

**Declaration DYNAMIC-EXTENT****Syntax:**

```
(dynamic-extent [[var* | (function fn)*]])
```

**Arguments:**

*var*—a *variable name*.

*fn*—a *function name*.

**Valid Context:**

*declaration*

**Binding Types Affected:**

*variable, function*

**Description:**

In some containing *form*, *F*, this declaration asserts for each *vari* (which need not be bound by *F*), and for each *value* *vij* that *vari* takes on, and for each *object* *xijk* that is an *otherwise inaccessible part* of *vij* at any time when *vij* becomes the value of *vari*, that just after the execution of *F* terminates, *xijk* is either *inaccessible* (if *F* established a *binding* for *vari*) or still an *otherwise inaccessible part* of the current value of *vari* (if *F* did not establish a *binding* for *vari*). The same relation holds for each *fni*, except that the *bindings* are in the *function namespace*.

The compiler is permitted to use this information in any way that is appropriate to the *implementation* and that does not conflict with the semantics of Common Lisp.

**dynamic-extent** declarations can be *free declarations* or *bound declarations*.

The *vars* and *fns* named in a **dynamic-extent** declaration must not refer to *symbol macro* or *macro* bindings.

**Examples:**

Since stack allocation of the initial value entails knowing at the *object*'s creation time that the *object* can be *stack-allocated*, it is not generally useful to make a **dynamic-extent declaration** for *variables* which have no lexically apparent initial value. For example, it is probably useful to write:

```
(defun f ()
  (let ((x (list 1 2 3)))
    (declare (dynamic-extent x))
    ...))
```

This would permit those compilers that wish to do so to *stack allocate* the list held by the local variable *x*. It is permissible, but in practice probably not as useful, to write:

```
(defun g (x) (declare (dynamic-extent x)) ...)
(defun f () (g (list 1 2 3)))
```

## CLHS: Declaration DYNAMIC-EXTENT

Most compilers would probably not stack allocate the argument to g in f because it would be a modularity violation for the compiler to assume facts about g from within f. Only an implementation that was willing to be responsible for recompiling f if the definition of g changed incompatibly could legitimately stack allocate the list argument to g in f.

Here is another example:

```
(declaim (inline g))
(defun g (x) (declare (dynamic-extent x)) ...)
(defun f () (g (list 1 2 3)))

(defun f ()
  (flet ((g (x) (declare (dynamic-extent x)) ...))
    (g (list 1 2 3))))
```

In the previous example, some compilers might determine that optimization was possible and others might not.

A variant of this is the so-called "stack allocated rest list" that can be achieved (in implementations supporting the optimization) by:

```
(defun f (&rest x)
  (declare (dynamic-extent x))
  ...)
```

Note that although the initial value of x is not explicit, the f function is responsible for assembling the list x from the passed arguments, so the f function can be optimized by the compiler to construct a stack-allocated list instead of a heap-allocated list in implementations that support such.

In the following example,

```
(let ((x (list 'a1 'b1 'c1))
      (y (cons 'a2 (cons 'b2 (cons 'c2 nil)))))
  (declare (dynamic-extent x y))
  ...)
```

The otherwise inaccessible parts of x are three conses, and the otherwise inaccessible parts of y are three other conses. None of the symbols a1, b1, c1, a2, b2, c2, or nil is an otherwise inaccessible part of x or y because each is interned and hence accessible by the package (or packages) in which it is interned. However, if a freshly allocated uninterned symbol had been used, it would have been an otherwise inaccessible part of the list which contained it.

```
; ; In this example, the implementation is permitted to stack allocate
; ; the list that is bound to X.
(let ((x (list 1 2 3)))
  (declare (dynamic-extent x))
  (print x)
  :done)
>> (1 2 3)
=> :DONE

; ; In this example, the list to be bound to L can be stack-allocated.
(defun zap (x y z)
  (do ((l (list x y z) (cdr l)))
      ((null l)))
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(declare (dynamic-extent l))
(prinl (car l))) => ZAP
(zap 1 2 3)
=> 123
=> NIL

;; Some implementations might open-code LIST-ALL-PACKAGES in a way
;; that permits using stack allocation of the list to be bound to L.
(do ((l (list-all-packages) (cdr l)))
  ((null l))
  (declare (dynamic-extent l))
  (let ((name (package-name (car l))))
    (when (string-search "COMMON-LISP" name) (print name))))
=> "COMMON-LISP"
=> "COMMON-LISP-USER"
=> NIL

;; Some implementations might have the ability to stack allocate
;; rest lists. A declaration such as the following should be a cue
;; to such implementations that stack-allocation of the rest list
;; would be desirable.
(defun add (&rest x)
  (declare (dynamic-extent x))
  (apply #'+ x)) => ADD
(add 1 2 3) => 6

(defun zap (n m)
  ;; Computes (RANDOM (+ M 1)) at relative speed of roughly O(N).
  ;; It may be slow, but with a good compiler at least it
  ;; doesn't waste much heap storage. :-}
  (let ((a (make-array n)))
    (declare (dynamic-extent a))
    (dotimes (i n)
      (declare (dynamic-extent i))
      (setf (aref a i) (random (+ i 1))))
    (aref a m))) => ZAP
(< (zap 5 3) 3) => true
```

The following are in error, since the value of *x* is used outside of its *extent*:

```
(length (list (let ((x (list 1 2 3))) ; Invalid
                (declare (dynamic-extent x))
                x)))
(progn (let ((x (list 1 2 3))) ; Invalid
         (declare (dynamic-extent x))
         x)
       nil)
```

**See Also:**

**declare**

**Notes:**

The most common optimization is to *stack allocate* the initial value of the *objects* named by the *vars*.

It is permissible for an implementation to simply ignore this declaration.

***Declaration FTYPE*****Syntax:**

```
(ftype type function-name*)
```

**Arguments:**

*function-name*—a function name.

*type*—a type specifier.

**Valid Context:**

declaration or proclamation

**Binding Types Affected:**

function

**Description:**

Specifies that the functions named by *function-names* are of the functional type *type*. For example:

```
(declare (ftype (function (integer list) t) ith)
          (ftype (function (number) float) sine cosine))
```

If one of the functions mentioned has a lexically apparent local definition (as made by flet or labels), then the declaration applies to that local definition and not to the global function definition. ftype declarations never apply to variable bindings (see type).

The lexically apparent bindings of *function-names* must not be macro definitions. (This is because ftype declares the functional definition of each function name to be of a particular subtype of function, and macros do not denote functions.)

ftype declarations can be free declarations or bound declarations. ftype declarations of functions that appear before the body of a flet or labels form that defines that function are bound declarations. Such declarations in other contexts are free declarations.

**See Also:**

declare, declaim, proclaim

***Declaration IGNORE, IGNORABLE*****Syntax:**

```
(ignore {var | (function fn)}*)
```

```
(ignorable {var | (function fn)}*)
```

**Arguments:**

*var*—a variable name.

*fn*—a function name.

**Valid Context:**

declaration

**Binding Types Affected:**

variable, function

**Description:**

The **ignore** and **ignorable** declarations refer to for-value references to variable bindings for the *vars* and to function bindings for the *fns*.

An **ignore declaration** specifies that for-value references to the indicated bindings will not occur within the scope of the declaration. Within the scope of such a declaration, it is desirable for a compiler to issue a warning about the presence of either a for-value reference to any *var* or *fn*, or a special declaration for any *var*.

An **ignorable declaration** specifies that for-value references to the indicated bindings might or might not occur within the scope of the declaration. Within the scope of such a declaration, it is not desirable for a compiler to issue a warning about the presence or absence of either a for-value reference to any *var* or *fn*, or a special declaration for any *var*.

When not within the scope of a **ignore** or **ignorable declaration**, it is desirable for a compiler to issue a warning about any *var* for which there is neither a for-value reference nor a special declaration, or about any *fn* for which there is no for-value reference.

Any warning about a "used" or "unused" binding must be of type style-warning, and may not affect program semantics.

The stream variables established by with-open-file, with-open-stream, with-input-from-string, and with-output-to-string, and all iteration variables are, by definition, always "used". Using (declare (ignore *v*)), for such a variable *v* has unspecified consequences.

**See Also:**

declare

**Declaration INLINE, NOTINLINE****Syntax:**

(inline *function-name*\* )

## CLHS: Declaration DYNAMIC-EXTENT

(notinline *function-name*\*)

### Arguments:

*function-name*—a function name.

### Valid Context:

declaration or proclamation

### Binding Types Affected:

function

### Description:

**inline** specifies that it is desirable for the compiler to produce inline calls to the functions named by *function-names*; that is, the code for a specified *function-name* should be integrated into the calling routine, appearing "in line" in place of a procedure call. A compiler is free to ignore this declaration. **inline** declarations never apply to variable bindings.

If one of the functions mentioned has a lexically apparent local definition (as made by flet or labels), then the declaration applies to that local definition and not to the global function definition.

While no conforming implementation is required to perform inline expansion of user-defined functions, those implementations that do attempt to recognize the following paradigm:

To define a function *f* that is not **inline** by default but for which (declare (**inline** *f*)) will make *f* be locally inlined, the proper definition sequence is:

```
(declare (inline f))
(defun f ...)
(declaim (notinline f))
```

The **inline** proclamation preceding the defun form ensures that the compiler has the opportunity save the information necessary for inline expansion, and the notinline proclamation following the defun form prevents *f* from being expanded inline everywhere.

**notinline** specifies that it is undesirable to compile the functions named by *function-names* in-line. A compiler is not free to ignore this declaration; calls to the specified functions must be implemented as out-of-line subroutine calls.

If one of the functions mentioned has a lexically apparent local definition (as made by flet or labels), then the declaration applies to that local definition and not to the global function definition.

In the presence of a compiler macro definition for *function-name*, a notinline declaration prevents that compiler macro from being used. An **inline** declaration may be used to encourage use of compiler macro definitions. **inline** and notinline declarations otherwise have no effect when the lexically visible definition of *function-name* is a macro definition.

**inline** and notinline declarations can be free declarations or bound declarations. **inline** and notinline

## CLHS: Declaration DYNAMIC-EXTENT

declarations of functions that appear before the body of a flet or labels form that defines that function are bound declarations. Such declarations in other contexts are free declarations.

### Examples:

```
; ; The globally defined function DISPATCH should be open-coded,  
; ; if the implementation supports inlining, unless a NOTINLINE  
; ; declaration overrides this effect.  
(declare (inline dispatch))  
(defun dispatch (x) (funcall (get (car x) 'dispatch) x))  
; ; Here is an example where inlining would be encouraged.  
(defun top-level-1 () (dispatch (read-command)))  
; ; Here is an example where inlining would be prohibited.  
(defun top-level-2 ()  
  (declare (notinline dispatch))  
  (dispatch (read-command)))  
; ; Here is an example where inlining would be prohibited.  
(declare (notinline dispatch))  
(defun top-level-3 () (dispatch (read-command)))  
; ; Here is an example where inlining would be encouraged.  
(defun top-level-4 ()  
  (declare (inline dispatch))  
  (dispatch (read-command)))
```

### See Also:

declare, claim, proclaim

## Declaration OPTIMIZE

### Syntax:

```
(optimize {quality | (quality value)}*)
```

### Arguments:

*quality*—an optimize quality.

*value*—one of the integers 0, 1, 2, or 3.

### Valid Context:

declaration or proclamation

**Binding Types Affected:** None.

### Description:

## CLHS: Declaration DYNAMIC-EXTENT

Advises the compiler that each *quality* should be given attention according to the specified corresponding *value*. Each *quality* must be a symbol naming an optimize quality; the names and meanings of the standard *optimize qualities* are shown in the next figure.

Name	Meaning
<u>compilation-speed</u>	speed of the compilation process
<u>debug</u>	ease of debugging
<u>safety</u>	run-time error checking
<u>space</u>	both code size and run-time space
<u>speed</u>	speed of the object code

**Figure 3–25. Optimize qualities**

There may be other, implementation-defined optimize qualities.

A *value* 0 means that the corresponding *quality* is totally unimportant, and 3 that the *quality* is extremely important; 1 and 2 are intermediate values, with 1 the neutral value. (*quality* 3) can be abbreviated to *quality*.

Note that code which has the optimization (`safety 3`), or just **safety**, is called safe code.

The consequences are unspecified if a *quality* appears more than once with different values.

### Examples:

```
(defun often-used-subroutine (x y)
  (declare (optimize (safety 2)))
  (error-check x y)
  (hairy-setup x)
  (do ((i 0 (+ i 1))
       (z x (cdr z)))
      ((null z))
    ; This inner loop really needs to burn.
    (declare (optimize speed))
    (declare (fixnum i))
  ))
```

### See Also:

[declare](#), [declaim](#), [proclaim](#), [Section 3.3.4 \(Declaration Scope\)](#)

### Notes:

An optimize declaration never applies to either a variable or a function binding. An optimize declaration can only be a free declaration. For more information, see [Section 3.3.4 \(Declaration Scope\)](#).

## Declaration SPECIAL

### Syntax:

```
(special var*)
```

### Arguments:

*var*—*a symbol*.

### Valid Context:

*declaration or proclamation*

### Binding Types Affected:

*variable*

### Description:

Specifies that all of the *vars* named are dynamic. This specifier affects variable *bindings* and affects references. All variable *bindings* affected are made to be dynamic *bindings*, and affected variable references refer to the current dynamic *binding*. For example:

```
(defun hack (thing *mod*)      ;The binding of the parameter
  (declare (special *mod*))   ; *mod* is visible to hack1,
  (hack1 (car thing)))       ; but not that of thing.
(defun hack1 (arg)
  (declare (special *mod*))   ;Declare references to *mod*
  (if (atom arg) *mod*      ;within hack1 to be special.
    (cons (hack1 (car arg)) (hack1 (cdr arg)))))
```

A **special** declaration does not affect inner *bindings* of a *var*; the inner *bindings* implicitly shadow a **special** declaration and must be explicitly re-declared to be **special**. **special** declarations never apply to function *bindings*.

**special** declarations can be either *bound declarations*, affecting both a binding and references, or *free declarations*, affecting only references, depending on whether the declaration is attached to a variable binding.

When used in a *proclamation*, a **special declaration specifier** applies to all *bindings* as well as to all references of the mentioned variables. For example, after

```
(declaim (special x))
```

then in a function definition such as

```
(defun example (x) ...)
```

the parameter *x* is bound as a dynamic variable rather than as a lexical variable.

### Examples:

```
(defun declare-eg (y)                      ;this y is special
  (declare (special y))
  (let ((y t))                            ;this y is lexical
    (list y
           (locally (declare (special y) y)))) ;this y refers to the
                                                ;special binding of y
=> DECLARE-EG
(declare-eg nil) => (T NIL)
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(setf (symbol-value 'x) 6)
(defun foo (x)                                ;a lexical binding of x
  (print x)
  (let ((x (1+ x)))                          ;a special binding of x
    (declare (special x))                    ;and a lexical reference
    (bar))
  (1+ x))
(defun bar ()
  (print (locally (declare (special x))
                  x)))
(foo 10)
>> 10
>> 11
=> 11

(setf (symbol-value 'x) 6)                      ;[1] 1st occurrence of x
(defun bar (x y)                            ;[2] 2nd occurrence of x -- same as 1st occurrence
  (let ((old-x x)                         ;[3] 3rd occurrence of x
        (x y))
    (declare (special x))
    (list old-x x)))
(bar 'first 'second) => (FIRST SECOND)

(defun few (x &optional (y *foo*))
  (declare (special *foo*))
  ...)
```

The reference to `*foo*` in the first line of this example is not **special** even though there is a **special** declaration in the second line.

```
(declaim (special prosp)) => implementation-dependent
(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)

(declaim (special x))          ;x is always special.
(defun example (x y)
  (declare (special y))
  (let ((y 3) (x (* x 2)))
    (print (+ y (locally (declare (special y)) y))))
    (let ((y 4)) (declare (special y)) (foo x)))) => EXAMPLE
```

In the contorted code above, the outermost and innermost *bindings* of `y` are dynamic, but the middle binding is lexical. The two arguments to `+` are different, one being the value, which is 3, of the lexical variable `y`, and the other being the value of the dynamic variable named `y` (a *binding* of which happens, coincidentally, to lexically surround it at an outer level). All the *bindings* of `x` and references to `x` are dynamic, however, because of the proclamation that `x` is always **special**.

**See Also:**

**defparameter, defvar**

## **Declaration TYPE**

### **Syntax:**

(**type** *typespec* *var*\* )

(*typespec* *var*\* )

### **Arguments:**

*typespec*—a type specifier.

*var*—a variable name.

### **Valid Context:**

declaration or proclamation

### **Binding Types Affected:**

variable

### **Description:**

Affects only variable bindings and specifies that the *vars* take on values only of the specified *typespec*. In particular, values assigned to the variables by setq, as well as the initial values of the *vars* must be of the specified *typespec*. type declarations never apply to function bindings (see ftype).

A type declaration of a symbol defined by symbol–macrolet is equivalent to wrapping a the expression around the expansion of that symbol, although the symbol's macro expansion is not actually affected.

The meaning of a type declaration is equivalent to changing each reference to a variable (*var*) within the scope of the declaration to (the *typespec* *var*), changing each expression assigned to the variable (*new-value*) within the scope of the declaration to (the *typespec* *new-value*), and executing (the *typespec* *var*) at the moment the scope of the declaration is entered.

A type declaration is valid in all declarations. The interpretation of a type declaration is as follows:

1. During the execution of any reference to the declared variable within the scope of the declaration, the consequences are undefined if the value of the declared variable is not of the declared type.
2. During the execution of any setq of the declared variable within the scope of the declaration, the consequences are undefined if the newly assigned value of the declared variable is not of the declared type.
3. At the moment the scope of the declaration is entered, the consequences are undefined if the value of the declared variable is not of the declared type.

A type declaration affects only variable references within its scope.

If nested type declarations refer to the same variable, then the value of the variable must be a member of the intersection of the declared types.

## CLHS: Declaration DYNAMIC-EXTENT

If there is a local type declaration for a dynamic variable, and there is also a global type proclamation for that same variable, then the value of the variable within the scope of the local declaration must be a member of the intersection of the two declared types.

type declarations can be free declarations or bound declarations.

A symbol cannot be both the name of a type and the name of a declaration. Defining a symbol as the name of a class, structure, condition, or type, when the symbol has been declared as a declaration name, or vice versa, signals an error.

Within the lexical scope of an array type declaration, all references to array elements are assumed to satisfy the expressed array element type (as opposed to the upgraded array element type). A compiler can treat the code within the scope of the array type declaration as if each access of an array element were surrounded by an appropriate the form.

### Examples:

```
(defun f (x y)
  (declare (type fixnum x y))
  (let ((z (+ x y)))
    (declare (type fixnum z))
    z)) => F
(f 1 2) => 3
;; The previous definition of F is equivalent to
(defun f (x y)
  ; This declaration is a shorthand form of the TYPE declaration
  (declare (fixnum x y))
  ; To declare the type of a return value, it's not necessary to
  ; create a named variable. A THE special form can be used instead.
  (the fixnum (+ x y))) => F
(f 1 2) => 3

(defvar *one-array* (make-array 10 :element-type '(signed-byte 5)))
(defvar *another-array* (make-array 10 :element-type '(signed-byte 8)))

(defun frob (an-array)
  (declare (type (array (signed-byte 5) 1) an-array))
  (setf (aref an-array 1) 31)
  (setf (aref an-array 2) 127)
  (setf (aref an-array 3) (* 2 (aref an-array 3)))
  (let ((foo 0))
    (declare (type (signed-byte 5) foo))
    (setf foo (aref an-array 0)))))

(frob *one-array*)
(frob *another-array*)
```

The above definition of frob is equivalent to:

```
(defun frob (an-array)
  (setf (the (signed-byte 5) (aref an-array 1)) 31)
  (setf (the (signed-byte 5) (aref an-array 2)) 127)
  (setf (the (signed-byte 5) (aref an-array 3))
        (* 2 (the (signed-byte 5) (aref an-array 3)))))
  (let ((foo 0))
    (declare (type (signed-byte 5) foo))
    (setf foo (the (signed-byte 5) (aref an-array 0)))))
```

## CLHS: Declaration DYNAMIC-EXTENT

Given an implementation in which *fixnums* are 29 bits but fixnum arrays are upgraded to signed 32-bit arrays, the following could be compiled with all *fixnum* arithmetic:

```
(defun bump-counters (counters)
  (declare (type (array fixnum *) bump-counters))
  (dotimes (i (length counters))
    (incf (aref counters i))))
```

**See Also:**

[declare](#), [claim](#), [proclaim](#)

**Notes:**

(*typespec var\**) is an abbreviation for (type *typespec var\**).

A type declaration for the arguments to a function does not necessarily imply anything about the type of the result. The following function is not permitted to be compiled using implementation-dependent fixnum-only arithmetic:

```
(defun f (x y) (declare (fixnum x y)) (+ x y))
```

To see why, consider (f most-positive-fixnum 1). Common Lisp defines that F must return a bignum here, rather than signal an error or produce a mathematically incorrect result. If you have special knowledge such "fixnum overflow" cases will not come up, you can declare the result value to be in the fixnum range, enabling some compilers to use more efficient arithmetic:

```
(defun f (x y)
  (declare (fixnum x y))
  (the fixnum (+ x y)))
```

Note, however, that in the three-argument case, because of the possibility of an implicit intermediate value growing too large, the following will not cause implementation-dependent fixnum-only arithmetic to be used:

```
(defun f (x y)
  (declare (fixnum x y z))
  (the fixnum (+ x y z)))
```

To see why, consider (f most-positive-fixnum 1 -1). Although the arguments and the result are all *fixnums*, an intermediate value is not a *fixnum*. If it is important that implementation-dependent fixnum-only arithmetic be selected in implementations that provide it, consider writing something like this instead:

```
(defun f (x y)
  (declare (fixnum x y z))
  (the fixnum (+ (the fixnum (+ x y)) z)))
```

## Condition Type ARITHMETIC-ERROR

**Class Precedence List:**

[arithmetic-error](#), [error](#), [serious-condition](#), [condition](#), [t](#)

**Description:**

The *type arithmetic-error* consists of error conditions that occur during arithmetic operations. The operation and operands are initialized with the initialization arguments named :operation and :operands to **make-condition**, and are accessed by the functions **arithmetic-error-operation** and **arithmetic-error-operands**.

**See Also:**

[arithmetic-error-operation](#), [arithmetic-error-operands](#)

**Condition Type CELL-ERROR****Class Precedence List:**

[cell-error](#), [error](#), [serious-condition](#), [condition](#), [t](#)

**Description:**

The *type cell-error* consists of error conditions that occur during a location *access*. The name of the offending cell is initialized by the :name initialization argument to **make-condition**, and is accessed by the function **cell-error-name**.

