and the functions **initialize-windows** and **windows-available-p** are described in the *Sun Common Lisp User's Guide.*

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

encode-universal-time

Purpose: The function **encode-universal-time** converts a time from Decoded Time format to Universal Time format and returns the resulting value.

Syntax: **encode-universal-time** *second minute hour date month year* &optional *time-zone* [*Function*]

Remarks: The *time-zone* argument defaults to the current time zone. This default value is adjusted for daylight saving time, if necessary. If the time zone is specified, there is no adjustment for daylight saving time.

Examples:

```
> (encode-universal-time 0 0 0 1 1 1900 0)
0
```

See Also: **decode-universal-time**

get-decoded-time

features

Purpose: The value of the variable ***features*** is a list of symbols. These symbols are the names of features that are provided by the current implementation of Common Lisp.

Syntax: ***features*** [*Variable*]

Remarks: The features **:common-lisp**, **:compiler**, and **:lucid** are some of the features that are known. These symbols are in the keyword package.

The ***features*** variable is used by the #+ and #- syntax in the Lisp reader. The constructs #+*feature* and #-*feature* control the reading of a given form based on the presence or absence of the feature in the ***features*** list. The reader and the #+ and #- syntax are discussed in the chapter "Input/Output."

get-decoded-time

Purpose: The function **get-decoded-time** returns the current time in Decoded Time format. It returns the time as nine values. These values correspond to the second, minute, hour, day, month, year, day of the week, a flag indicating whether the time is a daylight saving time value, and the time zone respectively.

Syntax: **get-decoded-time** *[Function]*

Examples:

```
> (get-decoded-time)        ;run at Tue May 13 15:35:10 PDT 1986
10
35
15
13
5
1986
1
T
8
```

See Also: **decode-universal-time**

encode-universal-time

get-internal-real-time

Purpose: The function **get-internal-real-time** returns the current time in Internal Time format. The result is an integer.

Syntax: **get-internal-real-time** [*Function*]

get-internal-run-time

Purpose: The function **get-internal-run-time** returns the current run time in Internal Time format. The result is an integer.

Syntax: **get-internal-run-time** *[Function]*

get-universal-time

Purpose: The function **get-universal-time** returns the current time in Universal Time format. The result is an integer.

Syntax: **get-universal-time** *[Function]*

inspect

Purpose: The function **inspect** is used for examining data structures. When called on an object, **inspect** prints information about the object and its components on the standard output. This function gives the user interactive control over what is printed out and allows the user to modify the given object. It returns as its value the last object examined.

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

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

The inspect facility is described in the *Sun Common Lisp User's Guide.*

See Also: **describe**

internal-time-units-per-second

Purpose: The value of the constant **internal-time-units-per-second** is an integer that defines the number of internal time units that are in one second.

Syntax: **internal-time-units-per-second** [*Constant*]

Remarks: These units form the basis of the Internal Time format representation.

lisp-implementation-type, lisp-implementation-version

Purpose: The functions **lisp-implementation-type** and **lisp-implementation-version** return strings that identify the current implementation of Common Lisp.

The function **lisp-implementation-type** returns a generic name for the Lisp.

The function **lisp-implementation-version** returns a detailed version name for the Lisp.

Syntax:

lisp-implementation-type [*Function*]

lisp-implementation-version [*Function*]

Examples:

```
> (lisp-implementation-type)
"Sun Common Lisp"
```

machine-type, machine-version, machine-instance

Purpose: The functions **machine-type**, **machine-version**, and **machine-instance** return strings that identify the machine on which the current instance of Common Lisp is running.

The function **machine-type** returns a generic name for the hardware.

The function **machine-version** returns a detailed version name for the hardware.

The function **machine-instance** returns the name of the specific machine.

Syntax:

machine-type [*Function*]

machine-version [*Function*]

machine-instance [*Function*]

quit

Purpose: The function **quit** is used to exit from Lisp. It terminates the Lisp environment.

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.

The function **quit** uses the special form **throw** to exit to the top level of Lisp before returning to the operating system environment. Thus, if **quit** is called from inside the special form **unwind-protect**, all the cleanup forms specified by the invocation of **unwind-protect** are executed before returning to the operating system environment. Thus, **quit** can be used to close all files before exiting Lisp.