**See Also:**

[cell-error-name](#)

**Condition Type CONDITION****Class Precedence List:**

[condition](#), [t](#)

**Description:**

All types of *conditions*, whether error or non-error, must inherit from this *type*.

No additional *subtype* relationships among the specified *subtypes* of *type condition* are allowed, except when explicitly mentioned in the text; however implementations are permitted to introduce additional *types* and one of these *types* can be a *subtype* of any number of the *subtypes* of *type condition*.

Whether a user-defined *condition type* has *slots* that are accessible by *with-slots* is *implementation-dependent*. Furthermore, even in an *implementation* in which user-defined *condition types* would have *slots*, it is *implementation-dependent* whether any *condition types* defined in this document have such *slots* or, if they do, what their *names* might be; only the reader functions documented by this specification may be relied upon by portable code.

*Conforming code* must observe the following restrictions related to *conditions*:

\* *define-condition*, not *defclass*, must be used to define new *condition types*.

- \* **make-condition**, not **make-instance**, must be used to create condition objects explicitly.
- \* The :report option of **define-condition**, not **defmethod** for **print-object**, must be used to define a condition reporter.
- \* **slot-value**, **slot-boundp**, **slot-makunbound**, and **with-slots** must not be used on condition objects. Instead, the appropriate accessor functions (defined by **define-condition**) should be used.

## **Condition Type CONTROL-ERROR**

Class Precedence List:

**control-error**, **error**, **serious-condition**, **condition**, **t**

Description:

The type control-error consists of error conditions that result from invalid dynamic transfers of control in a program. The errors that result from giving **throw** a tag that is not active or from giving **go** or **return-from** a tag that is no longer dynamically available are of type control-error.

## **Condition Type DIVISION-BY-ZERO**

Class Precedence List:

**division-by-zero**, **arithmetic-error**, **error**, **serious-condition**, **condition**, **t**

Description:

The type division-by-zero consists of error conditions that occur because of division by zero.

## **Condition Type END-OF-FILE**

Class Precedence List:

**end-of-file**, **stream-error**, **error**, **serious-condition**, **condition**, **t**

Description:

The type end-of-file consists of error conditions related to read operations that are done on streams that have no more data.

See Also:

**stream-error-stream**

## **Condition Type ERROR**

Class Precedence List:

**error**, **serious-condition**, **condition**, **t**

Description:

The type error consists of all conditions that represent errors.

## **Condition Type FILE-ERROR**

Class Precedence List:

file-error, error, serious-condition, condition, t

Description:

The type file-error consists of error conditions that occur during an attempt to open or close a file, or during some low-level transactions with a file system. The "offending pathname" is initialized by the :pathname initialization argument to make-condition, and is accessed by the function file-error-pathname.

See Also:

file-error-pathname, open, probe-file, directory, ensure-directories-exist

## **Condition Type FLOATING-POINT-INEXACT**

Class Precedence List:

floating-point-inexact, arithmetic-error, error, serious-condition, condition, t

Description:

The type floating-point-inexact consists of error conditions that occur because of certain floating point traps.

It is implementation-dependent whether floating point traps occur, and whether or how they may be enabled or disabled. Therefore, conforming code may establish handlers for this condition, but must not depend on its being *signaled*.

## **Condition Type FLOATING-POINT-OVERFLOW**

Class Precedence List:

floating-point-overflow, arithmetic-error, error, serious-condition, condition, t

Description:

The type floating-point-overflow consists of error conditions that occur because of floating-point overflow.

## **Condition Type FLOATING-POINT-UNDERFLOW**

Class Precedence List:

floating-point-underflow, arithmetic-error, error, serious-condition, condition, t

**Description:**

The *type floating-point-underflow* consists of error conditions that occur because of floating-point underflow.

**Condition Type FLOATING-POINT-INVALID-OPERATION****Class Precedence List:**

floating-point-invalid-operation, arithmetic-error, error, serious-condition, condition, t

**Description:**

The *type floating-point-invalid-operation* consists of error conditions that occur because of certain floating point traps.

It is *implementation-dependent* whether floating point traps occur, and whether or how they may be enabled or disabled. Therefore, conforming code may establish handlers for this condition, but must not depend on its being *signaled*.

**Condition Type PARSE-ERROR****Class Precedence List:**

parse-error, error, serious-condition, condition, t

**Description:**

The *type parse-error* consists of error conditions that are related to parsing.

**See Also:**

parse-namestring, reader-error

**Condition Type PACKAGE-ERROR****Class Precedence List:**

package-error, error, serious-condition, condition, t

**Description:**

The *type package-error* consists of *error conditions* related to operations on *packages*. The offending *package* (or *package name*) is initialized by the :packageinitialization argument to make-condition, and is accessed by the function package-error-package.

**See Also:**

package-error-package, Section 9 (Conditions)

**Condition Type PRINT-NOT-READABLE****Class Precedence List:**[print-not-readable](#), [error](#), [serious-condition](#), [condition](#), [t](#)**Description:**

The [type print-not-readable](#) consists of error conditions that occur during output while \*print-readably\* is true, as a result of attempting to write a printed representation with the Lisp printer that would not be correctly read back with the Lisp reader. The object which could not be printed is initialized by the :objectinitialization argument to [make-condition](#), and is accessed by the function print-not-readable-object.

**See Also:**[print-not-readable-object](#)**Condition Type PROGRAM-ERROR****Class Precedence List:**[program-error](#), [error](#), [serious-condition](#), [condition](#), [t](#)**Description:**

The [type program-error](#) consists of error conditions related to incorrect program syntax. The errors that result from naming a go tag or a block tag that is not lexically apparent are of type program-error.

**Condition Type READER-ERROR****Class Precedence List:**[reader-error](#), [parse-error](#), [stream-error](#), [error](#), [serious-condition](#), [condition](#), [t](#)**Description:**

The [type reader-error](#) consists of error conditions that are related to tokenization and parsing done by the Lisp reader.

**See Also:**[read](#), [stream-error-stream](#), [Section 23.1 \(Reader Concepts\)](#)**Condition Type SERIOUS-CONDITION****Class Precedence List:**[serious-condition](#), [condition](#), [t](#)

**Description:**

All *conditions* serious enough to require interactive intervention if not handled should inherit from the *type serious-condition*. This condition type is provided primarily so that it may be included as a *superclass* of other *condition types*; it is not intended to be signaled directly.

**Notes:**

Signaling a *serious condition* does not itself force entry into the debugger. However, except in the unusual situation where the programmer can assure that no harm will come from failing to *handle a serious condition*, such a *condition* is usually signaled with *error* rather than *signal* in order to assure that the program does not continue without *handling the condition*. (And conversely, it is conventional to use *signal* rather than *error* to signal conditions which are not *serious conditions*, since normally the failure to handle a non-serious condition is not reason enough for the debugger to be entered.)

**Condition Type SIMPLE-CONDITION****Class Precedence List:**

[simple-condition](#), [condition](#), [t](#)

**Description:**

The *type simple-condition* represents *conditions* that are signaled by *signal* whenever a *format-control* is supplied as the function's first argument. The *format control* and *format arguments* are initialized with the initialization arguments named :*format-control* and :*format-arguments* to *make-condition*, and are accessed by the *functions simple-condition-format-control* and *simple-condition-format-arguments*. If format arguments are not supplied to *make-condition*, *nil* is used as a default.

**See Also:**

[simple-condition-format-control](#), [simple-condition-format-arguments](#)

**Condition Type SIMPLE-ERROR****Class Precedence List:**

[simple-error](#), [simple-condition](#), [error](#), [serious-condition](#), [condition](#), [t](#)

**Description:**

The *type simple-error* consists of *conditions* that are signaled by *error* or *cerror* when a *format control* is supplied as the function's first argument.

**Condition Type SIMPLE-TYPE-ERROR****Class Precedence List:**

[simple-type-error](#), [simple-condition](#), [type-error](#), [error](#), [serious-condition](#), [condition](#), [t](#)

**Description:**

*Conditions* of type **simple-type-error** are like *conditions* of type **type-error**, except that they provide an alternate mechanism for specifying how the *condition* is to be *reported*; see the type **simple-condition**.

**See Also:**

[simple-condition](#), [simple-condition-format-control](#), [simple-condition-format-arguments](#),  
[type-error-datum](#), [type-error-expected-type](#)

**Condition Type SIMPLE-WARNING****Class Precedence List:**

[simple-warning](#), [simple-condition](#), [warning](#), [condition](#), [t](#)

**Description:**

The type **simple-warning** represents *conditions* that are signaled by **warn** whenever a *format control* is supplied as the function's first argument.

**Condition Type STREAM-ERROR****Class Precedence List:**

[stream-error](#), [error](#), [serious-condition](#), [condition](#), [t](#)

**Description:**

The type **stream-error** consists of error conditions that are related to receiving input from or sending output to a *stream*. The "offending stream" is initialized by the :streaminitialization argument to [make-condition](#), and is accessed by the [function stream-error-stream](#).

**See Also:**

[stream-error-stream](#)

**Condition Type STORAGE-CONDITION****Class Precedence List:**

[storage-condition](#), [serious-condition](#), [condition](#), [t](#)

**Description:**

The type **storage-condition** consists of serious conditions that relate to problems with memory management that are potentially due to *implementation-dependent* limits rather than semantic errors in *conforming programs*, and that typically warrant entry to the debugger if not handled. Depending on the details of the *implementation*, these might include such problems as stack overflow, memory region overflow, and storage exhausted.

**Notes:**

While some Common Lisp operations might signal *storage-condition* because they are defined to create *objects*, it is unspecified whether operations that are not defined to create *objects* create them anyway and so might also signal **storage-condition**. Likewise, the evaluator itself might create *objects* and so might signal **storage-condition**. (The natural assumption might be that such *object* creation is naturally inefficient, but even that is *implementation-dependent*.) In general, the entire question of how storage allocation is done is *implementation-dependent*, and so any operation might signal **storage-condition** at any time. Because such a *condition* is indicative of a limitation of the *implementation* or of the *image* rather than an error in a *program*, *objects* of *type storage-condition* are not of *type error*.

**Condition Type STYLE-WARNING****Class Precedence List:**

**style-warning**, **warning**, **condition**, **t**

**Description:**

The *type style-warning* includes those *conditions* that represent *situations* involving *code* that is *conforming code* but that is nevertheless considered to be faulty or substandard.

**See Also:**

**muffle-warning**

**Notes:**

An *implementation* might signal such a *condition* if it encounters *code* that uses deprecated features or that appears unaesthetic or inefficient.

An 'unused variable' warning must be of *type style-warning*.

In general, the question of whether *code* is faulty or substandard is a subjective decision to be made by the facility processing that *code*. The intent is that whenever such a facility wishes to complain about *code* on such subjective grounds, it should use this *condition type* so that any clients who wish to redirect or muffle superfluous warnings can do so without risking that they will be redirecting or muffling other, more serious warnings.

**Condition Type TYPE-ERROR****Class Precedence List:**

**type-error**, **error**, **serious-condition**, **condition**, **t**

**Description:**

The *type type-error* represents a situation in which an *object* is not of the expected type. The "offending datum" and "expected type" are initialized by the initialization arguments named :datum and :expected-type to **make-condition**, and are accessed by the functions **type-error-datum** and

**type-error-expected-type**

See Also:

**type-error-datum, type-error-expected-type****Condition Type UNBOUND-VARIABLE**

Class Precedence List:

**unbound-variable, cell-error, error, serious-condition, condition, t**

Description:

The *type unbound-variable* consists of *error conditions* that represent attempts to *read* the *value* of an *unbound variable*.

The name of the cell (see cell-error) is the *name* of the *variable* that was *unbound*.

See Also:

**cell-error-name****Condition Type UNBOUND-SLOT**

Class Precedence List:

**unbound-slot, cell-error, error, serious-condition, condition, t**

Description:

The *object* having the unbound slot is initialized by the :instanceinitialization argument to make-condition, and is accessed by the function unbound-slot-instance.

The name of the cell (see cell-error) is the name of the slot.

See Also:

**cell-error-name, unbound-slot-object, Section 9.1 (Condition System Concepts)****Condition Type UNDEFINED-FUNCTION**

Class Precedence List:

**undefined-function, cell-error, error, serious-condition, condition, t**

Description:

The *type undefined-function* consists of *error conditions* that represent attempts to *read* the definition of an *undefined function*.

## CLHS: Declaration DYNAMIC-EXTENT

The name of the cell (see [cell-error](#)) is the *function name* which was *funbound*.

See Also:

[cell-error-name](#)

### **Condition Type WARNING**

Class Precedence List:

[warning, condition, t](#)

Description:

The *type warning* consists of all types of warnings.

See Also:

[style-warning](#)

### **Function 1+, 1-**

Syntax:

**1+** *number* => *successor*

**1-** *number* => *predecessor*

Arguments and Values:

*number*—[a number](#).

*successor, predecessor*—[a number](#).

Description:

**1+** returns a [number](#) that is one more than its argument *number*. **1-** returns a [number](#) that is one less than its argument *number*.

Examples:

```
(1+ 99) => 100
(1- 100) => 99
(1+ (complex 0.0)) => #C(1.0 0.0)
(1- 5/3) => 2/3
```

Affected By: None.

Exceptional Situations:

## CLHS: Declaration DYNAMIC-EXTENT

Might signal **type-error** if its argument is not a number. Might signal **arithmetic-error**.

See Also:

**incf, decf**

Notes:

```
(1+ number) == (+ number 1)
(1- number) == (- number 1)
```

Implementors are encouraged to make the performance of both the previous expressions be the same.

### ***Function –***

Syntax:

- *number* => *negation*
- *minuend &rest subtrahends+* => *difference*

Arguments and Values:

*number, minuend, subtrahend*—–a number.

*negation, difference*—–a number.

Description:

The function – performs arithmetic subtraction and negation.

If only one *number* is supplied, the negation of that *number* is returned.

If more than one argument is given, it subtracts all of the *subtrahends* from the *minuend* and returns the result.

The function – performs necessary type conversions.

Examples:

```
(- 55.55) => -55.55
(- #c(3 -5)) => #C(-3 5)
(- 0) => 0
(eq1 (- 0.0) -0.0) => true
(- #c(100 45) #c(0 45)) => 100
(- 10 1 2 3 4) => 0
```

Affected By: None.

Exceptional Situations:

Might signal **type-error** if some argument is not a number. Might signal **arithmetic-error**.

**See Also:**

**Notes:** None.

## **Function ABORT, CONTINUE, MUFFLE-WARNING, STORE-VALUE, USE-VALUE**

**Syntax:**

**abort** &optional condition =>|

**continue** &optional condition => nil

**muffle-warning** &optional condition =>|

**store-value** value &optional condition => nil

**use-value** value &optional condition => nil

**Arguments and Values:**

*value*—an object.

*condition*—a condition object, or nil.

**Description:**

Transfers control to the most recently established applicable restart having the same name as the function. That is, the function abort searches for an applicable abort restart, the function continue searches for an applicable continue restart, and so on.

If no such restart exists, the functions continue, store-value, and use-value return nil, and the functions abort and muffle-warning signal an error of type control-error.

When *condition* is non-nil, only those restarts are considered that are either explicitly associated with that *condition*, or not associated with any condition; that is, the excluded restarts are those that are associated with a non-empty set of conditions of which the given *condition* is not an element. If *condition* is nil, all restarts are considered.

**Examples:**

```
;; Example of the ABORT restart

(defmacro abort-on-error (&body forms)
  `(handler-bind ((error #'abort))
     ,@forms) => ABORT-ON-ERROR
(abort-on-error (+ 3 5)) => 8
(abort-on-error (error "You lose."))
```

## CLHS: Declaration DYNAMIC-EXTENT

```

>> Returned to Lisp Top Level.

;;; Example of the CONTINUE restart

(defun real-sqrt (n)
  (when (minusp n)
    (setq n (- n))
    (cerror "Return sqrt(~D) instead." "Tried to take sqrt(-~D)." n))
  (sqrt n))

(real-sqrt 4) => 2
(real-sqrt -9)
>> Error: Tried to take sqrt(-9).
>> To continue, type :CONTINUE followed by an option number:
>> 1: Return sqrt(9) instead.
>> 2: Return to Lisp Toplevel.
>> Debug> (continue)
>> Return sqrt(9) instead.
=> 3

(handler-bind ((error #'(lambda (c) (continue))))
  (real-sqrt -9)) => 3

;;; Example of the MUFFLE-WARNING restart

(defun count-down (x)
  (do ((counter x (1- counter)))
      ((= counter 0) 'done)
      (when (= counter 1)
        (warn "Almost done"))
      (format t "~&~D~%" counter)))
=> COUNT-DOWN
(count-down 3)
>> 3
>> 2
>> Warning: Almost done
>> 1
=> DONE
(defun ignore-warnings-while-counting (x)
  (handler-bind ((warning #'ignore-warning))
    (count-down x)))
=> IGNORE-WARNINGS-WHILE-COUNTING
(defun ignore-warning (condition)
  (declare (ignore condition))
  (muffle-warning))
=> IGNORE-WARNING
(ignore-warnings-while-counting 3)
>> 3
>> 2
>> 1
=> DONE

;;; Example of the STORE-VALUE and USE-VALUE restarts

(defun careful-symbol-value (symbol)
  (check-type symbol symbol)
  (restart-case (if (boundp symbol)
                  (return-from careful-symbol-value
                               (symbol-value symbol))
                  (error 'unbound-variable
                         :name symbol)))

```

## CLHS: Declaration DYNAMIC-EXTENT

```
(use-value (value)
  :report "Specify a value to use this time."
  value)
(store-value (value)
  :report "Specify a value to store and use in the future."
  (setf (symbol-value symbol) value)))
(setq a 1234) => 1234
(careful-symbol-value 'a) => 1234
(makunbound 'a) => A
(careful-symbol-value 'a)
>> Error: A is not bound.
>> To continue, type :CONTINUE followed by an option number.
>> 1: Specify a value to use this time.
>> 2: Specify a value to store and use in the future.
>> 3: Return to Lisp Toplevel.
>> Debug> (use-value 12)
=> 12
(careful-symbol-value 'a)
>> Error: A is not bound.
>> To continue, type :CONTINUE followed by an option number.
>> 1: Specify a value to use this time.
>> 2: Specify a value to store and use in the future.
>> 3: Return to Lisp Toplevel.
>> Debug> (store-value 24)
=> 24
(careful-symbol-value 'a)
=> 24

;; Example of the USE-VALUE restart

(defun add-symbols-with-default (default &rest symbols)
  (handler-bind ((sys:unbound-symbol
                  #'(lambda (c)
                      (declare (ignore c))
                      (use-value default))))
    (apply #'+ (mapcar #'careful-symbol-value symbols))))
=> ADD-SYMBOLS-WITH-DEFAULT
(setq x 1 y 2) => 2
(add-symbols-with-default 3 'x 'y 'z) => 6
```

### Side Effects:

A transfer of control may occur if an appropriate restart is available, or (in the case of the function abort or the function muffle-warning) execution may be stopped.

### Affected By:

Each of these functions can be affected by the presence of a restart having the same name.

### Exceptional Situations:

If an appropriate abort restart is not available for the function abort, or an appropriate muffle-warning restart is not available for the function muffle-warning, an error of type control-error is signaled.

### See Also:

[invoke-restart](#), [Section 9.1.4.2 \(Restarts\)](#), [Section 9.1.4.2.2 \(Interfaces to Restarts\)](#), [assert](#), [ccase](#), [cerror](#), [check-type](#), [ctypecase](#), [use-value](#), [warn](#)

**Notes:**

```
(abort condition) == (invoke-restart 'abort)
(muffle-warning) == (invoke-restart 'muffle-warning)
(continue) == (let ((r (find-restart 'continue))) (if r (invoke-restart r)))
(use-value x) == (let ((r (find-restart 'use-value))) (if r (invoke-restart r x)))
(store-value x) == (let ((r (find-restart 'store-value))) (if r (invoke-restart r x)))
```

No functions defined in this specification are required to provide a [use-value restart](#).

**Function ABS****Syntax:**

**abs** *number* => *absolute-value*

**Arguments and Values:**

*number*—[a number](#).

*absolute-value*—[a non-negative real](#).

**Description:**

**abs** returns the absolute value of *number*.

If *number* is a [real](#), the result is of the same [type](#) as *number*.

If *number* is a [complex](#), the result is a positive [real](#) with the same magnitude as *number*. The result can be a [float](#) even if *number*'s components are [rationals](#) and an exact rational result would have been possible. Thus the result of (abs #c(3 4)) can be either 5 or 5.0, depending on the implementation.

**Examples:**

```
(abs 0) => 0
(abs 12/13) => 12/13
(abs -1.09) => 1.09
(abs #c(5.0 -5.0)) => 7.071068
(abs #c(5 5)) => 7.071068
(abs #c(3/5 4/5)) => 1 or approximately 1.0
(eq1 (abs -0.0) -0.0) => true
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:****Notes:**

## CLHS: Declaration DYNAMIC-EXTENT

If *number* is a complex, the result is equivalent to the following:

```
(sqrt (+ (expt (realpart number) 2) (expt (imagpart number) 2)))
```

An implementation should not use this formula directly for all complexes but should handle very large or very small components specially to avoid intermediate overflow or underflow.

## Function ACONS

**Syntax:**

**acons** *key datum alist => new-alist*

**Arguments and Values:**

*key*—an object.

*datum*—an object.

*alist*—an association list.

*new-alist*—an association list.

**Description:**

Creates a fresh cons, the cdr of which is *alist* and the car of which is another fresh cons, the car of which is *key* and the cdr of which is *datum*.

**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")
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[assoc](#), [pairlis](#)

**Notes:**

```
(acons key datum alist) == (cons (cons key datum) alist)
```

**Standard Generic Function ADD-METHOD****Syntax:**

**add-method** *generic-function method => generic-function*

**Method Signatures:**

**add-method** (*generic-function standard-generic-function*) (*method method*)

**Arguments and Values:**

*generic-function*—a *generic function object*.

*method*—a *method object*.

**Description:**

The generic function **add-method** adds a *method* to a *generic function*.

If *method* agrees with an existing *method* of *generic-function* on *parameter specializers* and *qualifiers*, the existing *method* is replaced.

**Examples:** None.

**Affected By:** None.

**Exceptional Situations:**

The *lambda list* of the method function of *method* must be congruent with the *lambda list* of *generic-function*, or an error of *type error* is signaled.

If *method* is a *method object* of another *generic function*, an error of *type error* is signaled.

**See Also:**

**defmethod**, **defgeneric**, **find-method**, **remove-method**, [Section 7.6.3 \(Agreement on Parameter Specializers and Qualifiers\)](#)

**Notes:** None.

**Function ADJOIN****Syntax:**

**adjoin** *item list &key key test test-not => new-list*

**Arguments and Values:**

*item*—an *object*.

*list*—a proper list.

*test*—a designator for a function of two arguments that returns a generalized boolean.

*test-not*—a designator for a function of two arguments that returns a generalized boolean.

*key*—a designator for a function of one argument, or nil.

*new-list*—a list.

### Description:

Tests whether *item* is the same as an existing element of *list*. If the *item* is not an existing element, adjoin adds it to *list* (as if by cons) and returns the resulting *list*; otherwise, nothing is added and the original *list* is returned.

The *test*, *test-not*, and *key* affect how it is determined whether *item* is the same as an element of *list*. For details, see [Section 17.2.1 \(Satisfying a Two-Argument Test\)](#).

### Examples:

```
(setq slist '()) => NIL
(adjoin 'a slist) => (A)
slist => NIL
(setq slist (adjoin '(test-item 1) slist)) => ((TEST-ITEM 1))
(adjoin '(test-item 1) slist) => ((TEST-ITEM 1) (TEST-ITEM 1))
(adjoin '(test-item 1) slist :test 'equal) => ((TEST-ITEM 1))
(adjoin '(new-test-item 1) slist :key #'cadr) => ((TEST-ITEM 1))
(adjoin '(new-test-item 1) slist) => ((NEW-TEST-ITEM 1) (TEST-ITEM 1))
```

**Affected By:** None.

### Exceptional Situations:

Should be prepared to signal an error of type type-error if *list* is not a proper list.

### See Also:

[pushnew](#), [Section 3.6 \(Traversal Rules and Side Effects\)](#)

### Notes:

The `:test-not` parameter is deprecated.

```
(adjoin item list :key fn)
== (if (member (fn item) list :key fn) list (cons item list))
```

## Function ADJUSTABLE-ARRAY-P

### Syntax:

**adjustable-array-p** *array* => *generalized-boolean*

**Arguments and Values:**

*array*—an array.

*generalized-boolean*—a generalized boolean.

**Description:**

Returns true if and only if adjust-array could return a value which is identical to *array* when given that *array* as its first argument.

**Examples:**

```
(adjustable-array-p
  (make-array 5
    :element-type 'character
    :adjustable t
    :fill-pointer 3)) => true
(adjustable-array-p (make-array 4)) => implementation-dependent
```

**Affected By:** None.

**Exceptional Situations:**

Should signal an error of type type-error if its argument is not an array.

**See Also:**

adjust-array, make-array

**Notes:** None.

**Function ADJUST-ARRAY****Syntax:**

**adjust-array** *array new-dimensions &key element-type initial-element initial-contents fill-pointer displaced-to displaced-index-offset*

=> *adjusted-array*

**Arguments and Values:**

*array*—an array.

*new-dimensions*—a valid array dimension or a list of valid array dimensions.

*element-type*—a type specifier.

*initial-element*—an object. *Initial-element* must not be supplied if either *initial-contents* or *displaced-to* is supplied.

## CLHS: Declaration DYNAMIC-EXTENT

*initial-contents*—an object. If array has rank greater than zero, then *initial-contents* is composed of nested sequences, the depth of which must equal the rank of array. Otherwise, array is zero-dimensional and *initial-contents* supplies the single element. *initial-contents* must not be supplied if either *initial-element* or *displaced-to* is given.

*fill-pointer*—a valid fill pointer for the array to be created, or t, or nil. The default is nil.

*displaced-to*—an array or nil. *initial-elements* and *initial-contents* must not be supplied if *displaced-to* is supplied.

*displaced-index-offset*—an object of type (fixnum 0 n) where n is (array-total-size displaced-to). *displaced-index-offset* may be supplied only if *displaced-to* is supplied.

*adjusted-array*—an array.

### Description:

**adjust-array** changes the dimensions or elements of array. The result is an array of the same type and rank as array, that is either the modified array, or a newly created array to which array can be displaced, and that has the given *new-dimensions*.

*New-dimensions* specify the size of each dimension of array.

*Element-type* specifies the type of the elements of the resulting array. If *element-type* is supplied, the consequences are unspecified if the upgraded array element type of *element-type* is not the same as the actual array element type of array.

If *initial-contents* is supplied, it is treated as for **make-array**. In this case none of the original contents of array appears in the resulting array.

If *fill-pointer* is an integer, it becomes the fill pointer for the resulting array. If *fill-pointer* is the symbol t, it indicates that the size of the resulting array should be used as the fill pointer. If *fill-pointer* is nil, it indicates that the fill pointer should be left as it is.

If *displaced-to non-nil*, a displaced array is created. The resulting array shares its contents with the array given by *displaced-to*. The resulting array cannot contain more elements than the array it is displaced to. If *displaced-to* is not supplied or nil, the resulting array is not a displaced array. If array A is created displaced to array B and subsequently array B is given to **adjust-array**, array A will still be displaced to array B. Although array might be a displaced array, the resulting array is not a displaced array unless *displaced-to* is supplied and not nil. The interaction between **adjust-array** and displaced arrays is as follows given three arrays, A, B, and C:

*A is not displaced before or after the call*  
(adjust-array A ...)

The dimensions of A are altered, and the contents rearranged as appropriate. Additional elements of A are taken from *initial-element*. The use of *initial-contents* causes all old contents to be discarded.

*A is not displaced before, but is displaced to C after the call*  
(adjust-array A ... :displaced-to C)

## CLHS: Declaration DYNAMIC-EXTENT

None of the original contents of A appears in A afterwards; A now contains the contents of C, without any rearrangement of C.

A is displaced to B before the call, and is displaced to C after the call

```
(adjust-array A ... :displaced-to B)
(adjust-array A ... :displaced-to C)
```

B and C might be the same. The contents of B do not appear in A afterward unless such contents also happen to be in C If *displaced-index-offset* is not supplied in the **adjust-array** call, it defaults to zero; the old offset into B is not retained.

A is displaced to B before the call, but not displaced afterward.

```
(adjust-array A ... :displaced-to B)
(adjust-array A ... :displaced-to nil)
```

A gets a new "data region," and contents of B are copied into it as appropriate to maintain the existing old contents; additional elements of A are taken from *initial-element* if supplied. However, the use of *initial-contents* causes all old contents to be discarded.

If *displaced-index-offset* is supplied, it specifies the offset of the resulting *array* from the beginning of the *array* that it is displaced to. If *displaced-index-offset* is not supplied, the offset is 0. The size of the resulting *array* plus the offset value cannot exceed the size of the *array* that it is displaced to.

If only *new-dimensions* and an *initial-element* argument are supplied, those elements of *array* that are still in bounds appear in the resulting *array*. The elements of the resulting *array* that are not in the bounds of *array* are initialized to *initial-element*; if *initial-element* is not provided, the consequences of later reading any such new *element* of *new-array* before it has been initialized are undefined.

If *initial-contents* or *displaced-to* is supplied, then none of the original contents of *array* appears in the new *array*.

The consequences are unspecified if *array* is adjusted to a size smaller than its *fill pointer* without supplying the *fill-pointer* argument so that its *fill-pointer* is properly adjusted in the process.

If A is displaced to B, the consequences are unspecified if B is adjusted in such a way that it no longer has enough elements to satisfy A.

If **adjust-array** is applied to an *array* that is *actually adjustable*, the *array* returned is *identical* to *array*. If the *array* returned by **adjust-array** is *distinct* from *array*, then the argument *array* is unchanged.

Note that if an *array* A is displaced to another *array* B, and B is displaced to another *array* C, and B is altered by **adjust-array**, A must now refer to the adjust contents of B. This means that an implementation cannot collapse the chain to make A refer to C directly and forget that the chain of reference passes through B. However, caching techniques are permitted as long as they preserve the semantics specified here.

### 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
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(setq beta (make-array '(2 3) :adjustable t))
=> #2A((NIL NIL NIL) (NIL NIL NIL))
  (adjust-array beta '(4 6) :displaced-to ada)
=> #2A((A B C NIL NIL NIL)
        (1 2 3 NIL NIL NIL)
        (NIL NIL NIL NIL NIL NIL)
        (NIL NIL NIL NIL NIL NIL))
(array-dimensions beta) => (4 6)
(aref beta 1 1) => 2
```

Suppose that the 4-by-4 array in `m` looks like this:

```
#2A(( alpha      beta      gamma      delta )
     ( epsilon    zeta      eta       theta )
     ( iota       kappa    lambda     mu      )
     ( nu         xi       omicron   pi      ))
```

Then the result of

```
(adjust-array m '(3 5) :initial-element 'baz)
```

is a 3-by-5 array with contents

```
#2A(( alpha      beta      gamma      delta      baz )
     ( epsilon    zeta      eta       theta      baz )
     ( iota       kappa    lambda     mu       baz ))
```

**Affected By:** None.

**Exceptional Situations:**

An error of type error is signaled if *fill-pointer* is supplied and non-nil but *array* has no fill pointer.

**See Also:**

adjustable-array-p, make-array, array-dimension-limit, array-total-size-limit, array

**Notes:** None.

## Standard Generic Function ALLOCATE-INSTANCE

**Syntax:**

```
allocate-instance class &rest initargs &key allow-other-keys => new-instance
```

**Method Signatures:**

```
allocate-instance (class standard-class) &rest initargs
```

```
allocate-instance (class structure-class) &rest initargs
```

**Arguments and Values:**

*class*—*a class*.

*initargs*—*a list of keyword/value pairs* (initialization argument *names* and *values*).

*new-instance*—*an object* whose *class* is *class*.

#### Description:

The generic function **allocate-instance** creates and returns a new instance of the *class*, without initializing it. When the *class* is a *standard class*, this means that the *slots* are *unbound*; when the *class* is a *structure class*, this means the *slots' values* are unspecified.

The caller of **allocate-instance** is expected to have already checked the initialization arguments.

The *generic function allocate-instance* is called by **make-instance**, as described in [Section 7.1 \(Object Creation and Initialization\)](#).

**Affected By:** None.

**Exceptional Situations:** None.

#### See Also:

[\*\*defclass\*\*](#), [\*\*make-instance\*\*](#), [\*\*class-of\*\*](#), [Section 7.1 \(Object Creation and Initialization\)](#)

#### Notes:

The consequences of adding *methods* to **allocate-instance** is unspecified. This capability might be added by the *Metaobject Protocol*.

## Function ALPHA-CHAR-P

#### Syntax:

**alpha-char-p** *character* => *generalized-boolean*

#### Arguments and Values:

*character*—*a character*.

*generalized-boolean*—*a generalized boolean*.

#### Description:

Returns *true* if *character* is an *alphabetic[1] character*; otherwise, returns *false*.

#### Examples:

```
(alpha-char-p #\a) => true
(alpha-char-p #\5) => false
(alpha-char-p #\Newline) => false
```

## CLHS: Declaration DYNAMIC-EXTENT

```
; ; This next example presupposes an implementation
; ; in which #\<ALPHA> is a defined character.
(alpha-char-p #\<ALPHA>) => implementation-dependent
```

### Affected By:

None. (In particular, the results of this predicate are independent of any special syntax which might have been enabled in the current readtable.)

### Exceptional Situations:

Should signal an error of type type-error if *character* is not a character.

### See Also:

[alphanumericp](#), Section 13.1.10 (Documentation of Implementation-Defined Scripts)

Notes: None.

## Function ALPHANUMERICP

### Syntax:

**alphanumericp** *character* => *generalized-boolean*

### Arguments and Values:

*character*—a character.

*generalized-boolean*—a generalized boolean.

### Description:

Returns true if *character* is an alphabetic[1] character or a numeric character; otherwise, returns false.

### Examples:

```
(alphanumericp #\Z) => true
(alphanumericp #\9) => true
(alphanumericp #\Newline) => false
(alphanumericp #\#) => false
```

### Affected By:

None. (In particular, the results of this predicate are independent of any special syntax which might have been enabled in the current readtable.)

### Exceptional Situations:

Should signal an error of type type-error if *character* is not a character.

### See Also:

Function ALPHANUMERICP

**alpha-char-p, graphic-char-p, digit-char-p****Notes:**

Alphanumeric characters are graphic as defined by **graphic-char-p**. The alphanumeric characters are a subset of the graphic characters. The standard characters A through Z, a through z, and 0 through 9 are alphanumeric characters.

```
(alphanumericp x)
== (or (alpha-char-p x) (not (null (digit-char-p x))))
```

**Function APPEND****Syntax:**

**append** &*rest lists* => *result*

**Arguments and Values:**

*list*—each must be a *proper list* except the last, which may be any *object*.

*result*—an *object*. This will be a *list* unless the last *list* was not a *list* and all preceding *lists* were *null*.

**Description:**

**append** returns a new *list* that is the concatenation of the copies. *lists* are left unchanged; the *list structure* of each of *lists* except the last is copied. The last argument is not copied; it becomes the *cdr* of the final *dotted pair* of the concatenation of the preceding *lists*, or is returned directly if there are no preceding *non-empty lists*.

**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
(append 'a) => A
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**nconc, concatenate**

**Notes:** None.

## Function APPLY

### Syntax:

**apply** *function* &*rest args+* => *result\**

### Arguments and Values:

*function*—a function designator.

*args*—a spreadable argument list designator.

*results*—the values returned by *function*.

### Description:

Applies the *function* to the *args*.

When the *function* receives its arguments via &*rest*, it is permissible (but not required) for the implementation to bind the rest parameter to an object that shares structure with the last argument to **apply**. Because a function can neither detect whether it was called via **apply** nor whether (if so) the last argument to **apply** was a constant, conforming programs must neither rely on the list structure of a rest list to be freshly consed, nor modify that list structure.

**setf** can be used with **apply** in certain circumstances; see [Section 5.1.2.5 \(APPLY Forms as Places\)](#).

### Examples:

```
(setq f '+) => +
(apply f '(1 2)) => 3
(setq f #'-) => #<FUNCTION ->
(apply f '(1 2)) => -1
(apply #'max 3 5 '(2 7 3)) => 7
(apply 'cons '((+ 2 3) 4)) => ((+ 2 3) . 4)
(apply #'+'()) => 0

(defparameter *some-list* '(a b c))
(defun strange-test (&rest x) (eq x *some-list*))
(apply #'strange-test *some-list*) => implementation-dependent

(defun bad-boy (&rest x) (rplacd x 'y))
(bad-boy 'a 'b 'c) has undefined consequences.
(apply #'bad-boy *some-list*) has undefined consequences.

(defun foo (size &rest keys &key double &allow-other-keys)
  (let ((v (apply #'make-array size :allow-other-keys t keys)))
    (if double (concatenate (type-of v) v v) v)))
(foo 4 :initial-contents '(a b c d) :double t)
=> #(A B C D A B C D)
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[\*\*funcall\*\*](#), [\*\*fdefinition\*\*](#), [\*\*function\*\*](#), [Section 3.1 \(Evaluation\)](#), [Section 5.1.2.5 \(APPLY Forms as Places\)](#)

**Notes:** None.

## **Function APROPOS, APROPOS-LIST**

**Syntax:**

**apropos** *string &optional package => <no values>*

**apropos-list** *string &optional package => symbols*

**Arguments and Values:**

*string*—a [\*string designator\*](#).

*package*—a [\*package designator\*](#) or [\*\*nil\*\*](#). The default is [\*\*nil\*\*](#).

*symbols*—a [\*list of symbols\*](#).

**Description:**

These functions search for [\*interned symbols\*](#) whose [\*names\*](#) contain the substring *string*.

For [\*\*apropos\*\*](#), as each such [\*symbol\*](#) is found, its name is printed on [\*standard output\*](#). In addition, if such a [\*symbol\*](#) is defined as a [\*function\*](#) or [\*dynamic variable\*](#), information about those definitions might also be printed.

For [\*\*apropos-list\*\*](#), no output occurs as the search proceeds; instead a list of the matching [\*symbols\*](#) is returned when the search is complete.

If *package* is [\*non-nil\*](#), only the [\*symbols accessible\*](#) in that *package* are searched; otherwise all [\*symbols accessible\*](#) in any [\*package\*](#) are searched.

Because a [\*symbol\*](#) might be available by way of more than one inheritance path, [\*\*apropos\*\*](#) might print information about the [\*same symbol\*](#) more than once, or [\*\*apropos-list\*\*](#) might return a [\*list\*](#) containing duplicate [\*symbols\*](#).

Whether or not the search is case-sensitive is [\*implementation-defined\*](#).

**Examples:** None.

**Affected By:**

The set of [\*symbols\*](#) which are currently [\*interned\*](#) in any [\*packages\*](#) being searched.

[\*\*apropos\*\*](#) is also affected by [\*\*\\*standard-output\\*\*\*](#).

**Exceptional Situations:** None.

**See Also:** None.

**Notes:** None.

## ***Function ARRAY-DIMENSIONS***

**Syntax:**

**array-dimensions** *array* => *dimensions*

**Arguments and Values:**

*array*—an array.

*dimensions*—a list of integers.

**Description:**

Returns a list of the dimensions of *array*. (If *array* is a vector with a fill pointer, that fill pointer is ignored.)

**Examples:**

```
(array-dimensions (make-array 4)) => (4)
(array-dimensions (make-array '(2 3))) => (2 3)
(array-dimensions (make-array 4 :fill-pointer 2)) => (4)
```

**Affected By:** None.

**Exceptional Situations:**

Should signal an error of type type-error if its argument is not an array.

**See Also:**

**array-dimension**

**Notes:** None.

## ***Function ARRAY-DIMENSION***

**Syntax:**

**array-dimension** *array* *axis-number* => *dimension*

**Arguments and Values:**

*array*—an array.

*axis-number*—an integer greater than or equal to zero and less than the rank of the *array*.

*dimension*—a non-negative integer.

**Description:**

**array-dimension** returns the *axis-number dimension*[1] of *array*. (Any *fill pointer* is ignored.)

**Examples:**

```
(array-dimension (make-array 4) 0) => 4
(array-dimension (make-array '(2 3)) 1) => 3
```

**Affected By:**

None.

**Exceptional Situations:** None.

**See Also:**

**array-dimensions, length**

**Notes:**

```
(array-dimension array n) == (nth n (array-dimensions array))
```

## Function ARRAY-DISPLACEMENT

**Syntax:**

**array-displacement** *array* => *displaced-to*, *displaced-index-offset*

**Arguments and Values:**

*array*—an array.

*displaced-to*—an *array* or nil.

*displaced-index-offset*—a non-negative fixnum.

**Description:**

If the *array* is a displaced array, returns the values of the :displaced-to and :displaced-index-offset options for the *array* (see the functions make-array and adjust-array). If the *array* is not a displaced array, nil and 0 are returned.

If **array-displacement** is called on an *array* for which a non-nil object was provided as the :displaced-to argument to make-array or adjust-array, it must return that object as its first value. It is implementation-dependent whether **array-displacement** returns a non-nil primary value for any other *array*.

**Examples:**

```
(setq a1 (make-array 5)) => #<ARRAY 5 simple 46115576>
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(setq a2 (make-array 4 :displaced-to a1
                      :displaced-index-offset 1))
=> #<ARRAY 4 indirect 46117134>
(array-displacement a2)
=> #<ARRAY 5 simple 46115576>, 1
(setq a3 (make-array 2 :displaced-to a2
                      :displaced-index-offset 2))
=> #<ARRAY 2 indirect 46122527>
(array-displacement a3)
=> #<ARRAY 4 indirect 46117134>, 2
```

**Affected By:** None.

**Exceptional Situations:**

Should signal an error of type type-error if *array* is not an array.

**See Also:**

[make-array](#)

**Notes:** None.

## Function ARRAY-ELEMENT-TYPE

**Syntax:**

**array-element-type** *array* => *typespec*

**Arguments and Values:**

*array*—an array.

*typespec*—a type specifier.

**Description:**

Returns a type specifier which represents the actual array element type of the array, which is the set of objects that such an *array* can hold. (Because of array upgrading, this type specifier can in some cases denote a supertype of the expressed array element type of the *array*.)

**Examples:**

```
(array-element-type (make-array 4)) => T
(array-element-type (make-array 12 :element-type '(unsigned-byte 8)))
=> implementation-dependent
(array-element-type (make-array 12 :element-type '(unsigned-byte 5)))
=> implementation-dependent

(array-element-type (make-array 5 :element-type '(mod 5)))
```

could be (mod 5), (mod 8), fixnum, t, or any other type of which (mod 5) is a subtype.

**Affected By:**

The *implementation*.

**Exceptional Situations:**

Should signal an error of *type type-error* if its argument is not an *array*.

**See Also:**

[array](#), [make-array](#), [subtypep](#), [upgraded-array-element-type](#)

**Notes:** None.

**Function ARRAY-HAS-FILL-POINTER-P****Syntax:**

**array-has-fill-pointer-p** *array* => *generalized-boolean*

**Arguments and Values:**

*array*—an *array*.

*generalized-boolean*—a *generalized boolean*.

**Description:**

Returns *true* if *array* has a *fill pointer*; otherwise returns *false*.

**Examples:**

```
(array-has-fill-pointer-p (make-array 4)) => implementation-dependent
(array-has-fill-pointer-p (make-array '(2 3))) => false
(array-has-fill-pointer-p
  (make-array 8
    :fill-pointer 2
    :initial-element 'filler)) => true
```

**Affected By:** None.**Exceptional Situations:**

Should signal an error of *type type-error* if its argument is not an *array*.

**See Also:**

[make-array](#), [fill-pointer](#)

**Notes:**

## CLHS: Declaration DYNAMIC-EXTENT

Since arrays of rank other than one cannot have a fill pointer, **array-has-fill-pointer-p** always returns nil when its argument is such an array.

### Function ARRAY-IN-BOUNDS-P

#### Syntax:

**array-in-bounds-p** *array &rest subscripts => generalized-boolean*

#### Arguments and Values:

*array*—an array.

*subscripts*—a list of integers of length equal to the rank of the array.

*generalized-boolean*—a generalized boolean.

#### Description:

Returns true if the *subscripts* are all in bounds for *array*; otherwise returns false. (If *array* is a vector with a fill pointer, that fill pointer is ignored.)

#### Examples:

```
(setq a (make-array '(7 11) :element-type 'string-char))
(array-in-bounds-p a 0 0) => true
(array-in-bounds-p a 6 10) => true
(array-in-bounds-p a 0 -1) => false
(array-in-bounds-p a 0 11) => false
(array-in-bounds-p a 7 0) => false
```

**Affected By:** None.

**Exceptional Situations:** None.

#### See Also:

#### array-dimensions

#### Notes:

```
(array-in-bounds-p array subscripts)
== (and (not (some #'minusp (list subscripts)))
        (every #'< (list subscripts) (array-dimensions array)))
```

### Function ARRAY-RANK

#### Syntax:

**array-rank** *array => rank*

#### Arguments and Values:

*array*---an array.

*rank*---a non-negative integer.

### Description:

Returns the number of dimensions of *array*.

### Examples:

```
(array-rank (make-array '()) ) => 0
(array-rank (make-array 4)) => 1
(array-rank (make-array '(4))) => 1
(array-rank (make-array '(2 3))) => 2
```

**Affected By:** None.

### Exceptional Situations:

Should signal an error of type type-error if its argument is not an array.

### See Also:

array-rank-limit, make-array

**Notes:** None.

## Function ARRAY-ROW-MAJOR-INDEX

### Syntax:

**array-row-major-index** *array &rest subscripts => index*

### Arguments and Values:

*array*---an array.

*subscripts*---a list of *valid array indices* for the *array*.

*index*---a valid array row-major index for the *array*.

### Description:

Computes the position according to the row-major ordering of *array* for the element that is specified by *subscripts*, and returns the offset of the element in the computed position from the beginning of *array*.

For a one-dimensional *array*, the result of **array-row-major-index** equals *subscript*.

**array-row-major-index** ignores fill pointers.

### Examples:

## CLHS: Declaration DYNAMIC-EXTENT

```
(setq a (make-array '(4 7) :element-type '(unsigned-byte 8)))
(array-row-major-index a 1 2) => 9
(array-row-major-index
  (make-array '(2 3 4)
    :element-type '(unsigned-byte 8)
    :displaced-to a
    :displaced-index-offset 4)
  0 2 1) => 9
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:** None.