The function **quit** is an extension to Common Lisp.

See Also: **abort**

throw

unwind-protect

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***.

Memory management is discussed in the *Sun Common Lisp User's Guide.*

short-site-name, long-site-name

Purpose: The functions **short-site-name** and **long-site-name** return strings that identify the location of the machine on which the current instance of Common Lisp is running.

The function **short-site-name** returns a short or abbreviated name.

The function **long-site-name** returns the full name.

Syntax:

short-site-name [*Function*]

long-site-name [*Function*]

Remarks: These strings are set by the user at installation time.

sleep

Purpose: The function **sleep** causes Common Lisp to pause for at least the specified number of seconds.

The function **sleep** returns **nil**.

Syntax: **sleep** *seconds* [*Function*]

Remarks: The *seconds* argument is an integer.

Examples:

```
> (sleep 1)
NIL
```

software-type, software-version

Purpose: The functions **software-type** and **software-version** return strings that identify the software on which Common Lisp is running.

The function **software-type** returns a generic name for the software.

The function **software-version** returns a detailed version name for the software.

Syntax: **software-type** *[Function]*

software-version *[Function]*

Examples:

```
> (software-type)
"UNIX"
```

step

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

The **step** macro evaluates a given form or function and allows the user to intervene during the course of evaluation. It returns the result of evaluating the form.

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

Remarks: The *function-name* argument is an extension to Common Lisp.

The step facility is described in the *Sun Common Lisp User's Guide.*

time

Purpose: The macro **time** evaluates its argument and returns the result. It prints timing statistics about the execution of the form on the stream that is the value of the variable ***trace-output***.

Syntax: **time** *form* [*Macro*]

Remarks: The accuracy of the results depends on the accuracy of the corresponding functions provided by the underlying operating system.

trace, untrace

Purpose: The macros **trace** and **untrace** control the invocation of the trace facility.

The macro **trace** with arguments *trace-spec* traces the specified functions. It returns as its value a list of the function names. If **trace** is called with no arguments, a list of all functions that are currently being traced is returned.

The macro **untrace** with arguments *function-name* untraces the specified functions. The macro **untrace** with no arguments untraces all the functions currently being traced.

If a function is already being traced, **trace** calls **untrace** before starting the new trace.

Calling **trace** on a macro traces the macro expansion, not the evaluation of the form. Special forms cannot be traced.

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

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

Remarks: Each *trace-spec* argument is the name of a function or a list consisting of the function name followed by keyword options. These keyword options are extensions to Common Lisp. They describe the circumstances under which the function is to be traced.

The trace facility and the use of its keyword extensions are described further in the *Sun Common Lisp User's Guide.*

See Also: **step**

Appendix A. Alphabetical Listing of Common Lisp Functions

This appendix is a listing of all Common Lisp functions, macros, constants, variables, and special forms, including all extensions to Common Lisp described in this 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` [*Function*]
`:messages`
`:warnings`
`:fast-entry`
`:tail-merge`
`:notinline`
`:target`

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* | {*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 contained in this manual. They are categorized according to use.

Note: These extensions are not part of the Common Lisp specification.

Program Structure

define-function *name function* [*Function*]

redefinition-action [*Variable*]

Macros

define-macro *name function* [*Function*]

The Evaluator

decache-eval [*Function*]

grindef &rest *function-name* [*Macro*]

prompt [*Variable*]

source-code *function* [*Function*]

Packages

delete-package *package* [*Function*]

Numbers

fixnump *object* [*Function*]

Lists

assq *object a-list* [*Function*]

list-nreverse *list* [*Function*]

list-reverse *list* [*Function*]

memq *object list* [*Function*]

Input/Output

ignore-extra-right-parens [*Variable*]

print-structure [*Variable*]

pp-line-length [*Variable*]

Environmental Features

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

arglist *function* [*Function*]

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

Index

A

B

C

D

G

H

I

K

L

M

N

O

P

Q

R

S

U

V

W

Y

Z

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