**Notes:**

A possible definition of **array-row-major-index**, with no error-checking, is

```
(defun array-row-major-index (a &rest subscripts)
  (apply #'+ (maplist #'(lambda (x y)
    (* (car x) (apply #'* (cdr y))))
    subscripts
    (array-dimensions a))))
```

## Function ARRAY-TOTAL-SIZE

**Syntax:**

**array-total-size** *array* => *size*

**Arguments and Values:**

*array*—an array.

*size*—a non-negative integer.

**Description:**

Returns the array total size of the *array*.

**Examples:**

```
(array-total-size (make-array 4)) => 4
(array-total-size (make-array 4 :fill-pointer 2)) => 4
(array-total-size (make-array 0)) => 0
(array-total-size (make-array '(4 2))) => 8
(array-total-size (make-array '(4 0))) => 0
(array-total-size (make-array '())) => 1
```

**Affected By:** None.

**Exceptional Situations:**

## CLHS: Declaration DYNAMIC-EXTENT

Should signal an error of type type-error if its argument is not an array.

See Also:

make-array, array-dimensions

Notes:

If the array is a vector with a fill pointer, the fill pointer is ignored when calculating the array total size.

Since the product of no arguments is one, the array total size of a zero-dimensional array is one.

```
(array-total-size x)
  == (apply #'* (array-dimensions x))
  == (reduce #'* (array-dimensions x))
```

## Accessor AREF

Syntax:

**aref** *array* &*rest subscripts* => *element*

```
(setf (aref array &rest subscripts) new-element)
```

Arguments and Values:

*array*---an array.

*subscripts*---a list of valid array indices for the *array*.

*element, new-element*---an object.

Description:

Accesses the array element specified by the subscripts. If no subscripts are supplied and array is zero rank, aref accesses the sole element of array.

aref ignores fill pointers. It is permissible to use aref to access any array element, whether active or not.

Examples:

If the variable *foo* names a 3-by-5 array, then the first index could be 0, 1, or 2, and the second index could be 0, 1, 2, 3, or 4. The array elements can be referred to by using the function aref; for example, (**aref** *foo* 2 1) refers to element (2, 1) of the array.

```
(aref (setq alpha (make-array 4)) 3) => implementation-dependent
(setf (aref alpha 3) 'sirens) => SIRENS
(aref alpha 3) => SIRENS
(aref (setq beta (make-array '(2 4)
                           :element-type '(unsigned-byte 2)
                           :initial-contents '((0 1 2 3) (3 2 1 0))))
      1 2) => 1
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(setq gamma '(0 2))
(apply #'aref beta gamma) => 2
(setf (apply #'aref beta gamma) 3) => 3
(apply #'aref beta gamma) => 3
(aref beta 0 2) => 3
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[bit](#), [char](#), [elt](#), [row-major-aref](#), [svref](#), Section 3.2.1 (Compiler Terminology)

**Notes:** None.

## ***Function ARITHMETIC-ERROR-OPERANDS, ARITHMETIC-ERROR-OPERATION***

**Syntax:**

**arithmetic-error-operands** *condition => operands*

**arithmetic-error-operation** *condition => operation*

**Arguments and Values:**

*condition*—a condition of type **arithmetic-error**.

*operands*—a list.

*operation*—a function designator.

**Description:**

**arithmetic-error-operands** returns a list of the operands which were used in the offending call to the operation that signaled the *condition*.

**arithmetic-error-operation** returns a list of the offending operation in the offending call that signaled the *condition*.

**Examples:** None.

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[arithmetic-error](#), Section 9 (Conditions)**Notes:****Function ARRAYP****Syntax:****arrayp** *object* => *generalized-boolean***Arguments and Values:***object*---an [object](#).*generalized-boolean*---a [generalized boolean](#).**Description:**Returns [true](#) if *object* is of [type array](#); otherwise, returns [false](#).**Examples:**

```
(arrayp (make-array '(2 3 4) :adjustable t)) => true
(arrayp (make-array 6)) => true
(arrayp #*1011) => true
(arrayp "hi") => true
(arrayp 'hi) => false
(arrayp 12) => false
```

**Affected By:** None.**Exceptional Situations:** None.**See Also:**[typep](#)**Notes:**(arrayp *object*) == (typep *object* 'array)**Function ASH****Syntax:****ash** *integer count* => *shifted-integer***Arguments and Values:***integer*---an [integer](#).

*count*---an integer.

*shifted-integer*—an integer.

### Description:

**ash** performs the arithmetic shift operation on the binary representation of *integer*, which is treated as if it were binary.

**ash** shifts *integer* arithmetically left by *count* bit positions if *count* is positive, or right *count* bit positions if *count* is negative. The shifted value of the same sign as *integer* is returned.

Mathematically speaking, [ash](#) performs the computation  $\text{floor}(\text{integer} \times 2^{\text{count}})$ . Logically, [ash](#) moves all of the bits in *integer* to the left, adding zero-bits at the right, or moves them to the right, discarding bits.

**ash** is defined to behave as if *integer* were represented in two's complement form, regardless of how *integers* are represented internally.

## Examples:

**Affected By:** None.

## **Exceptional Situations:**

Should signal an error of **type type–error** if *integer* is not an *integer*. Should signal an error of **type type–error** if *count* is not an *integer*. Might signal **arithmetic–error**.

**See Also:** None.

## Notes:

```
(logbitp j (ash n k))
== (and (>= j k) (logbitp (- j k) n))
```

### ***Function ASIN, ACOS, ATAN***

### Syntax:

**asin** *number* => *radians*

**acos** *number* => *radians*

**atan** *number1* &*optional number2* => radians

### **Arguments and Values:**

## CLHS: Declaration DYNAMIC-EXTENT

*number*—*a number*.

*number1*—*a number* if *number2* is not supplied, or a *real* if *number2* is supplied.

*number2*—*a real*.

*radians*—*a number* (of radians).

### Description:

**asin**, **acos**, and **atan** compute the arc sine, arc cosine, and arc tangent respectively.

The arc sine, arc cosine, and arc tangent (with only *number1* supplied) functions can be defined mathematically for *number* or *number1* specified as *x* as in the next figure.

Function	Definition
Arc sine	$-i \log(ix + \sqrt{1-x^2})$
Arc cosine	$(\pi/2) - \arcsin x$
Arc tangent	$-i \log((1+ix)\sqrt{1/(1+x^2)})$

### Figure 12–14. Mathematical definition of arc sine, arc cosine, and arc tangent

These formulae are mathematically correct, assuming completely accurate computation. They are not necessarily the simplest ones for real-valued computations.

If both *number1* and *number2* are supplied for **atan**, the result is the arc tangent of *number1*/*number2*. The value of **atan** is always between  $-\pi/2$  (exclusive) and  $\pi/2$  (inclusive) when minus zero is not supported. The range of the two-argument arc tangent when minus zero is supported includes  $-\pi/2$ .

For a *real number1*, the result is a *real* and lies between  $-\pi/2$  and  $\pi/2$  (both exclusive). *number1* can be a *complex* if *number2* is not supplied. If both are supplied, *number2* can be zero provided *number1* is not zero.

The following definition for arc sine determines the range and branch cuts:

$$\arcsin z = -i \log(iz + \sqrt{1-z^2})$$

The branch cut for the arc sine function is in two pieces: one along the negative real axis to the left of  $-1$  (inclusive), continuous with quadrant II, and one along the positive real axis to the right of  $1$  (inclusive), continuous with quadrant IV. The range is that strip of the complex plane containing numbers whose real part is between  $-\pi/2$  and  $\pi/2$ . A number with real part equal to  $-\pi/2$  is in the range if and only if its imaginary part is non-negative; a number with real part equal to  $\pi/2$  is in the range if and only if its imaginary part is non-positive.

The following definition for arc cosine determines the range and branch cuts:

$$\arccos z = \pi/2 - \arcsin z$$

or, which are equivalent,

$$\arccos z = -i \log(z + i \sqrt{1-z^2})$$

## CLHS: Declaration DYNAMIC-EXTENT

$$\arccos z = 2 \log((\sqrt{(1+z)/2}) + i \sqrt{((1-z)/2)})/i$$

The branch cut for the arc cosine function is in two pieces: one along the negative real axis to the left of  $-1$  (inclusive), continuous with quadrant II, and one along the positive real axis to the right of  $1$  (inclusive), continuous with quadrant IV. This is the same branch cut as for arc sine. The range is that strip of the complex plane containing numbers whose real part is between  $0$  and  $\pi$ . A number with real part equal to  $0$  is in the range if and only if its imaginary part is non-negative; a number with real part equal to  $\pi$  is in the range if and only if its imaginary part is non-positive.

The following definition for (one-argument) arc tangent determines the range and branch cuts:

$$\arctan z = \log(1+iz) - \log(1-iz)/(2i)$$

Beware of simplifying this formula; "obvious" simplifications are likely to alter the branch cuts or the values on the branch cuts incorrectly. The branch cut for the arc tangent function is in two pieces: one along the positive imaginary axis above  $i$  (exclusive), continuous with quadrant II, and one along the negative imaginary axis below  $-i$  (exclusive), continuous with quadrant IV. The points  $i$  and  $-i$  are excluded from the domain. The range is that strip of the complex plane containing numbers whose real part is between  $-\pi/2$  and  $\pi/2$ . A number with real part equal to  $-\pi/2$  is in the range if and only if its imaginary part is strictly positive; a number with real part equal to  $\pi/2$  is in the range if and only if its imaginary part is strictly negative. Thus the range of arc tangent is identical to that of arc sine with the points  $-\pi/2$  and  $\pi/2$  excluded.

For [atan](#), the signs of *number1* (indicated as  $x$ ) and *number2* (indicated as  $y$ ) are used to derive quadrant information. The next figure details various special cases. The asterisk (\*) indicates that the entry in the figure applies to implementations that support minus zero.

$y$	Condition	$x$	Condition	Cartesian locus	Range of result
$y = 0$		$x > 0$		Positive x-axis	$0$
* $y = +0$		$x > 0$		Positive x-axis	$+0$
* $y = -0$		$x > 0$		Positive x-axis	$-0$
$y > 0$		$x > 0$		Quadrant I	$0 < \text{result} < \pi/2$
$y > 0$		$x = 0$		Positive y-axis	$\pi/2$
$y > 0$		$x < 0$		Quadrant II	$\pi/2 < \text{result} < \pi$
$y = 0$		$x < 0$		Negative x-axis	$\pi$
* $y = +0$		$x < 0$		Negative x-axis	$+0$
* $y = -0$		$x < 0$		Negative x-axis	$-0$
$y < 0$		$x < 0$		Quadrant III	$-\pi < \text{result} < -\pi/2$
$y < 0$		$x = 0$		Negative y-axis	$-\pi/2$
$y < 0$		$x > 0$		Quadrant IV	$-\pi/2 < \text{result} < 0$
$y = 0$		$x = 0$		Origin	undefined consequences
* $y = +0$		$x = +0$		Origin	$+0$
* $y = -0$		$x = +0$		Origin	$-0$
* $y = +0$		$x = -0$		Origin	$+0$
* $y = -0$		$x = -0$		Origin	$-0$

**Figure 12–15. Quadrant information for arc tangent**

### Examples:

```
(asin 0) => 0.0
acos #c(0 1) => #C(1.5707963267948966 -0.8813735870195432)
(/ (atan 1 (sqrt 3)) 6) => 0.087266
(atan #c(0 2)) => #C(-1.5707964 0.54930615)
```

**Affected By:** None.

**Exceptional Situations:**

acos and asin should signal an error of type type-error if *number* is not a number. atan should signal type-error if one argument is supplied and that argument is not a number, or if two arguments are supplied and both of those arguments are not *reals*.

acos, asin, and atan might signal arithmetic-error.

**See Also:**

log, sqrt, Section 12.1.3.3 (Rule of Float Substitutability)

**Notes:**

The result of either asin or acos can be a complex even if *number* is not a complex; this occurs when the absolute value of *number* is greater than one.

## Function ASSOC, ASSOC-IF, ASSOC-IF-NOT

**Syntax:**

**assoc** *item alist &key key test test-not => entry*

**assoc-if** *predicate alist &key key => entry*

**assoc-if-not** *predicate alist &key key => entry*

**Arguments and Values:**

*item*—an object.

*alist*—an association list.

*predicate*—a designator for a function of one argument that returns a generalized boolean.

*test*—a designator for a function of two arguments that returns a generalized boolean.

*test-not*—a designator for a function of two arguments that returns a generalized boolean.

*key*—a designator for a function of one argument, or nil.

*entry*—a cons that is an element of *alist*, or nil.

**Description:**

assoc, assoc-if, and assoc-if-not return the first cons in *alist* whose car satisfies the *test*, or nil if no such cons is found.

## CLHS: Declaration DYNAMIC-EXTENT

For assoc, assoc-if, and assoc-if-not, if nil appears in *alist* in place of a pair, it is ignored.

### Examples:

```
(setq values '((x . 100) (y . 200) (z . 50))) => ((X . 100) (Y . 200) (Z . 50))
(assoc 'y values) => (Y . 200)
(rplacd (assoc 'y values) 201) => (Y . 201)
(assoc 'y values) => (Y . 201)
(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)
(assoc 'r '((a . b) (c . d) (r . x) (s . y) (r . z))) => (R . X)
(assoc 'goo '((foo . bar) (zoo . goo))) => NIL
(assoc '2 '((1 a b c) (2 b c d) (-7 x y z))) => (2 B C D)
(setq alist '("one" . 1) ("2" . 2) ("three" . 3))
=> ("one" . 1) ("2" . 2) ("three" . 3))
(assoc-if-not #'alpha-char-p alist
              :key #'(lambda (x) (char x 0))) => ("2" . 2)
```

**Affected By:** None.

### Exceptional Situations:

Should be prepared to signal an error of type type-error if *alist* is not an association list.

### See Also:

rassoc, find, member, position, Section 3.6 (Traversal Rules and Side Effects)

### Notes:

The `:test-not` parameter is deprecated.

The *function assoc-if-not* is deprecated.

It is possible to rplacd the result of assoc, provided that it is not nil, in order to "update" *alist*.

The two expressions

```
(assoc item list :test fn)
```

and

```
(find item list :test fn :key #'car)
```

are equivalent in meaning with one exception: if nil appears in *alist* in place of a pair, and *item* is nil, find will compute the car of the nil in *alist*, find that it is equal to *item*, and return nil, whereas assoc will ignore the nil in *alist* and continue to search for an actual cons whose car is nil.

**Function ATOM****Syntax:**

**atom** *object* => *generalized-boolean*

**Arguments and Values:**

*object*---an object.

*generalized-boolean*---a generalized boolean.

**Description:**

Returns true if *object* is of type atom; otherwise, returns false.

**Examples:**

```
(atom 'sss) => true
(atom (cons 1 2)) => false
(atom nil) => true
(atom '()) => true
(atom 3) => true
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:** None.

**Notes:**

```
(atom object) == (typep object 'atom) == (not (consp object))
== (not (typep object 'cons)) == (typep object '(not cons))
```

**Function BOOLE****Syntax:**

**boole** *op integer-1 integer-2* => *result-integer*

**Arguments and Values:**

*Op*---a bit-wise logical operation specifier.

*integer-1*---an integer.

*integer-2*---an integer.

*result-integer*---an integer.

**Description:**

**boole** performs bit-wise logical operations on *integer-1* and *integer-2*, which are treated as if they were binary and in two's complement representation.

The operation to be performed and the return value are determined by *op*.

**boole** returns the values specified for any *op* in the next figure.

Op	Result
<u>boole-1</u>	integer-1
<u>boole-2</u>	integer-2
<u>boole-andc1</u>	and complement of integer-1 with integer-2
<u>boole-andc2</u>	and integer-1 with complement of integer-2
<u>boole-and</u>	and
<u>boole-c1</u>	complement of integer-1
<u>boole-c2</u>	complement of integer-2
<u>boole-clr</u>	always 0 (all zero bits)
<u>boole-eqv</u>	equivalence (exclusive nor)
<u>boole-ior</u>	inclusive or
<u>boole-nand</u>	not-and
<u>boole-nor</u>	not-or
<u>boole-orc1</u>	or complement of integer-1 with integer-2
<u>boole-orc2</u>	or integer-1 with complement of integer-2
<u>boole-set</u>	always -1 (all one bits)
<u>boole-xor</u>	exclusive or

**Figure 12–17. Bit-Wise Logical Operations**

**Examples:**

```
(boole boole-ior 1 16) => 17
(boole boole-and -2 5) => 4
(boole boole-eqv 17 15) => -31

;; These examples illustrate the result of applying BOOLE and each
;; of the possible values of OP to each possible combination of bits.
(progn
  (format t "~&Results of (BOOLE <op> #b0011 #b0101) ...~
            ~%---Op-----Decimal----Binary----Bits---~%")
  (dolist (symbol '(boole-1      boole-2      boole-and      boole-andc1
                  boole-andc2  boole-c1    boole-c2    boole-clr
                  boole-eqv   boole-ior   boole-nand  boole-nor
                  boole-orc1  boole-orc2  boole-set   boole-xor))
    (let ((result (boole (symbol-value symbol) #b0011 #b0101)))
      (format t "~& ~A~13T~3,' D~23T~*:~5,' B~31T ...~4,'0B~%"'
              symbol result (logand result #b1111)))))

>> Results of (BOOLE <op> #b0011 #b0101) ...
>> ---Op-----Decimal----Binary----Bits---
>>   BOOLE-1      3          11      ...0011
>>   BOOLE-2      5          101     ...0101
>>   BOOLE-AND    1          1       ...0001
>>   BOOLE-ANDC1  4          100     ...0100
>>   BOOLE-ANDC2  2          10      ...0010
>>   BOOLE-C1    -4         -100    ...1100
>>   BOOLE-C2    -6         -110    ...1010
>>   BOOLE-CLR   0          0       ...0000
>>   BOOLE-EQV   -7         -111    ...1001
```

## CLHS: Declaration DYNAMIC-EXTENT

```
>> BOOLE-IOR      7      111      ...0111
>> BOOLE-NAND    -2      -10      ...1110
>> BOOLE-NOR     -8      -1000     ...1000
>> BOOLE-ORC1    -3      -11      ...1101
>> BOOLE-ORC2    -5      -101      ...1011
>> BOOLE-SET     -1      -1      ...1111
>> BOOLE-XOR     6      110      ...0110
=> NIL
```

**Affected By:** None.

### Exceptional Situations:

Should signal **type-error** if its first argument is not a *bit-wise logical operation specifier* or if any subsequent argument is not an **integer**.

### See Also:

### **logand**

### Notes:

In general,

```
(boole boole-and x y) == (logand x y)
```

*Programmers* who would prefer to use numeric indices rather than *bit-wise logical operation specifiers* can get an equivalent effect by a technique such as the following:

```
; ; The order of the values in this 'table' are such that
; ; (logand (boole (elt boole-n-vector n) #b0101 #b0011) #b1111) => n
(defconstant boole-n-vector
  (vector boole-clr  boole-and  boole-andc1 boole-2
          boole-andc2 boole-1   boole-xor   boole-ior
          boole-nor   boole-eqv  boole-c1   boole-orc1
          boole-c2   boole-orc2 boole-nand boole-set))
=> BOOLE-N-VECTOR
(proclaim '(inline boole-n))
=> implementation-dependent
(defun boole-n (n integer &rest more-integers)
  (apply #'boole (elt boole-n-vector n) integer more-integers))
=> BOOLE-N
(boole-n #b0111 5 3) => 7
(boole-n #b0001 5 3) => 1
(boole-n #b1101 5 3) => -3
(loop for n from #b0000 to #b1111 collect (boole-n n 5 3))
=> (0 1 2 3 4 5 6 7 -8 -7 -6 -5 -4 -3 -2 -1)
```

## Function BOUNDP

### Syntax:

**boundp** *symbol* => *generalized-boolean*

### Arguments and Values:

*symbol*---a symbol.

*generalized-boolean*---a generalized boolean.

#### Description:

Returns true if *symbol* is bound; otherwise, returns false.

#### Examples:

```
(setq x 1) => 1
(boundp 'x) => true
(makunbound 'x) => x
(boundp 'x) => false
(let ((x 2)) (boundp 'x)) => false
(let ((x 2)) (declare (special x)) (boundp 'x)) => true
```

**Affected By:** None.

#### Exceptional Situations:

Should signal an error of type type-error if *symbol* is not a symbol.

#### See Also:

set, setq, symbol-value, makunbound

#### Notes:

The function bound determines only whether a symbol has a value in the global environment; any lexical bindings are ignored.

## Function BREAK

#### Syntax:

**break** &*optional format-control* &*rest format-arguments* => nil

#### Arguments and Values:

*format-control*---a format control. The default is implementation-dependent.

*format-arguments*---format arguments for the *format-control*.

#### Description:

**break** formats *format-control* and *format-arguments* and then goes directly into the debugger without allowing any possibility of interception by programmed error-handling facilities.

If the continue restart is used while in the debugger, **break** immediately returns nil without taking any unusual recovery action.

## CLHS: Declaration DYNAMIC-EXTENT

**break** binds **\*debugger-hook\*** to **nil** before attempting to enter the debugger.

### Examples:

```
(break "You got here with arguments: ~:S." '(FOO 37 A))
>> BREAK: You got here with these arguments: FOO, 37, A.
>> To continue, type :CONTINUE followed by an option number:
>> 1: Return from BREAK.
>> 2: Top level.
>> Debug> :CONTINUE 1
>> Return from BREAK.
=> NIL
```

### Side Effects:

The debugger is entered.

### Affected By:

**\*debug-io\*.**

**Exceptional Situations:** None.

### See Also:

[\*\*error\*\*](#), [\*\*invoke-debugger\*\*](#).

### Notes:

**break** is used as a way of inserting temporary debugging "breakpoints" in a program, not as a way of signaling errors. For this reason, **break** does not take the *continue-format-control argument* that **cerror** takes. This and the lack of any possibility of interception by *condition handling* are the only program-visible differences between **break** and **cerror**.

The user interface aspects of **break** and **cerror** are permitted to vary more widely, in order to accomodate the interface needs of the *implementation*. For example, it is permissible for a *Lisp read-eval-print loop* to be entered by **break** rather than the conventional debugger.

**break** could be defined by:

```
(defun break (&optional (format-control "Break") &rest format-arguments)
  (with-simple-restart (continue "Return from BREAK. ")
    (let ((*debugger-hook* nil))
      (invoke-debugger
        (make-condition 'simple-condition
                      :format-control format-control
                      :format-arguments format-arguments))))
  nil)
```

## Function BROADCAST-STREAM-STREAMS

### Syntax:

Function BROADCAST-STREAM-STREAMS

## CLHS: Declaration DYNAMIC-EXTENT

**broadcast-stream-streams** *broadcast-stream => streams*

### Arguments and Values:

*broadcast-stream*—a broadcast stream.

*streams*—a list of streams.

### Description:

Returns a list of output streams that constitute all the streams to which the *broadcast-stream* is broadcasting.

**Examples:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:** None.

**Notes:** None.

## Function BIT-AND, BIT-ANDC1, BIT-ANDC2, BIT-EQV, BIT-IOR, BIT-NAND, BIT-NOR, BIT-NOT, BIT-NOTC1, BIT-NOTC2, BIT-XOR

### Syntax:

**bit-and** *bit-array1 bit-array2 &optional opt-arg => resulting-bit-array*

**bit-andc1** *bit-array1 bit-array2 &optional opt-arg => resulting-bit-array*

**bit-andc2** *bit-array1 bit-array2 &optional opt-arg => resulting-bit-array*

**bit-eqv** *bit-array1 bit-array2 &optional opt-arg => resulting-bit-array*

**bit-ior** *bit-array1 bit-array2 &optional opt-arg => resulting-bit-array*

**bit-nand** *bit-array1 bit-array2 &optional opt-arg => resulting-bit-array*

**bit-nor** *bit-array1 bit-array2 &optional opt-arg => resulting-bit-array*

Function BIT-AND, BIT-ANDC1, BIT-ANDC2, BIT-EQV, BIT-IOR, BIT-NAND, BIT-NOR, BIT-NOT, BIT-

## CLHS: Declaration DYNAMIC-EXTENT

**bit-orc1** *bit-array1* *bit-array2* &optional *opt-arg* => *resulting-bit-array*

**bit-orc2** *bit-array1* *bit-array2* &optional *opt-arg* => *resulting-bit-array*

**bit-xor** *bit-array1* *bit-array2* &optional *opt-arg* => *resulting-bit-array*

**bit-not** *bit-array* &optional *opt-arg* => *resulting-bit-array*

### Arguments and Values:

*bit-array*, *bit-array1*, *bit-array2*—[a bit array](#).

*Opt-arg*—[a bit array](#), or **t**, or **nil**. The default is **nil**.

*Bit-array*, *bit-array1*, *bit-array2*, and *opt-arg* (if an [array](#)) must all be of the same [rank](#) and [dimensions](#).

*resulting-bit-array*—[a bit array](#).

### Description:

These functions perform bit-wise logical operations on *bit-array1* and *bit-array2* and return an [array](#) of matching [rank](#) and [dimensions](#), such that any given bit of the result is produced by operating on corresponding bits from each of the arguments.

In the case of **bit-not**, an [array](#) of [rank](#) and [dimensions](#) matching *bit-array* is returned that contains a copy of *bit-array* with all the bits inverted.

If *opt-arg* is of type ([array](#) [bit](#)) the contents of the result are destructively placed into *opt-arg*. If *opt-arg* is the symbol **t**, *bit-array* or *bit-array1* is replaced with the result; if *opt-arg* is **nil** or omitted, a new [array](#) is created to contain the result.

The next figure indicates the logical operation performed by each of the [functions](#).

Function	Operation
-----	-----
<a href="#">bit-and</a>	and
<a href="#">bit-eqv</a>	equivalence (exclusive nor)
<a href="#">bit-not</a>	complement
<a href="#">bit-ior</a>	inclusive or
<a href="#">bit-xor</a>	exclusive or
<a href="#">bit-nand</a>	complement of bit-array1 and bit-array2
<a href="#">bit-nor</a>	complement of bit-array1 or bit-array2
<a href="#">bit-andc1</a>	and complement of bit-array1 with bit-array2
<a href="#">bit-andc2</a>	and complement of bit-array1 with bit-array2
<a href="#">bit-orc1</a>	or complement of bit-array1 with bit-array2
<a href="#">bit-orc2</a>	or complement of bit-array1 with bit-array2

Figure 15-4. Bit-wise Logical Operations on Bit Arrays

### Examples:

Function BIT-AND, BIT-ANDC1, BIT-ANDC2, BIT-EQV, BIT-IOR, BIT-NAND, BIT-NOR, BIT-NOT, BIT-

## CLHS: Declaration DYNAMIC-EXTENT

```
(bit-and (setq ba #*11101010) #*01101011) => #*01101010
(bit-and #*1100 #*1010) => #*1000
(bit-andc1 #*1100 #*1010) => #*0010
(setq rba (bit-andc2 ba #*00110011 t)) => #*11001000
(eq rba ba) => true
(bit-not (setq ba #*11101010)) => #*00010101
(setq rba (bit-not ba
                    (setq tba (make-array 8
                                         :element-type 'bit)))) => #*00010101
(equal rba tba) => true
(bit-xor #*1100 #*1010) => #*0110
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**lognot, logand**

**Notes:** None.

## Accessor BIT, SBIT

**Syntax:**

**bit** *bit–array &rest subscripts => bit*

**sbit** *bit–array &rest subscripts => bit*

(**setf** (**bit** *bit–array &rest subscripts*) *new-bit*)

(**setf** (**sbit** *bit–array &rest subscripts*) *new-bit*)

**Arguments and Values:**

*bit–array*—for **bit**, a *bit array*; for **sbit**, a *simple bit array*.

*subscripts*—a *list* of *valid array indices* for the *bit–array*.

*bit*—a *bit*.

**Description:**

**bit** and **sbit access** the *bit–array element* specified by *subscripts*.

These *functions* ignore the *fill pointer* when accessing *elements*.

**Examples:**

```
(bit (setq ba (make-array 8
```

## CLHS: Declaration DYNAMIC-EXTENT

```
: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
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[aref](#), [Section 3.2.1 \(Compiler Terminology\)](#)

**Notes:**

**bit** and **sbit** are like **aref** except that they require *arrays* to be a *bit array* and a *simple bit array*, respectively.

**bit** and **sbit**, unlike **char** and **schar**, allow the first argument to be an *array* of any *rank*.

## Function BIT-VECTOR-P

**Syntax:**

**bit-vector-p** *object* => *generalized-boolean*

**Arguments and Values:**

*object*—an *object*.

*generalized-boolean*—a *generalized boolean*.

**Description:**

Returns *true* if *object* is of type **bit-vector**; otherwise, returns *false*.

**Examples:**

```
(bit-vector-p (make-array 6
                           :element-type 'bit
                           :fill-pointer t)) => true
(bit-vector-p #*) => true
(bit-vector-p (make-array 6)) => false
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**typep****Notes:**

```
(bit-vector-p object) == (typep object 'bit-vector)
```

**Function BUTLAST, NBUTLAST****Syntax:**

**butlast** *list* &optional *n* => *result-list*

**nbutlast** *list* &optional *n* => *result-list*

**Arguments and Values:**

*list*—a *list*, which might be a *dotted list* but must not be a *circular list*.

*n*—a non-negative *integer*.

*result-list*—a *list*.

**Description:**

**butlast** returns a copy of *list* from which the last *n* conses have been omitted. If *n* is not supplied, its value is 1. If there are fewer than *n* conses in *list*, **nil** is returned and, in the case of **nbutlast**, *list* is not modified.

**nbutlast** is like **butlast**, but **nbutlast** may modify *list*. It changes the *cdr* of the *cons* *n*+1 from the end of the *list* to **nil**.

**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 '(a b c d)) => (A B C)
(butlast '((a b) (c d))) => ((A B))
(butlast '(a)) => NIL
(butlast nil) => NIL
(setq foo (list 'a 'b 'c 'd)) => (A B C D)
(nbutlast foo) => (A B C)
foo => (A B C)
(nbutlast (list 'a)) => NIL
(nbutlast '()) => NIL
```

**Affected By:** None.

**Exceptional Situations:**

Should signal an error of **type type-error** if *list* is not a proper list or a dotted list. Should signal an error of **type type-error** if *n* is not a non-negative integer.

**See Also:** None.

**Notes:**

```
(butlast list n) == (ldiff list (last list n))
```

**Function BYTE, BYTE-SIZE, BYTE-POSITION****Syntax:**

**byte** *size position => bytespec*

**byte-size** *bytespec => size*

**byte-position** *bytespec => position*

**Arguments and Values:**

*size, position*—a non-negative integer.

*bytespec*—a byte specifier.

**Description:**

**byte** returns a byte specifier that indicates a byte of width *size* and whose bits have weights  $2^{position + size - 1}$  through  $2^position$ , and whose representation is implementation-dependent.

**byte-size** returns the number of bits specified by *bytespec*.

**byte-position** returns the position specified by *bytespec*.

**Examples:**

```
(setq b (byte 100 200)) => #<BYTE-SPECIFIER size 100 position 200>
(byte-size b) => 100
(byte-position b) => 200
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**ldb, dpb**

**Notes:**

```
(byte-size (byte j k)) == j
(byte-position (byte j k)) == k
```

A byte of size 0 is permissible; it refers to a byte of width zero. For example,

```
(ldb (byte 0 3) #o7777) => 0
(dpdb #o7777 (byte 0 3) 0) => 0
```

**Local Function CALL-NEXT-METHOD****Syntax:**

**call-next-method** &*rest args* => *result\**

**Arguments and Values:**

*arg*—an object.

*results*—the values returned by the method it calls.

**Description:**

The function call-next-method can be used within the body forms (but not the lambda list) of a method defined by a method-defining form to call the next method.

If there is no next method, the generic function **no-next-method** is called.

The type of method combination used determines which methods can invoke **call-next-method**. The standard method combination type allows **call-next-method** to be used within primary methods and around methods. For generic functions using a type of method combination defined by the short form of **define-method-combination**, **call-next-method** can be used in around methods only.

When **call-next-method** is called with no arguments, it passes the current method's original arguments to the next method. Neither argument defaulting, nor using **setq**, nor rebinding variables with the same names as parameters of the method affects the values **call-next-method** passes to the method it calls.

When **call-next-method** is called with arguments, the next method is called with those arguments.

If **call-next-method** is called with arguments but omits optional arguments, the next method called defaults those arguments.

The function call-next-method returns any values that are returned by the next method.

The function call-next-method has lexical scope and indefinite extent and can only be used within the body of a method defined by a method-defining form.

Whether or not **call-next-method** is fbound in the global environment is implementation-dependent; however, the restrictions on redefinition and shadowing of **call-next-method** are the same as for symbols in the COMMON-LISP package which are fbound in the global environment. The consequences of attempting to

## CLHS: Declaration DYNAMIC-EXTENT

use **call-next-method** outside of a *method-defining form* are undefined.

**Examples:** None.

**Affected By:**

**defmethod, call-method, define-method-combination.**

**Exceptional Situations:**

When providing arguments to **call-next-method**, the following rule must be satisfied or an error of *type error* should be signaled: the ordered set of *applicable methods* for a changed set of arguments for **call-next-method** must be the same as the ordered set of *applicable methods* for the original arguments to the *generic function*. Optimizations of the error checking are possible, but they must not change the semantics of **call-next-method**.

**See Also:**

**define-method-combination, defmethod, next-method-p, no-next-method, call-method, Section 7.6.6 (Method Selection and Combination), Section 7.6.6.2 (Standard Method Combination), Section 7.6.6.4 (Built-in Method Combination Types)**

**Notes:** None.

CLHS: Declaration DYNAMIC-EXTENT

**Accessor CAR, CDR, CAAR, CADR, CDAR, CDDR, CAAAR, CAADR, CADAR, CADDR, CDAAR, CDADR, CDDAR, CDDDR, CAAAAR, CAAADR, CAADAR, CAADDR, CADAAR, CADADR, CADDAR, CADDDR, CDAAAR, CDAADR, CDADAR, CDADDR, CDDAAR, CDDADR, CDDDR, CDDDDR**

## Syntax:

**car**  $x \Rightarrow object$

**cdr**  $x \Rightarrow object$

**caar**  $x \Rightarrow object$

**cadr**  $x \Rightarrow object$

**cdar**  $x \Rightarrow object$

**caddr**  $x \Rightarrow object$

**caaar**  $x \Rightarrow object$

**caadr** *x => object*

**cadar**  $x \Rightarrow object$

**caddr**  $x \Rightarrow object$

**cdaar**  $x \Rightarrow object$

**cdadr** *x* => *object*

**cddar**  $x \Rightarrow object$

**cddd** *x => object*

**caaaar**  $x \Rightarrow object$

**caaaddr**  $x \Rightarrow object$

**caadar**  $x \Rightarrow object$

**caaddr** *x* => *object*

**cadaar**  $x \Rightarrow object$

**cadaddr**  $x \Rightarrow object$

**caddr**  $x \Rightarrow object$

**caddr** *x => object*

## CLHS: Declaration DYNAMIC-EXTENT

**cdaaar**  $x \Rightarrow object$

**cdaadr**  $x \Rightarrow object$

**cdadar**  $x \Rightarrow object$

**cdaddr**  $x \Rightarrow object$

**cddaar**  $x \Rightarrow object$

**cddadr**  $x \Rightarrow object$

**cdddar**  $x \Rightarrow object$

**cddddr**  $x \Rightarrow object$

(**setf** (**car**  $x$ )  $new-object$ )

(**setf** (**cdr**  $x$ )  $new-object$ )

(**setf** (**caar**  $x$ )  $new-object$ )

(**setf** (**cadr**  $x$ )  $new-object$ )

(**setf** (**cdar**  $x$ )  $new-object$ )

(**setf** (**cddr**  $x$ )  $new-object$ )

(**setf** (**caaar**  $x$ )  $new-object$ )

(**setf** (**caaddr**  $x$ )  $new-object$ )

(**setf** (**cadar**  $x$ )  $new-object$ )

(**setf** (**caddr**  $x$ )  $new-object$ )

(**setf** (**cdaar**  $x$ )  $new-object$ )

(**setf** (**cdadr**  $x$ )  $new-object$ )

(**setf** (**cddar**  $x$ )  $new-object$ )

(**setf** (**cdddr**  $x$ )  $new-object$ )

(**setf** (**caaaaar**  $x$ )  $new-object$ )

(**setf** (**caaaddr**  $x$ )  $new-object$ )

(**setf** (**caadar**  $x$ )  $new-object$ )

(**setf** (**caaddr**  $x$ )  $new-object$ )

## CLHS: Declaration DYNAMIC-EXTENT

```
(setf (cadaar x) new-object)  
  
(setf (cadadr x) new-object)  
  
(setf (caddar x) new-object)  
  
(setf (cadddr x) new-object)  
  
(setf (cdaaar x) new-object)  
  
(setf (cdaadr x) new-object)  
  
(setf (cdadar x) new-object)  
  
(setf (cdaddr x) new-object)  
  
(setf (cddaar x) new-object)  
  
(setf (cddadr x) new-object)  
  
(setf (cdddar x) new-object)  
  
(setf (cddddr x) new-object)
```

### Pronunciation:

**cadr**: ['ka,duhr]

**caddr**: ['kaduh,duhr] or ['ka,dduhr]

**cdr**: ['k,duhr]

**cddr**: ['kduh,duhr] or ['kuh,dduhr]

### Arguments and Values:

*x*---a list.

*object*---an object.

*new-object*---an object.

### Description:

If *x* is a cons, **car** returns the car of that cons. If *x* is nil, **car** returns nil.

If *x* is a cons, **cdr** returns the cdr of that cons. If *x* is nil, **cdr** returns nil.

Functions are provided which perform compositions of up to four **car** and **cdr** operations. Their names consist of a C, followed by two, three, or four occurrences of A or D, and finally an R. The series of A's and D's in each function's name is chosen to identify the series of **car** and **cdr** operations that is performed by the function.

Accessor CAR, CDR, CAAR, CADR, CDAR, CDDR, CAAAR, CAADR, CADAR, CADDR, CDAAR, CDDAR, CDDDR

CLHS: Declaration DYNAMIC-EXTENT

The order in which the A's and D's appear is the inverse of the order in which the corresponding operations are performed. The next figure defines the relationships precisely.

This <u>place</u> ...	Is equivalent to this <u>place</u> ...
(caar x)	(car (car x))
(cadr x)	(car (cdr x))
(cdar x)	(cdr (car x))
(cddr x)	(cdr (cdr x))
(caaar x)	(car (car (car x))))
(caaddr x)	(car (car (cdr x))))
(cadar x)	(car (cdr (car x))))
(caddr x)	(car (cdr (cdr x))))
(cdaar x)	(cdr (car (car x))))
(cdadr x)	(cdr (car (cdr x))))
(cddar x)	(cdr (cdr (car x))))
(cddd़r x)	(cdr (cdr (cdr x))))
(caaaaar x)	(car (car (car (car x)))))
(caaaddr x)	(car (car (car (cdr x)))))
(caadar x)	(car (car (cdr (car x)))))
(caaddr x)	(car (car (cdr (cdr x)))))
(cadaar x)	(car (cdr (car (car x)))))
(cadadr x)	(car (cdr (car (cdr x)))))
(caddar x)	(car (cdr (cdr (car x)))))
(cadddr x)	(car (cdr (cdr (cdr x)))))
(cdaaar x)	(cdr (car (car (car x)))))
(cdaaddr x)	(cdr (car (car (cdr x)))))
(cdadar x)	(cdr (car (cdr (car x)))))
(cdaddr x)	(cdr (car (cdr (cdr x)))))
(cddaar x)	(cdr (cdr (car (car x)))))
(cddadar x)	(cdr (cdr (car (cdr x)))))
(cdddar x)	(cdr (cdr (cdr (car x)))))
(cddddr x)	(cdr (cdr (cdr (cdr x)))))

**Figure 14–6.** CAR and CDR variants

**setf** can also be used with any of these functions to change an existing component of *x*, but **setf** will not make new components. So, for example, the **car** of a **cons** can be assigned with **setf** of **car**, but the **car** of **nil** cannot be assigned with **setf** of **car**. Similarly, the **car** of the **car** of a **cons** whose **car** is a **cons** can be assigned with **setf** of **caar**, but neither **nil** nor a **cons** whose **car** is **nil** can be assigned with **setf** of **caar**.

The argument  $x$  is permitted to be a *dotted list* or a *circular list*.

## Examples:

```
(car nil) => NIL
(cdr '(1 . 2)) => 2
(cdr '(1 2)) => (2)
(cadr '(1 2)) => 2
(car '(a b c)) => A
(cdr '(a b c)) => (B C)
```

**Affected By:** None.

### **Exceptional Situations:**

The functions `car` and `cdr` should signal `type-error` if they receive an argument which is not a `list`. The other functions (`caar`, `cadr`, ... `cdaddr`) should behave for the purpose of error checking as if defined by appropriate

calls to [car](#) and [cdr](#).

See Also:

[rplaca](#), [first](#), [rest](#)

Notes:

The [car](#) of a [cons](#) can also be altered by using [rplaca](#), and the [cdr](#) of a [cons](#) can be altered by using [rplacd](#).

```
(car x)    ==  (first x)
(cadr x)   ==  (second x) ==  (car (cdr x))
(caddr x)  ==  (third x) ==  (car (cdr (cdr x)))
(cadddr x) ==  (fourth x) ==  (car (cdr (cdr (cdr x))))
```

## Function CELL-ERROR-NAME

Syntax:

**cell-error-name** *condition => name*

Arguments and Values:

*condition*—a [condition](#) of type **cell-error**.

*name*—an [object](#).

Description:

Returns the [name](#) of the offending cell involved in the [situation](#) represented by *condition*.

The nature of the result depends on the specific [type](#) of *condition*. For example, if the *condition* is of [type unbound-variable](#), the result is the [name](#) of the [unbound variable](#) which was being *accessed*, if the *condition* is of [type undefined-function](#), this is the [name](#) of the [undefined function](#) which was being *accessed*, and if the *condition* is of [type unbound-slot](#), this is the [name](#) of the [slot](#) which was being *accessed*.

Examples: None.

Affected By: None.

Exceptional Situations: None.

See Also:

[cell-error](#), [unbound-slot](#), [unbound-variable](#), [undefined-function](#), [Section 9.1 \(Condition System Concepts\)](#)

Notes: None.

## Function CERROR

### Syntax:

**cerror** *continue-format-control* *datum* &rest *arguments* => ***nil***

### Arguments and Values:

*Continue-format-control*—a format control.

*datum*, *arguments*—designators for a condition of default type **simple-error**.

### Description:

**cerror** effectively invokes **error** on the condition named by *datum*. As with any function that implicitly calls **error**, if the condition is not handled, (*invoke-debugger condition*) is executed. While signaling is going on, and while in the debugger if it is reached, it is possible to continue code execution (i.e., to return from **cerror**) using the continue restart.

If *datum* is a condition, *arguments* can be supplied, but are used only in conjunction with the *continue-format-control*.

### Examples:

```
(defun real-sqrt (n)
  (when (minusp n)
    (setq n (- n))
    (cerror "Return sqrt(~D) instead." "Tried to take sqrt(~D)." n)
    (sqrt n))

  (real-sqrt 4)
=> 2.0

  (real-sqrt -9)
>> Correctable error in REAL-SQRT: Tried to take sqrt(-9).
>> Restart options:
>>   1: Return sqrt(9) instead.
>>   2: Top level.
>> Debug> :continue 1
=> 3.0

(define-condition not-a-number (error)
  ((argument :reader not-a-number-argument :initarg :argument))
  (:report (lambda (condition stream)
             (format stream "~S is not a number."
                     (not-a-number-argument condition)))))

(defun assure-number (n)
  (loop (when (numberp n) (return n))
        (cerror "Enter a number."
                'not-a-number :argument n)
        (format t "~&Type a number: ")
        (setq n (read))
        (fresh-line)))

  (assure-number 'a))
```

## CLHS: Declaration DYNAMIC-EXTENT

```

>> Correctable error in ASSURE-NUMBER: A is not a number.
>> Restart options:
>> 1: Enter a number.
>> 2: Top level.
>> Debug> :continue 1
>> Type a number: 1/2
=> 1/2

(defun assure-large-number (n)
  (loop (when (and (numberp n) (> n 73)) (return n))
        (cerror "Enter a number~:[~; a bit larger than ~D~]."
                "~*~A is not a large number."
                (numberp n) n)
        (format t "~&Type a large number: ")
        (setq n (read))
        (fresh-line)))

(assure-large-number 10000)
=> 10000

(assure-large-number 'a)
>> Correctable error in ASSURE-LARGE-NUMBER: A is not a large number.
>> Restart options:
>> 1: Enter a number.
>> 2: Top level.
>> Debug> :continue 1
>> Type a large number: 88
=> 88

(assure-large-number 37)
>> Correctable error in ASSURE-LARGE-NUMBER: 37 is not a large number.
>> Restart options:
>> 1: Enter a number a bit larger than 37.
>> 2: Top level.
>> Debug> :continue 1
>> Type a large number: 259
=> 259

(define-condition not-a-large-number (error)
  ((argument :reader not-a-large-number-argument :initarg :argument)
   (:report (lambda (condition stream)
              (format stream "~S is not a large number."
                      (not-a-large-number-argument condition)))))

(defun assure-large-number (n)
  (loop (when (and (numberp n) (> n 73)) (return n))
        (cerror "Enter a number~3*~:[~; a bit larger than ~*~D~]."
                'not-a-large-number
                :argument n
                :ignore (numberp n)
                :ignore n
                :allow-other-keys t)
        (format t "~&Type a large number: ")
        (setq n (read))
        (fresh-line)))

(assure-large-number 'a)
>> Correctable error in ASSURE-LARGE-NUMBER: A is not a large number.
>> Restart options:
>> 1: Enter a number.

```

## CLHS: Declaration DYNAMIC-EXTENT

```
>> 2: Top level.  
>> Debug> :continue 1  
>> Type a large number: 88  
=> 88  
  
(assure-large-number 37)  
>> Correctable error in ASSURE-LARGE-NUMBER: A is not a large number.  
>> Restart options:  
>> 1: Enter a number a bit larger than 37.  
>> 2: Top level.  
>> Debug> :continue 1  
>> Type a large number: 259  
=> 259
```

### Affected By:

#### \*break-on-signals\*

Existing handler bindings.

**Exceptional Situations:** None.

### See Also:

[error](#), [format](#), [handler-bind](#), [\\*break-on-signals\\*](#), [simple-type-error](#)

### Notes:

If *datum* is a condition type rather than a string, the **format** directive `~*` may be especially useful in the continue-format-control in order to ignore the keywords in the initialization argument list. For example:

```
(cerror "enter a new value to replace ~*~s"  
       'not-a-number  
       :argument a)
```

## Function CHARACTER

### Syntax:

**character** *character* => *denoted-character*

### Arguments and Values:

*character*—a character designator.

*denoted-character*—a character.

### Description:

Returns the character denoted by the character designator.

### Examples:

## CLHS: Declaration DYNAMIC-EXTENT

```
(character #\a) => #\a
(character "a") => #\a
(character 'a) => #\A
(character '\a) => #\a
(character 65.) is an error.
(character 'apple) is an error.
```

**Affected By:** None.

**Exceptional Situations:**

Should signal an error of ***type type-error*** if *object* is not a ***character designator***.

**See Also:**

**coerce**

**Notes:**

```
(character object) == (coerce object 'character)
```

## Accessor CHAR, SCHAR

**Syntax:**

**char** *string index => character*

**schar** *string index => character*

```
(setf (char string index) new-character)
```

```
(setf (schar string index) new-character)
```

**Arguments and Values:**

*string*—for **char**, a ***string***; for **schar**, a ***simple string***.

*index*—a ***valid array index*** for the *string*.

*character, new-character*—a ***character***.

**Description:**

**char** and **schar** access the ***element*** of *string* specified by *index*.

**char** ignores ***fill pointers*** when accessing ***elements***.

**Examples:**

```
(setq my-simple-string (make-string 6 :initial-element #\A)) => "AAAAAA"
(schar my-simple-string 4) => #\A
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(setf (schar my-simple-string 4) #\B) => #\B
my-simple-string => "AAAABA"
(setq my-filled-string
      (make-array 6 :element-type 'character
                  :fill-pointer 5
                  :initial-contents my-simple-string))
=> "AAAAB"
(char my-filled-string 4) => #\B
(char my-filled-string 5) => #\A
(setf (char my-filled-string 3) #\C) => #\C
(setf (char my-filled-string 5) #\D) => #\D
(setf (fill-pointer my-filled-string) 6) => 6
my-filled-string => "AACBD"
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[aref, elt, Section 3.2.1 \(Compiler Terminology\)](#)

**Notes:**

```
(char s j) == (aref (the string s) j)
```

## Function CHAR-CODE

**Syntax:**

**char-code** *character* => *code*

**Arguments and Values:**

*character*—a character.

*code*—a character code.

**Description:**

**char-code** returns the code attribute of *character*.

**Examples:**

```
; ; An implementation using ASCII character encoding
; ; might return these values:
(char-code #\$) => 36
(char-code #\a) => 97
```

**Affected By:** None.

**Exceptional Situations:**

Should signal an error of type type-error if *character* is not a character.

**See Also:****char-code-limit****Notes:** None.***Function CHAR-INT*****Syntax:****char-int** *character => integer***Arguments and Values:***character*—a character.*integer*—a non-negative integer.**Description:**

Returns a non-negative integer encoding the *character* object. The manner in which the integer is computed is implementation-dependent. In contrast to sxhash, the result is not guaranteed to be independent of the particular Lisp image.

If *character* has no implementation-defined attributes, the results of **char-int** and **char-code** are the same.

```
(char= c1 c2) == (= (char-int c1) (char-int c2))
```

for characters *c1* and *c2*.

**Examples:**

```
(char-int #\A) => 65      ; implementation A
(char-int #\A) => 577     ; implementation B
(char-int #\A) => 262145  ; implementation C
```

**Affected By:** None.**Exceptional Situations:** None.**See Also:****char-code****Notes:** None.***Function CHAR-NAME*****Syntax:****char-name** *character => name*

**Arguments and Values:**

*character*—a character.

*name*—a string or nil.

**Description:**

Returns a string that is the name of the *character*, or nil if the *character* has no name.

All non-graphic characters are required to have names unless they have some implementation-defined attribute which is not null. Whether or not other characters have names is implementation-dependent.

The standard characters <Newline> and <Space> have the respective names "Newline" and "Space". The semi-standard characters <Tab>, <Page>, <Rubout>, <Linefeed>, <Return>, and <Backspace> (if they are supported by the implementation) have the respective names "Tab", "Page", "Rubout", "Linefeed", "Return", and "Backspace" (in the indicated case, even though name lookup by "#\" and by the function name-char is not case sensitive).

**Examples:**

```
(char-name #\ ) => "Space"
(char-name #\Space) => "Space"
(char-name #\Page) => "Page"

(char-name #\a)
=> NIL
OR=> "LOWERCASE-a"
OR=> "Small-A"
OR=> "LA01"

(char-name #\A)
=> NIL
OR=> "UPPERCASE-A"
OR=> "Capital-A"
OR=> "LA02"

;; Even though its CHAR-NAME can vary, #\A prints as #\A
(prin1-to-string (read-from-string (format nil "#\~A" (or (char-name #\A) "A"))))
=> "#\A"
```

**Affected By:** None.

**Exceptional Situations:**

Should signal an error of type type-error if *character* is not a character.

**See Also:**

name-char, Section 22.1.3.2 (Printing Characters)

**Notes:**

## CLHS: Declaration DYNAMIC-EXTENT

Non-graphic characters having names are written by the Lisp printer as "#\" followed by the their name; see Section 22.1.3.2 (Printing Characters).

## Function CHAR-UPCASE, CHAR-DOWNCASE

### Syntax:

**char-upcase** *character* => *corresponding-character*

**char-downcase** *character* => *corresponding-character*

### Arguments and Values:

*character, corresponding-character*—a character.

### Description:

If *character* is a lowercase character, **char-upcase** returns the corresponding uppercase character. Otherwise, **char-upcase** just returns the given *character*.

If *character* is an uppercase character, **char-downcase** returns the corresponding lowercase character. Otherwise, **char-downcase** just returns the given *character*.

The result only ever differs from *character* in its code attribute; all implementation-defined attributes are preserved.

### Examples:

```
(char-upcase #\a) => #\A
(char-upcase #\A) => #\A
(char-downcase #\a) => #\a
(char-downcase #\A) => #\a
(char-upcase #\9) => #\9
(char-downcase #\9) => #\9
(char-upcase #\@) => #\@
(char-downcase #\@) => #\@
;; Note that this next example might run for a very long time in
;; some implementations if CHAR-CODE-LIMIT happens to be very large
;; for that implementation.
(dotimes (code char-code-limit)
  (let ((char (code-char code)))
    (when char
      (unless (cond ((upper-case-p char) (char= (char-upcase (char-downcase char)) char))
                    ((lower-case-p char) (char= (char-downcase (char-upcase char)) char))
                    (t (and (char= (char-upcase (char-downcase char)) char)
                            (char= (char-downcase (char-upcase char)) char))))
        (return char)))))  
=> NIL
```

**Affected By:** None.

### Exceptional Situations:

## CLHS: Declaration DYNAMIC-EXTENT

Should signal an error of type type-error if *character* is not a character.

See Also:

upper-case-p, alpha-char-p, Section 13.1.4.3 (Characters With Case), Section 13.1.10 (Documentation of Implementation-Defined Scripts)

Notes:

If the *corresponding-char* is different than *character*, then both the *character* and the *corresponding-char* have case.

Since char-equal ignores the case of the characters it compares, the *corresponding-character* is always the same as *character* under char-equal.

**Function CHAR=, CHAR/=, CHAR<, CHAR>, CHAR<=, CHAR>=, CHAR-EQUAL, CHAR-NOT-EQUAL, CHAR-LESSP, CHAR-GREATERP, CHAR-NOT-GREATERP, CHAR-NOT-LESSP**

Syntax:

**char=** &rest *characters+* => *generalized-boolean*

**char/=** &rest *characters+* => *generalized-boolean*

**char<** &rest *characters+* => *generalized-boolean*

**char>** &rest *characters+* => *generalized-boolean*

**char<=** &rest *characters+* => *generalized-boolean*

**char>=** &rest *characters+* => *generalized-boolean*

**char-equal** &rest *characters+* => *generalized-boolean*

**char-not-equal** &rest *characters+* => *generalized-boolean*

**char-lessp** &rest *characters+* => *generalized-boolean*

## CLHS: Declaration DYNAMIC-EXTENT

**char=greaterp** &rest *characters*+ => *generalized-boolean*

**char-not-greaterp** &rest *characters*+ => *generalized-boolean*

**char-not-lessp** &rest *characters*+ => *generalized-boolean*

### Arguments and Values:

*character*—a character.

*generalized-boolean*—a generalized boolean.

### Description:

These predicates compare characters.

**char=** returns true if all *characters* are the same; otherwise, it returns false. If two *characters* differ in any implementation-defined attributes, then they are not **char=**.

**char/=** returns true if all *characters* are different; otherwise, it returns false.

**char<** returns true if the *characters* are monotonically increasing; otherwise, it returns false. If two *characters* have identical implementation-defined attributes, then their ordering by **char<** is consistent with the numerical ordering by the predicate **<** on their codes.

**char>** returns true if the *characters* are monotonically decreasing; otherwise, it returns false. If two *characters* have identical implementation-defined attributes, then their ordering by **char>** is consistent with the numerical ordering by the predicate **>** on their codes.

**char<=** returns true if the *characters* are monotonically nondecreasing; otherwise, it returns false. If two *characters* have identical implementation-defined attributes, then their ordering by **char<=** is consistent with the numerical ordering by the predicate **<=** on their codes.

**char>=** returns true if the *characters* are monotonically nonincreasing; otherwise, it returns false. If two *characters* have identical implementation-defined attributes, then their ordering by **char>=** is consistent with the numerical ordering by the predicate **>=** on their codes.

**char-equal**, **char-not-equal**, **char-lessp**, **char-greaterp**, **char-not-greaterp**, and **char-not-lessp** are similar to **char=**, **char/=**, **char<**, **char>**, **char<=**, **char>=**, respectively, except that they ignore differences in case and might have an implementation-defined behavior for non-simple characters. For example, an implementation might define that **char-equal**, etc. ignore certain implementation-defined attributes. The effect, if any, of each implementation-defined attribute upon these functions must be specified as part of the definition of that attribute.

### Examples:

```
(char= #\d #\d) => true
(char= #\A #\a) => false
(char= #\d #\x) => false
(char= #\d #\D) => false
(char/= #\d #\d) => false
(char/= #\d #\x) => true
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(char/= #\d #\D) => true
(char= #\d #\d #\d #\d) => true
(char/= #\d #\d #\d #\d) => false
(char= #\d #\d #\x #\d) => false
(char/= #\d #\d #\x #\d) => false
(char= #\d #\y #\x #\c) => false
(char/= #\d #\y #\x #\c) => true
(char= #\d #\c #\d) => false
(char/= #\d #\c #\d) => false
(char< #\d #\x) => true
(char<= #\d #\x) => true
(char< #\d #\d) => false
(char<= #\d #\d) => true
(char< #\a #\e #\y #\z) => true
(char<= #\a #\e #\y #\z) => true
(char< #\a #\e #\e #\y) => false
(char<= #\a #\e #\e #\y) => true
(char> #\e #\d) => true
(char>= #\e #\d) => true
(char> #\d #\c #\b #\a) => true
(char>= #\d #\c #\b #\a) => true
(char> #\d #\d #\c #\a) => false
(char>= #\d #\d #\c #\a) => true
(char> #\e #\d #\b #\c #\a) => false
(char>= #\e #\d #\b #\c #\a) => false
(char> #\z #\A) => implementation-dependent
(char> #\Z #\a) => implementation-dependent
(char-equal #\A #\a) => true
(stable-sort (list #\b #\A #\B #\a #\c #\C) #'char-lessp)
=> (#\A #\a #\b #\B #\c #\C)
(stable-sort (list #\b #\A #\B #\a #\c #\C) #'char<)
=> (#\A #\B #\C #\a #\b #\c) ;Implementation A
=> (#\a #\b #\c #\A #\B #\C) ;Implementation B
=> (#\a #\A #\b #\B #\c #\C) ;Implementation C
=> (#\A #\a #\B #\b #\C #\c) ;Implementation D
=> (#\A #\B #\a #\b #\C #\c) ;Implementation E
```

**Affected By:** None.

**Exceptional Situations:**

Should signal an error of **type program-error** if at least one *character* is not supplied.

**See Also:**

**Notes:**

If characters differ in their **code attribute** or any **implementation-defined attribute**, they are considered to be different by **char=**.

There is no requirement that (**eq** *c1* *c2*) be true merely because (**char=** *c1* *c2*) is **true**. While **eq** can distinguish two **characters** that **char=** does not, it is distinguishing them not as **characters**, but in some sense on the basis of a lower level implementation characteristic. If (**eq** *c1* *c2*) is **true**, then (**char=** *c1* *c2*) is also true. **eql** and **equal** compare **characters** in the same way that **char=** does.

The manner in which **case** is used by **char=equal**, **char-not=equal**, **char-lessp**, **char-greaterp**, **char-not-greaterp**, and **char-not-lessp** implies an ordering for **standard characters** such that *A=a*, *B=b*,

Function CHAR=, CHAR/=, CHAR<, CHAR>, CHAR<=, CHAR>=, CHAR-EQUAL, CHAR-NOT-EQUAL, C

and so on, up to  $Z=z$ , and furthermore either  $9 < A$  or  $Z < 0$ .

## **Standard Generic Function CHANGE-CLASS**

### Syntax:

**change-class** *instance new-class &key &allow-other-keys => instance*

### Method Signatures:

**change-class** (*instance standard-object*) (*new-class standard-class*) *&rest initargs*

**change-class** (*instance t*) (*new-class symbol*) *&rest initargs*

### Arguments and Values:

*instance*—an object.

*new-class*—a class designator.

*initargs*—an initialization argument list.

### Description:

The generic function change-class changes the class of an *instance* to *new-class*. It destructively modifies and returns the *instance*.

If in the old class there is any slot of the same name as a local slot in the *new-class*, the value of that slot is retained. This means that if the slot has a value, the value returned by slot-value after change-class is invoked is eql to the value returned by slot-value before change-class is invoked. Similarly, if the slot was unbound, it remains unbound. The other slots are initialized as described in Section 7.2 (Changing the Class of an Instance).

After completing all other actions, change-class invokes update-instance-for-different-class. The generic function update-instance-for-different-class can be used to assign values to slots in the transformed instance. See Section 7.2.2 (Initializing Newly Added Local Slots).

If the second of the above methods is selected, that method invokes change-class on *instance*, (find-class *new-class*), and the *initargs*.

### Examples:

```
(defclass position () ())

(defclass x-y-position (position)
  ((x :initform 0 :initarg :x)
   (y :initform 0 :initarg :y)))

(defclass rho-theta-position (position)
  ((rho :initform 0)
   (theta :initform 0)))
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(defmethod update-instance-for-different-class :before ((old x-y-position)
                                                       (new rho-theta-position)
                                                       &key)
  ;; Copy the position information from old to new to make new
  ;; be a rho-theta-position at the same position as old.
  (let ((x (slot-value old 'x))
        (y (slot-value old 'y)))
    (setf (slot-value new 'rho) (sqrt (+ (* x x) (* y y))))
      (slot-value new 'theta) (atan y x)))

;; At this point an instance of the class x-y-position can be
;; changed to be an instance of the class rho-theta-position using
;; change-class:

(setq p1 (make-instance 'x-y-position :x 2 :y 0))

(change-class p1 'rho-theta-position)

;; The result is that the instance bound to p1 is now an instance of
;; the class rho-theta-position. The update-instance-for-different-class
;; method performed the initialization of the rho and theta slots based
;; on the value of the x and y slots, which were maintained by
;; the old instance.
```

**Examples:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[update-instance-for-different-class](#), [Section 7.2 \(Changing the Class of an Instance\)](#)

**Notes:**

The generic function [change-class](#) has several semantic difficulties. First, it performs a destructive operation that can be invoked within a [method](#) on an [instance](#) that was used to select that [method](#). When multiple [methods](#) are involved because [methods](#) are being combined, the [methods](#) currently executing or about to be executed may no longer be applicable. Second, some implementations might use compiler optimizations of slot [access](#), and when the [class of an instance](#) is changed the assumptions the compiler made might be violated. This implies that a programmer must not use [change-class](#) inside a [method](#) if any [methods](#) for that [generic function access](#) any [slots](#), or the results are undefined.

## Function CHARACTERP

**Syntax:**

**characterp** *object* => *generalized-boolean*

**Arguments and Values:**

*object*—an [object](#).

Function CHARACTERP

86

*generalized-boolean*---a generalized boolean.

#### Description:

Returns true if *object* is of type character; otherwise, returns false.

#### Examples:

```
(characterp #\a) => true
(characterp 'a) => false
(characterp "a") => false
(characterp 65.) => false
(characterp #\Newline) => true
;; This next example presupposes an implementation
;; in which #\Rubout is an implementation-defined character.
(characterp #\Rubout) => true
```

**Affected By:** None.

**Exceptional Situations:** None.

#### See Also:

character (type and function), typep

#### Notes:

```
(characterp object) == (typep object 'character)
```

## Function CIS

#### Syntax:

**cis radians** => *number*

#### Arguments and Values:

*radians*---a real.

*number*---a complex.

#### Description:

**cis** returns the value of  $e^{i \cdot \text{radians}}$ , which is a complex in which the real part is equal to the cosine of *radians*, and the imaginary part is equal to the sine of *radians*.

#### Examples:

```
(cis 0) => #C(1.0 0.0)
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**Notes:** None.

## ***Function CLASS-OF***

**Syntax:**

**class-of** *object* => *class*

**Arguments and Values:**

*object*—an *object*.

*class*—a *class object*.

**Description:**

Returns the *class* of which the *object* is a *direct instance*.

**Examples:**

```
(class-of 'fred) => #<BUILT-IN-CLASS SYMBOL 610327300>
(class-of 2/3) => #<BUILT-IN-CLASS RATIO 610326642>

(defclass book () ()) => #<STANDARD-CLASS BOOK 33424745>
(class-of (make-instance 'book)) => #<STANDARD-CLASS BOOK 33424745>

(defclass novel (book) ()) => #<STANDARD-CLASS NOVEL 33424764>
(class-of (make-instance 'novel)) => #<STANDARD-CLASS NOVEL 33424764>

(defstruct kons kar kdr) => KONS
(class-of (make-kons :kar 3 :kdr 4)) => #<STRUCTURE-CLASS KONS 250020317>
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**make-instance**, **type-of**

**Notes:** None.

## ***Standard Generic Function CLASS-NAME***

**Syntax:**

**class-name** *class => name*

**Method Signatures:**

**class-name** (*class class*)

**Arguments and Values:**

*class*—a class object.

*name*—a symbol.

**Description:**

Returns the name of the given *class*.

**Examples:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[find-class](#), [Section 4.3 \(Classes\)](#)

**Notes:**

If *S* is a symbol such that *S* = (class-name *C*) and *C* = (find-class *S*), then *S* is the proper name of *C*. For further discussion, see [Section 4.3 \(Classes\)](#).

The name of an anonymous class is nil.

## Function CLEAR-INPUT

**Syntax:**

**clear-input** &*optional input-stream => nil*

**Arguments and Values:**

*input-stream*—an input stream designator. The default is standard input.

**Description:**

Clears any available input from *input-stream*.

If clear-input does not make sense for *input-stream*, then clear-input does nothing.

**Examples:**

## CLHS: Declaration DYNAMIC-EXTENT

```
; ; The exact I/O behavior of this example might vary from implementation
; ; to implementation depending on the kind of interactive buffering that
; ; occurs. (The call to SLEEP here is intended to help even out the
; ; differences in implementations which do not do line-at-a-time buffering.)

(defun read-sleepily (&optional (clear-p nil) (zzz 0))
  (list (progn (print '>) (read))
        ; Note that input typed within the first ZZZ seconds
        ; will be discarded.
        (progn (print '>
)
        (if zzz (sleep zzz))
        (print '>>
)
        (if clear-p (clear-input))
        (read)))))

(read-sleepily)
>> > 10
>> >
>> >> 20
=> (10 20)

(read-sleepily t)
>> > 10
>> >
>> >> 20
=> (10 20)

(read-sleepily t 10)
>> > 10
>> > 20 ; Some implementations won't echo typeahead here.
>> >> 30
=> (10 30)
```

### Side Effects:

The *input-stream* is modified.

### Affected By:

#### \*standard-input\*

### Exceptional Situations:

Should signal an error of *type-type-error* if *input-stream* is not a *stream designator*.

### See Also:

#### clear-output

**Notes:** None.

## **Function CLOSE**

### Syntax:

**close** *stream* &*key* *abort* => *result*

**Arguments and Values:**

*stream*—a *stream* (either *open* or *closed*).

*abort*—a *generalized boolean*. The default is *false*.

*result*—*t* if the *stream* was *open* at the time it was received as an *argument*, or *implementation-dependent* otherwise.

**Description:**

**close** closes *stream*. Closing a *stream* means that it may no longer be used in input or output operations. The act of closing a *file stream* ends the association between the *stream* and its associated *file*; the transaction with the *file system* is terminated, and input/output may no longer be performed on the *stream*.

If *abort* is *true*, an attempt is made to clean up any side effects of having created *stream*. If *stream* performs output to a file that was created when the *stream* was created, the file is deleted and any previously existing file is not superseded.

It is permissible to close an already closed *stream*, but in that case the *result* is *implementation-dependent*.

After *stream* is closed, it is still possible to perform the following query operations upon it: **streamp**, **pathname**, **truename**, **merge-pathnames**, **pathname-host**, **pathname-device**, **pathname-directory**, **pathname-name**, **pathname-type**, **pathname-version**, **namestring**, **file-namestring**, **directory-namestring**, **host-namestring**, **enough-namestring**, **open**, **probe-file**, and **directory**.

The effect of **close** on a *constructed stream* is to close the argument *stream* only. There is no effect on the *constituents* of *composite streams*.

For a *stream* created with **make-string-output-stream**, the result of **get-output-stream-string** is unspecified after **close**.

**Examples:**

```
(setq s (make-broadcast-stream)) => #<BROADCAST-STREAM>
(close s) => T
(output-stream-p s) => true
```

**Side Effects:**

The *stream* is *closed* (if necessary). If *abort* is *true* and the *stream* is an *output file stream*, its associated *file* might be deleted.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**open**

**Notes:** None.

## Function CLRHASH

**Syntax:**

**clrhash** *hash-table* => *hash-table*

**Arguments and Values:**

*hash-table*—a hash table.

**Description:**

Removes all entries from *hash-table*, and then returns that empty hash table.

**Examples:**

```
(setq table (make-hash-table)) => #<HASH-TABLE EQL 0/120 32004073>
(dotimes (i 100) (setf (gethash i table) (format nil "~R" i))) => NIL
(hash-table-count table) => 100
(gethash 57 table) => "fifty-seven", true
(clrhash table) => #<HASH-TABLE EQL 0/120 32004073>
(hash-table-count table) => 0
(gethash 57 table) => NIL, false
```

**Side Effects:**

The *hash-table* is modified.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:** None.

**Notes:** None.

## Function COMPILE

**Syntax:**

**compile** *name &optional definition* => *function, warnings-p, failure-p*

**Arguments and Values:**

*name*—a function name, or nil.

*definition*—a lambda expression or a function. The default is the function definition of *name* if it names a function, or the macro function of *name* if it names a macro. The consequences are undefined if no *definition* is supplied when the *name* is nil.

*function*—the *function-name*, or a compiled function.

*warnings-p*—a generalized boolean.

*failure-p*—a generalized boolean.

#### Description:

Compiles an interpreted function.

**compile** produces a compiled function from *definition*. If the *definition* is a lambda expression, it is coerced to a function. If the *definition* is already a compiled function, **compile** either produces that function itself (i.e., is an identity operation) or an equivalent function.

If the *name* is nil, the resulting compiled function is returned directly as the primary value. If a non-nil name is given, then the resulting compiled function replaces the existing function definition of *name* and the *name* is returned as the primary value; if *name* is a symbol that names a macro, its macro function is updated and the *name* is returned as the primary value.

Literal objects appearing in code processed by the **compile** function are neither copied nor *coalesced*. The code resulting from the execution of **compile** references objects that are eql to the corresponding objects in the source code.

**compile** is permitted, but not required, to establish a handler for conditions of type error. For example, the handler might issue a warning and restart compilation from some implementation-dependent point in order to let the compilation proceed without manual intervention.

The secondary value, *warnings-p*, is false if no conditions of type error or warning were detected by the compiler, and true otherwise.

The tertiary value, *failure-p*, is false if no conditions of type error or warning (other than style-warning) were detected by the compiler, and true otherwise.

#### Examples:

```
(defun foo () "bar") => FOO
(compiled-function-p #'foo) => implementation-dependent
(compile 'foo) => FOO
(compiled-function-p #'foo) => true
(setf (symbol-function 'foo)
      (compile nil '(lambda () "replaced")))) => #<Compiled-Function>
(foo) => "replaced"
```

#### Affected By:

\*error-output\*, \*macroexpand-hook\*.

The presence of macro definitions and proclamations.

#### Exceptional Situations:

## CLHS: Declaration DYNAMIC-EXTENT

The consequences are undefined if the *lexical environment* surrounding the *function* to be compiled contains any *bindings* other than those for *macros*, *symbol macros*, or *declarations*.

For information about errors detected during the compilation process, see [Section 3.2.5 \(Exceptional Situations in the Compiler\)](#).

**See Also:**

[compile-file](#)

**Notes:** None.

## Function COMPILE-FILE-PATHNAME

**Syntax:**

**compile-file-pathname** *input-file* &*key output-file* &*allow-other-keys* => *pathname*

**Arguments and Values:**

*input-file*—a *pathname designator*. (Default fillers for unspecified components are taken from [\\*default-pathname-defaults\\*](#).)

*output-file*—a *pathname designator*. The default is *implementation-defined*.

*pathname*—a *pathname*.

**Description:**

Returns the *pathname* that [compile-file](#) would write into, if given the same arguments.

The defaults for the *output-file* are taken from the *pathname* that results from merging the *input-file* with the *value* of [\\*default-pathname-defaults\\*, except that the type component should default to the appropriate \*implementation-defined\* default type for \*compiled files\*.](#)

If *input-file* is a *logical pathname* and *output-file* is unsupplied, the result is a *logical pathname*. If *input-file* is a *logical pathname*, it is translated into a physical pathname as if by calling [translate-logical-pathname](#). If *input-file* is a *stream*, the *stream* can be either open or closed. [compile-file-pathname](#) returns the same *pathname* after a file is closed as it did when the file was open. It is an error if *input-file* is a *stream* that is created with [make-two-way-stream](#), [make-echo-stream](#), [make-broadcast-stream](#), [make-concatenated-stream](#), [make-string-input-stream](#), [make-string-output-stream](#).

If an implementation supports additional keyword arguments to [compile-file](#), [compile-file-pathname](#) must accept the same arguments.

**Examples:**

See [logical-pathname-translations](#).

**Affected By:** None.

**Exceptional Situations:**

An error of **type file-error** might be signaled if either *input-file* or *output-file* is wild.

**See Also:**

**compile-file**, **pathname**, **logical-pathname**, Section 20.1 (File System Concepts), Section 19.1.2 (Pathnames as Filenames)

**Notes:** None.

## Function COMPILE-FILE

**Syntax:**

**compile-file** *input-file* &*key output-file verbose print external-format*

=> *output-truename, warnings-p, failure-p*

**Arguments and Values:**

*input-file*—a pathname designator. (Default fillers for unspecified components are taken from \*default-pathname-defaults\*.)

*output-file*—a pathname designator. The default is implementation-defined.

*verbose*—a generalized boolean. The default is the value of \*compile-verbose\*.

*print*—a generalized boolean. The default is the value of \*compile-print\*.

*external-format*—an external file format designator. The default is :default.

*output-truename*—a pathname (the truename of the output file), or nil.

*warnings-p*—a generalized boolean.

*failure-p*—a generalized boolean.

**Description:**

**compile-file** transforms the contents of the file specified by *input-file* into implementation-dependent binary data which are placed in the file specified by *output-file*.

The file to which *input-file* refers should be a source file. *output-file* can be used to specify an output pathname; the actual pathname of the compiled file to which compiled code will be output is computed as if by calling compile-file-pathname.

If *input-file* or *output-file* is a logical pathname, it is translated into a physical pathname as if by calling translate-logical-pathname.

## CLHS: Declaration DYNAMIC-EXTENT

If *verbose* is *true*, **compile-file** prints a message in the form of a comment (i.e., with a leading *semicolon*) to *standard output* indicating what *file* is being *compiled* and other useful information. If *verbose* is *false*, **compile-file** does not print this information.

If *print* is *true*, information about *top level forms* in the file being compiled is printed to *standard output*. Exactly what is printed is *implementation-dependent*, but nevertheless some information is printed. If *print* is *nil*, no information is printed.

The *external-format* specifies the *external file format* to be used when opening the *file*; see the *function open*. **compile-file** and **load** must cooperate in such a way that the resulting *compiled file* can be *loaded* without specifying an *external file format* anew; see the *function load*.

**compile-file** binds *\*readtable\** and *\*package\** to the values they held before processing the file.

*\*compile-file-truename\** is bound by **compile-file** to hold the *truename* of the *pathname* of the file being compiled.

*\*compile-file-pathname\** is bound by **compile-file** to hold a *pathname* denoted by the first argument to **compile-file**, merged against the defaults; that is, (*pathname* (*merge-pathnames input-file*)).

The compiled *functions* contained in the *compiled file* become available for use when the *compiled file* is *loaded* into Lisp. Any function definition that is processed by the compiler, including #'(lambda ...) forms and local function definitions made by **flet**, **labels** and **defun** forms, result in an *object* of type **compiled-function**.

The *primary value* returned by **compile-file**, *output-truename*, is the *truename* of the output file, or *nil* if the file could not be created.

The *secondary value*, *warnings-p*, is *false* if no *conditions* of *type error* or *warning* were detected by the compiler, and *true* otherwise.

The *tertiary value*, *failure-p*, is *false* if no *conditions* of *type error* or *warning* (other than *style-warning*) were detected by the compiler, and *true* otherwise.

For general information about how *files* are processed by the *file compiler*, see [Section 3.2.3 \(File Compilation\)](#).

*Programs* to be compiled by the *file compiler* must only contain *externalizable objects*; for details on such *objects*, see [Section 3.2.4 \(Literal Objects in Compiled Files\)](#). For information on how to extend the set of *externalizable objects*, see the *function make-load-form* and [Section 3.2.4.4 \(Additional Constraints on Externalizable Objects\)](#).

**Examples:** None.

**Affected By:**

*\*error-output\*, \*standard-output\*, \*compile-verbose\*, \*compile-print\**

The computer's file system.

**Exceptional Situations:**

## CLHS: Declaration DYNAMIC-EXTENT

For information about errors detected during the compilation process, see [Section 3.2.5 \(Exceptional Situations in the Compiler\)](#).

An error of [type file-error](#) might be signaled if `(wild-pathname-p input-file)` returns true.

If either the attempt to open the [source file](#) for input or the attempt to open the [compiled file](#) for output fails, an error of [type file-error](#) is signaled.

**See Also:**

[compile](#), [declare](#), [eval-when](#), [pathname](#), [logical-pathname](#), [Section 20.1 \(File System Concepts\)](#), [Section 19.1.2 \(Pathnames as Filenames\)](#)

**Notes:** None.

## Accessor COMPILER-MACRO-FUNCTION

**Syntax:**

`compiler-macro-function name &optional environment => function`

`(setf (compiler-macro-function name &optional environment) new-function)`

**Arguments and Values:**

`name`—a [function name](#).

`environment`—an [environment object](#).

`function, new-function`—a [compiler macro function](#), or [nil](#).

**Description:**

[Accesses](#) the [compiler macro function](#) named `name`, if any, in the `environment`.

A value of [nil](#) denotes the absence of a [compiler macro function](#) named `name`.

**Examples:** None.

**Affected By:** None.

**Exceptional Situations:**

The consequences are undefined if `environment` is [non-nil](#) in a use of `setf` of [compiler-macro-function](#).

**See Also:**

[define-compiler-macro](#), [Section 3.2.2.1 \(Compiler Macros\)](#)

**Notes:** None.

***Function COMPILED-FUNCTION-P*****Syntax:**

**compiled-function-p** *object* => *generalized-boolean*

**Arguments and Values:**

*object* --- an *object*.

*generalized-boolean* --- a *generalized boolean*.

**Description:**

Returns *true* if *object* is of type **compiled-function**; otherwise, returns *false*.

**Examples:**

```
(defun f (x) x) => F
  (compiled-function-p #'f)
=> false
OR=> true
  (compiled-function-p 'f) => false
  (compile 'f) => F
  (compiled-function-p #'f) => true
  (compiled-function-p 'f) => false
  (compiled-function-p (compile nil '(lambda (x) x)))
=> true
  (compiled-function-p #'(lambda (x) x))
=> false
OR=> true
  (compiled-function-p '(lambda (x) x)) => false
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**compile, compile-file, compiled-function**

**Notes:**

```
(compiled-function-p object) == (typep object 'compiled-function)
```

***Function CODE-CHAR*****Syntax:**

**code-char** *code* => *char-p*

**Arguments and Values:**

*code*—a character code.

*char-p*—a character or nil.

**Description:**

Returns a character with the code attribute given by *code*. If no such character exists and one cannot be created, nil is returned.

**Examples:**

```
(code-char 65.) => #\A ;in an implementation using ASCII codes
(code-char (char-code #\Space)) => #\Space ;in any implementation
```

**Affected By:**

The implementation's character encoding.

**Exceptional Situations:** None.

**See Also:**

char-code

**Notes:****Function COERCE****Syntax:**

**coerce** *object result-type => result*

**Arguments and Values:**

*object*—an object.

*result-type*—a type specifier.

*result*—an object, of type result-type except in situations described in Section 12.1.5.3 (Rule of Canonical Representation for Complex Rationals).

**Description:**

Coerces the *object* to type result-type.

If *object* is already of type result-type, the *object* itself is returned, regardless of whether it would have been possible in general to coerce an object of some other type to result-type.

Otherwise, the *object* is *coerced* to type result-type according to the following rules:

**sequence**

If the *result-type* is a *recognizable subtype* of **list**, and the *object* is a **sequence**, then the *result* is a **list** that has the *same elements* as *object*.

If the *result-type* is a *recognizable subtype* of **vector**, and the *object* is a **sequence**, then the *result* is a **vector** that has the *same elements* as *object*. If *result-type* is a specialized *type*, the *result* has an *actual array element type* that is the result of *upgrading* the element type part of that *specialized type*. If no element type is specified, the element type defaults to **t**. If the *implementation* cannot determine the element type, an error is signaled.

**character**

If the *result-type* is **character** and the *object* is a *character designator*, the *result* is the *character* it denotes.

**complex**

If the *result-type* is **complex** and the *object* is a **real**, then the *result* is obtained by constructing a **complex** whose real part is the *object* and whose imaginary part is the result of *coercing* an **integer** zero to the *type* of the *object* (using **coerce**). (If the real part is a **rational**, however, then the result must be represented as a **rational** rather than a **complex**; see [Section 12.1.5.3 \(Rule of Canonical Representation for Complex Rationals\)](#). So, for example, (**coerce** 3 'complex) is permissible, but will return 3, which is not a **complex**.)

**float**

If the *result-type* is any of **float**, **short-float**, **single-float**, **double-float**, **long-float**, and the *object* is a **real**, then the *result* is a **float** of *type result-type* which is equal in sign and magnitude to the *object* to whatever degree of representational precision is permitted by that **float** representation. (If the *result-type* is **float** and *object* is not already a **float**, then the *result* is a **single float**.)

**function**

If the *result-type* is **function**, and *object* is any *function name* that is *fbound* but that is globally defined neither as a *macro name* nor as a *special operator*, then the *result* is the *functional value* of *object*.

If the *result-type* is **function**, and *object* is a *lambda expression*, then the *result* is a *closure* of *object* in the *null lexical environment*.

**t**

Any *object* can be *coerced* to an *object* of *type t*. In this case, the *object* is simply returned.

**Examples:**

```
(coerce '(a b c) 'vector) => #(A B C)
(coerce 'a 'character) => #\A
(coerce 4.56 'complex) => #C(4.56 0.0)
(coerce 4.5s0 'complex) => #C(4.5s0 0.0s0)
(coerce 7/2 'complex) => 7/2
(coerce 0 'short-float) => 0.0s0
(coerce 3.5L0 'float) => 3.5L0
(coerce 7/2 'float) => 3.5
(coerce (cons 1 2) t) => (1 . 2)
```

All the following *forms* should signal an error:

```
(coerce '(a b c) '(vector * 4))
(coerce #(a b c) '(vector * 4))
(coerce '(a b c) '(vector * 2))
(coerce #(a b c) '(vector * 2))
(coerce "foo" '(string 2))
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(coerce #(\a #\b #\c) '(string 2))
(coerce '(0 1) '(simple-bit-vector 3))
```

**Affected By:** None.

**Exceptional Situations:**

If a coercion is not possible, an error of type type-error is signaled.

`(coerce x 'nil)` always signals an error of type type-error.

An error of type error is signaled if the *result-type* is function but *object* is a symbol that is not fbound or if the symbol names a macro or a special operator.

An error of type type-error should be signaled if *result-type* specifies the number of elements and *object* is of a different length.

**See Also:**

rational, floor, char-code, char-int

**Notes:**

Coercions from floats to rationals and from ratios to integers are not provided because of rounding problems.

```
(coerce x 't) == (identity x) == x
```

## Function COMPUTE-RESTARTS

**Syntax:**

**compute-restarts** &optional condition => restarts

**Arguments and Values:**

*condition*—a condition object, or nil.

*restarts*—a list of restarts.

**Description:**

**compute-restarts** uses the dynamic state of the program to compute a list of the restarts which are currently active.

The resulting list is ordered so that the innermost (more-recently established) restarts are nearer the head of the list.

When *condition* is non-nil, only those restarts are considered that are either explicitly associated with that *condition*, or not associated with any condition; that is, the excluded restarts are those that are associated with a non-empty set of conditions of which the given *condition* is not an element. If *condition* is nil, all restarts are considered.

## CLHS: Declaration DYNAMIC-EXTENT

**compute-restarts** returns all *applicable restarts*, including anonymous ones, even if some of them have the same name as others and would therefore not be found by **find-restart** when given a *symbol* argument.

Implementations are permitted, but not required, to return *distinct lists* from repeated calls to **compute-restarts** while in the same dynamic environment. The consequences are undefined if the *list* returned by **compute-restarts** is every modified.

### Examples:

```
; One possible way in which an interactive debugger might present
; restarts to the user.
(defun invoke-a-restart ()
  (let ((restarts (compute-restarts)))
    (do ((i 0 (+ i 1)) (r restarts (cdr r))) ((null r))
        (format t "~&~D: ~A~%" i (car r)))
      (let ((n nil) (k (length restarts)))
        (loop (when (and (typep n 'integer) (>= n 0) (< n k))
                        (return t))
               (format t "~-Option: ")
               (setq n (read))
               (fresh-line))
        (invoke-restart-interactively (nth n restarts)))))

(restart-case (invoke-a-restart)
  (one () 1)
  (two () 2)
  (nil () :report "Who knows?" 'anonymous)
  (one () 'I)
  (two () 'II))
>> 0: ONE
>> 1: TWO
>> 2: Who knows?
>> 3: ONE
>> 4: TWO
>> 5: Return to Lisp Toplevel.
>> Option: 4
=> II

; Note that in addition to user-defined restart points, COMPUTE-RESTARTS
; also returns information about any system-supplied restarts, such as
; the "Return to Lisp Toplevel" restart offered above.
```

**Side Effects:** None.

**Affected By:**

Existing restarts.

**Exceptional Situations:** None.

**See Also:**

**find-restart**, **invoke-restart**, **restart-bind**

**Notes:** None.

## Function COMPLEX

### Syntax:

**complex** *realpart* &*optional imagpart* => *complex*

### Arguments and Values:

*realpart*—a real.

*imagpart*—a real.

*complex*—a rational or a complex.

### Description:

**complex** returns a number whose real part is *realpart* and whose imaginary part is *imagpart*.

If *realpart* is a rational and *imagpart* is the rational number zero, the result of **complex** is *realpart*, a rational. Otherwise, the result is a complex.

If either *realpart* or *imagpart* is a float, the non-float is converted to a float before the complex is created. If *imagpart* is not supplied, the imaginary part is a zero of the same type as *realpart*; i.e., (`(coerce 0 (type-of realpart))`) is effectively used.

Type upgrading implies a movement upwards in the type hierarchy lattice. In the case of complexes, the type-specifier must be a subtype of (upgraded-complex-part-type type-specifier). If type-specifier1 is a subtype of type-specifier2, then (upgraded-complex-element-type 'type-specifier1) must also be a subtype of (upgraded-complex-element-type 'type-specifier2). Two disjoint types can be upgraded into the same thing.

### 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 0.99)
(complex 3/2 0.0) => #C(1.5 0.0)
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

realpart, imagpart, Section 2.4.8.11 (Sharpsign C)

**Notes:** None.

***Function COMPLEXP*****Syntax:**

**complexp** *object* => *generalized-boolean*

**Arguments and Values:**

*object*---an *object*.

*generalized-boolean*---a *generalized boolean*.

**Description:**

Returns *true* if *object* is of *type complex*; otherwise, returns *false*.

**Examples:**

```
(complexp 1.2d2) => false
(complexp #c(5/3 7.2)) => true
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**complex** (*function* and *type*), **typep**

**Notes:**

```
(complexp object) == (typep object 'complex)
```

***Function COMPLEMENT*****Syntax:**

**complement** *function* => *complement-function*

**Arguments and Values:**

*function*---a *function*.

*complement-function*---a *function*.

**Description:**

## CLHS: Declaration DYNAMIC-EXTENT

Returns a function that takes the same arguments as function, and has the same side-effect behavior as function, but returns only a single value: a generalized boolean with the opposite truth value of that which would be returned as the primary value of function. That is, when the function would have returned true as its primary value the complement-function returns false, and when the function would have returned false as its primary value the complement-function returns true.

### Examples:

```
(funcall (complement #'zerop) 1) => true
(funcall (complement #'characterp) #\A) => false
(funcall (complement #'member) 'a '(a b c)) => false
(funcall (complement #'member) 'd '(a b c)) => true
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

### See Also:

**not**

### Notes:

```
(complement x) == #'(lambda (&rest arguments) (not (apply x arguments)))
```

In Common Lisp, functions with names like "xxx-if-not" are related to functions with names like "xxx-if" in that

```
(xxx-if-not f . arguments) == (xxx-if (complement f) . arguments)
```

For example,

```
(find-if-not #'zerop '(0 0 3)) ==
(find-if (complement #'zerop) '(0 0 3)) => 3
```

Note that since the "xxx-if-not" functions and the :test-not arguments have been deprecated, uses of "xxx-if" functions or :test arguments with complement are preferred.

## Standard Generic Function COMPUTE-APPLICABLE-METHODS

### Syntax:

**compute-applicable-methods** *generic-function function-arguments => methods*

### Method Signatures:

**compute-applicable-methods** (*generic-function standard-generic-function*)

### Arguments and Values:

*generic-function*---a generic function.

*function-arguments*---a list of arguments for the *generic-function*.

*methods*---a list of method objects.

#### Description:

Given a *generic-function* and a set of *function-arguments*, the function compute-applicable-methods returns the set of methods that are applicable for those arguments sorted according to precedence order. See Section 7.6.6 (Method Selection and Combination).

#### Affected By:

defmethod

**Exceptional Situations:** None.

#### See Also:

**Notes:** None.

## **Function CONCATENATED-STREAM-STREAMS**

#### Syntax:

**concatenated-stream-streams** *concatenated-stream => streams*

#### Arguments and Values:

*concatenated-stream* -- a concatenated stream.

*streams*---a list of input streams.

#### Description:

Returns a list of input streams that constitute the ordered set of streams the *concatenated-stream* still has to read from, starting with the current one it is reading from. The list may be *empty* if no more streams remain to be read.

The consequences are undefined if the list structure of the *streams* is ever modified.

**Examples:** None.

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:** None.

**Notes:** None.

## Function CONCATENATE

**Syntax:**

`concatenate result-type &rest sequences => result-sequence`

**Arguments and Values:**

*result-type*—a sequence type specifier.

*sequences*—a sequence.

*result-sequence*—a proper sequence of type *result-type*.

**Description:**

**concatenate** returns a sequence that contains all the individual elements of all the *sequences* in the order that they are supplied. The sequence is of type *result-type*, which must be a subtype of type sequence.

All of the *sequences* are copied from; the result does not share any structure with any of the *sequences*. Therefore, if only one *sequence* is provided and it is of type *result-type*, **concatenate** is required to copy *sequence* rather than simply returning it.

It is an error if any element of the *sequences* cannot be an element of the sequence result. If the *result-type* is a subtype of list, the result will be a list.

If the *result-type* is a subtype of vector, then if the implementation can determine the element type specified for the *result-type*, the element type of the resulting array is the result of *upgrading* that element type; or, if the implementation can determine that the element type is unspecified (or \*), the element type of the resulting array is t; otherwise, an error is signaled.

**Examples:**

```
(concatenate 'string "all" " " "together" " " "now") => "all together now"
(concatenate 'list "ABC" '(d e f) #(1 2 3) #*1011)
=> (#\A #\B #\C D E F 1 2 3 1 0 1 1)
(concatenate 'list) => NIL

(concatenate '(vector * 2) "a" "bc") should signal an error
```

**Affected By:** None.

**Exceptional Situations:**

An error is signaled if the *result-type* is neither a recognizable subtype of list, nor a recognizable subtype of vector.

An error of type type-error should be signaled if *result-type* specifies the number of elements and the sum of *sequences* is different from that number.

**See Also:****append****Notes:** None.***Function CONJUGATE*****Syntax:****conjugate** *number* => *conjugate***Arguments and Values:***number*—a number.*conjugate*—a number.**Description:**Returns the complex conjugate of *number*. The conjugate of a *real* number is itself.**Examples:**

```
(conjugate #c(0 -1)) => #C(0 1)
(conjugate #c(1 1)) => #C(1 -1)
(conjugate 1.5) => 1.5
(conjugate #C(3/5 4/5)) => #C(3/5 -4/5)
(conjugate #C(0.0D0 -1.0D0)) => #C(0.0D0 1.0D0)
(conjugate 3.7) => 3.7
```

**Side Effects:** None.**Affected By:** None.**Exceptional Situations:** None.**See Also:** None.**Notes:**For a complex number *z*,(conjugate *z*) == (complex (realpart *z*) (- (imagpart *z*)))***Function CONS*****Syntax:****cons** *object-1* *object-2* => *cons*

**Arguments and Values:**

*object-1*—an object.

*object-2*—an object.

*cons*—a cons.

**Description:**

Creates a fresh cons, the car of which is *object-1* and the cdr of which is *object-2*.

**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) => (A . B)
(cons 'a (cons 'b (cons 'c '()))) => (A B C)
(cons 'a '(b c d)) => (A B C D)
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

list

**Notes:*****Function CONSTANTLY*****Syntax:**

**constantly** *value* => *function*

**Arguments and Values:**

*value*—an object.

*function*—a function.

**Description:**

**constantly** returns a function that accepts any number of arguments, that has no side-effects, and that always returns *value*.

**Examples:**

```
(mapcar (constantly 3) '(a b c d)) => (3 3 3 3)
(defmacro with-vars (vars &body forms)
  `((lambda ,vars ,@forms) ,@(mapcar (constantly nil) vars)))
=> WITH-VARS
(macroexpand '(with-vars (a b) (setq a 3 b (* a a)) (list a b)))
=> ((LAMBDA (A B) (SETQ A 3 B (* A A)) (LIST A B)) NIL NIL), true
```

**Affected By:** None.**Exceptional Situations:** None.**See Also:****identity****Notes:****constantly** could be defined by:

```
(defun constantly (object)
  #'(lambda (&rest arguments) object))
```

**Function CONSP****Syntax:****consp** *object* => *generalized-boolean***Arguments and Values:***object*---an *object*.*generalized-boolean*---a *generalized boolean*.**Description:**Returns *true* if *object* is of *type cons*; otherwise, returns *false*.**Examples:**

```
(consp nil) => false
(consp (cons 1 2)) => true
```

The *empty list* is not a *cons*, so

```
(consp '()) == (consp 'nil) => false
```

**Side Effects:** None.**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**listp**

**Notes:**

```
(consp object) == (typep object 'cons) == (not (typep object 'atom)) == (typep object '(not
```

## Function CONSTANTP

**Syntax:**

**constantp** *form* &optional *environment* => *generalized-boolean*

**Arguments and Values:**

*form*---a *form*.

*environment*---an *environment object*. The default is **nil**.

*generalized-boolean*---a *generalized boolean*.

**Description:**

Returns *true* if *form* can be determined by the *implementation* to be a *constant form* in the indicated *environment*; otherwise, it returns *false* indicating either that the *form* is not a *constant form* or that it cannot be determined whether or not *form* is a *constant form*.

The following kinds of *forms* are considered *constant forms*:

\* *Self-evaluating objects* (such as *numbers*, *characters*, and the various kinds of *arrays*) are always considered *constant forms* and must be recognized as such by **constantp**.

\* *Constant variables*, such as *keywords*, symbols defined by Common Lisp as constant (such as **nil**, **t**, and **pi**), and symbols declared as constant by the user in the indicated environment using **defconstant** are always considered *constant forms* and must be recognized as such by **constantp**.

\* *quote forms* are always considered *constant forms* and must be recognized as such by **constantp**.

\* An *implementation* is permitted, but not required, to detect additional *constant forms*. If it does, it is also permitted, but not required, to make use of information in the environment. Examples of *constant forms* for which **constantp** might or might not return *true* are: (*sqrt pi*), (*+ 3 2*), (*length '(a b c)*), and (*let ((x 7)) (zerop x)*).

If an *implementation* chooses to make use of the *environment* information, such actions as expanding *macros* or performing function inlining are permitted to be used, but not required; however, expanding *compiler macros* is not permitted.

**Examples:**

```
(constantp 1) => true
(constantp 'temp) => false
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(constantp "temp") => true
(defconstant this-is-a-constant 'never-changing) => THIS-IS-A-CONSTANT
(constantp 'this-is-a-constant) => true
(constantp "temp") => true
(setq a 6) => 6
(constantp a) => true
(constantp '(sin pi)) => implementation-dependent
(constantp '(car '(x))) => implementation-dependent
(constantp '(eql x x)) => implementation-dependent
(constantp '(typep x 'nil)) => implementation-dependent
(constantp '(typep x 't)) => implementation-dependent
(constantp '(values this-is-a-constant)) => implementation-dependent
(constantp '(values 'x 'y)) => implementation-dependent
(constantp '(let ((a '(a b c))) (+ (length a) 6))) => implementation-dependent
```

**Side Effects:** None.

**Affected By:**

The state of the global environment (e.g., which symbols have been declared to be the names of constant variables).

**Exceptional Situations:** None.

**See Also:**

**defconstant**

**Notes:** None.

## Function COUNT, COUNT-IF, COUNT-IF-NOT

**Syntax:**

**count** *item sequence &key from-end start end key test test-not => n*

**count-if** *predicate sequence &key from-end start end key => n*

**count-if-not** *predicate sequence &key from-end start end key => n*

**Arguments and Values:**

*item*—an object.

*sequence*—a proper sequence.

*predicate*—a designator for a function of one argument that returns a generalized boolean.

*from-end*—a generalized boolean. The default is false.

*test*—a designator for a function of two arguments that returns a generalized boolean.

## CLHS: Declaration DYNAMIC-EXTENT

*test-not*—a designator for a function of two arguments that returns a generalized boolean.

*start, end*—bounding index designators of sequence. The defaults for *start* and *end* are 0 and nil, respectively.

*key*—a designator for a function of one argument, or nil.

*n*—a non-negative integer less than or equal to the length of sequence.

### Description:

count, count-if, and count-if-not count and return the number of elements in the sequence bounded by *start* and *end* that satisfy the test.

The *from-end* has no direct effect on the result. However, if *from-end* is true, the elements of sequence will be supplied as arguments to the *test*, *test-not*, and *key* in reverse order, which may change the side-effects, if any, of those functions.

### Examples:

```
(count #\a "how many A's are there in here?") => 2
(count-if-not #'oddp '((1) (2) (3) (4)) :key #'car) => 2
(count-if #'upper-case-p "The Crying of Lot 49" :start 4) => 2
```

**Side Effects:** None.

**Affected By:** None.

### Exceptional Situations:

Should be prepared to signal an error of type type-error if sequence is not a proper sequence.

### See Also:

### Notes:

The :test-not argument is deprecated.

The function count-if-not is deprecated.

## Function COPY-ALIST

### Syntax:

**copy-alist** *alist* => *new-alist*

### Arguments and Values:

*alist*—an association list.

*new-alist*—an association list.

**Description:**

**copy-alist** returns a copy of *alist*.

The list structure of *alist* is copied, and the elements of *alist* which are conses are also copied (as conses only). Any other objects which are referred to, whether directly or indirectly, by the *alist* continue to be shared.

**Examples:**

```
(defparameter *alist* (acons 1 "one" (acons 2 "two" '())))
*alist* => ((1 . "one") (2 . "two"))
(defparameter *list-copy* (copy-list *alist*))
*list-copy* => ((1 . "one") (2 . "two"))
(defparameter *alist-copy* (copy-alist *alist*))
*alist-copy* => ((1 . "one") (2 . "two"))
(setf (cdr (assoc 2 *alist-copy*)) "deux") => "deux"
*alist-copy* => ((1 . "one") (2 . "deux"))
*alist* => ((1 . "one") (2 . "two"))
(setf (cdr (assoc 1 *list-copy*)) "uno") => "uno"
*list-copy* => ((1 . "uno") (2 . "two"))
*alist* => ((1 . "uno") (2 . "two"))
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**copy-list**

**Notes:** None.

## Function COPY-LIST

**Syntax:**

**copy-list** *list* => *copy*

**Arguments and Values:**

*list*—a proper list or a dotted list.

*copy*—a list.

**Description:**

Returns a copy of *list*. If *list* is a dotted list, the resulting *list* will also be a dotted list.

Only the list structure of *list* is copied; the elements of the resulting list are the same as the corresponding elements of the given *list*.

**Examples:**

```
(setq lst (list 1 (list 2 3))) => (1 (2 3))
(setq slst lst) => (1 (2 3))
(setq clst (copy-list lst)) => (1 (2 3))
(eq slst lst) => true
(eq clst lst) => false
(equal clst lst) => true
(rplaca lst "one") => ("one" (2 3))
slst => ("one" (2 3))
clst => (1 (2 3))
(setqf (caaddr lst) "two") => "two"
lst => ("one" ("two" 3))
slst => ("one" ("two" 3))
clst => (1 ("two" 3))
```

**Side Effects:** None.**Affected By:** None.**Exceptional Situations:**

The consequences are undefined if *list* is a circular list.

**See Also:**

[copy-alist](#), [copy-seq](#), [copy-tree](#)

**Notes:**

The copy created is equal to *list*, but not eq.

**Function COPY-PPRINT-DISPATCH****Syntax:**

**copy-pprint-dispatch** &*optional table* => *new-table*

**Arguments and Values:**

*table*—a pprint dispatch table, or nil.

*new-table*—a fresh pprint dispatch table.

**Description:**

Creates and returns a copy of the specified *table*, or of the value of \*print-pprint-dispatch\* if no *table* is specified, or of the initial value of \*print-pprint-dispatch\* if nil is specified.

**Examples:** None.**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:**

Should signal an error of type type-error if *table* is not a pprint dispatch table.

**See Also:** None.

**Notes:** None.

## **Function COPY-READTABLE**

**Syntax:**

**copy-readtable** &optional *from-readtable* *to-readtable* => *readtable*

**Arguments and Values:**

*from-readtable*—a readtable designator. The default is the current readtable.

*to-readtable*—a readtable or nil. The default is nil.

*readtable*—the *to-readtable* if it is non-nil, or else a fresh readtable.

**Description:**

**copy-readtable** copies *from-readtable*.

If *to-readtable* is nil, a new readtable is created and returned. Otherwise the readtable specified by *to-readtable* is modified and returned.

**copy-readtable** copies the setting of readtable-case.

**Examples:**

```
(setq zvar 123) => 123
(set-syntax-from-char #\z #\` (setq table2 (copy-readtable))) => T
zvar => 123
(copy-readtable table2 *readtable*) => #<READTABLE 614000277>
zvar => VAR
(setq *readtable* (copy-readtable)) => #<READTABLE 46210223>
zvar => VAR
(setq *readtable* (copy-readtable nil)) => #<READTABLE 46302670>
zvar => 123
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

readtable, \*readtable\*

**Notes:**

```
(setq *readtable* (copy-readtable nil))
```

restores the input syntax to standard Common Lisp syntax, even if the *initial readtable* has been clobbered (assuming it is not so badly clobbered that you cannot type in the above expression).

On the other hand,

```
(setq *readtable* (copy-readtable))
```

replaces the current *readtable* with a copy of itself. This is useful if you want to save a copy of a readtable for later use, protected from alteration in the meantime. It is also useful if you want to locally bind the readtable to a copy of itself, as in:

```
(let ((*readtable* (copy-readtable))) ...)
```

## **Function COPY-SEQ**

**Syntax:**

**copy-seq** *sequence* => *copied-sequence*

**Arguments and Values:**

*sequence*—a *proper sequence*.

*copied-sequence*—a *proper sequence*.

**Description:**

Creates a copy of *sequence*. The *elements* of the new *sequence* are the *same* as the corresponding *elements* of the given *sequence*.

If *sequence* is a *vector*, the result is a *fresh simple array* of *rank* one that has the same *actual array element type* as *sequence*. If *sequence* is a *list*, the result is a *fresh list*.

**Examples:**

```
(setq str "a string") => "a string"
(equalp str (copy-seq str)) => true
(eql str (copy-seq str)) => false
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:**

Should be prepared to signal an error of *type type-error* if *sequence* is not a *proper sequence*.

**See Also:**

Function COPY-SEQ

**copy-list****Notes:**

From a functional standpoint,

```
(copy-seq x) == (subseq x 0)
```

However, the programmer intent is typically very different in these two cases.

***Function COPY-STRUCTURE*****Syntax:**

**copy-structure** *structure* => *copy*

**Arguments and Values:**

*structure*—a structure.

*copy*—a copy of the *structure*.

**Description:**

Returns a copy[6] of the *structure*.

Only the *structure* itself is copied; not the values of the slots.

**Examples:** None.**Side Effects:** None.**Affected By:** None.**Exceptional Situations:** None.**See Also:**

the :copier option to **defstruct**

**Notes:**

The *copy* is the same as the given *structure* under **equalp**, but not under **equal**.

***Function COPY-SYMBOL*****Syntax:**

**copy-symbol** *symbol* &*optional copy-properties* => *new-symbol*

**Arguments and Values:**

*symbol*—a symbol.

*copy-properties*—a generalized boolean. The default is false.

*new-symbol*—a fresh, uninterned symbol.

**Description:**

**copy-symbol** returns a fresh, uninterned symbol, the name of which is string= to and possibly the same as the name of the given *symbol*.

If *copy-properties* is false, the *new-symbol* is neither bound nor fbound and has a null property list. If *copy-properties* is true, then the initial value of *new-symbol* is the value of *symbol*, the initial function definition of *new-symbol* is the functional value of *symbol*, and the property list of *new-symbol* is a copy[2] of the property list of *symbol*.

**Examples:**

```
(setq fred 'fred-smith) => FRED-SMITH
(setf (symbol-value fred) 3) => 3
(setq fred-clone-1a (copy-symbol fred nil)) => #:FRED-SMITH
(setq fred-clone-1b (copy-symbol fred nil)) => #:FRED-SMITH
(setq fred-clone-2a (copy-symbol fred t)) => #:FRED-SMITH
(setq fred-clone-2b (copy-symbol fred t)) => #:FRED-SMITH
(eq fred fred-clone-1a) => false
(eq fred-clone-1a fred-clone-1b) => false
(eq fred-clone-2a fred-clone-2b) => false
(eq fred-clone-1a fred-clone-2a) => false
(symbol-value fred) => 3
(boundp fred-clone-1a) => false
(symbol-value fred-clone-2a) => 3
(setf (symbol-value fred-clone-2a) 4) => 4
(symbol-value fred) => 3
(symbol-value fred-clone-2a) => 4
(symbol-value fred-clone-2b) => 3
(boundp fred-clone-1a) => false
(setf (symbol-function fred) #'(lambda (x) x)) => #<FUNCTION anonymous>
(fboundp fred) => true
(fboundp fred-clone-1a) => false
(fboundp fred-clone-2a) => false
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:**

Should signal an error of type type-error if *symbol* is not a symbol.

**See Also:**

**make-symbol**

**Notes:**

Implementors are encouraged not to copy the string which is the symbol's name unnecessarily. Unless there is a good reason to do so, the normal implementation strategy is for the new-symbol's name to be identical to the given symbol's name.

**Function COPY-TREE****Syntax:**

**copy-tree** *tree* => *new-tree*

**Arguments and Values:**

*tree*—a tree.

*new-tree*—a tree.

**Description:**

Creates a copy of a tree of conses.

If *tree* is not a cons, it is returned; otherwise, the result is a new cons of the results of calling **copy-tree** on the car and cdr of *tree*. In other words, all conses in the tree represented by *tree* are copied recursively, stopping only when non-conses are encountered.

**copy-tree** does not preserve circularities and the sharing of substructure.

**Examples:**

```
(setq object (list (cons 1 "one")
                   (cons 2 (list 'a 'b 'c))))
=> ((1 . "one") (2 A B C))
(setq object-too object) => ((1 . "one") (2 A B C))
(setq copy-as-list (copy-list object))
(setq copy-as-alist (copy-alist object))
(setq copy-as-tree (copy-tree object))
(eq object object-too) => true
(eq copy-as-tree object) => false
(eql copy-as-tree object) => false
(equal copy-as-tree object) => true
(setq (first (cdr (second object))) "a"
      (car (second object)) "two"
      (car object) '(one . 1)) => (ONE . 1)
object => ((ONE . 1) ("two" "a" B C))
object-too => ((ONE . 1) ("two" "a" B C))
copy-as-list => ((1 . "one") ("two" "a" B C))
copy-as-alist => ((1 . "one") (2 "a" B C))
copy-as-tree => ((1 . "one") (2 A B C))
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**tree-equal**

**Notes:** None.

**Function DECODE-FLOAT, SCALE-FLOAT, FLOAT-RADIX, FLOAT-SIGN,  
FLOAT-DIGITS, FLOAT-PRECISION, INTEGER-DECODE-FLOAT**

**Syntax:**

**decode-float** *float* => *significand, exponent, sign*

**scale-float** *float integer* => *scaled-float*

**float-radix** *float* => *float-radix*

**float-sign** *float-1 &optional float-2* => *signed-float*

**float-digits** *float* => *digits1*

**float-precision** *float* => *digits2*

**integer-decode-float** *float* => *significand, exponent, integer-sign*

**Arguments and Values:**

*digits1*---a non-negative integer.

*digits2*---a non-negative integer.

*exponent*---an integer.

*float*---a float.

*float-1*---a float.

*float-2*---a float.

*float-radix*---an integer.

*integer*---a non-negative integer.

## CLHS: Declaration DYNAMIC-EXTENT

*integer-sign*—the *integer* -1, or the *integer* 1.

*scaled-float*—a *float*.

*sign*—A *float* of the same *type* as *float* but numerically equal to 1.0 or -1.0.

*signed-float*—a *float*.

*significand*—a *float*.

### Description:

**decode-float** computes three values that characterize *float*. The first value is of the same *type* as *float* and represents the significand. The second value represents the exponent to which the radix (notated in this description by *b*) must be raised to obtain the value that, when multiplied with the first result, produces the absolute value of *float*. If *float* is zero, any *integer* value may be returned, provided that the identity shown for **scale-float** holds. The third value is of the same *type* as *float* and is 1.0 if *float* is greater than or equal to zero or -1.0 otherwise.

**decode-float** divides *float* by an integral power of *b* so as to bring its value between 1/*b* (inclusive) and 1 (exclusive), and returns the quotient as the first value. If *float* is zero, however, the result equals the absolute value of *float* (that is, if there is a negative zero, its significand is considered to be a positive zero).

**scale-float** returns (\* *float* (expt (float *b* *float*) *integer*)), where *b* is the radix of the floating-point representation. *float* is not necessarily between 1/*b* and 1.

**float-radix** returns the radix of *float*.

**float-sign** returns a number *z* such that *z* and *float-1* have the same sign and also such that *z* and *float-2* have the same absolute value. If *float-2* is not supplied, its value is (float 1 *float-1*). If an implementation has distinct representations for negative zero and positive zero, then (float-sign -0.0) => -1.0.

**float-digits** returns the number of radix *b* digits used in the representation of *float* (including any implicit digits, such as a "hidden bit").

**float-precision** returns the number of significant radix *b* digits present in *float*; if *float* is a *float* zero, then the result is an *integer* zero.

For *normalized floats*, the results of **float-digits** and **float-precision** are the same, but the precision is less than the number of representation digits for a *denormalized* or zero number.

**integer-decode-float** computes three values that characterize *float* – the significand scaled so as to be an *integer*, and the same last two values that are returned by **decode-float**. If *float* is zero, **integer-decode-float** returns zero as the first value. The second value bears the same relationship to the first value as for **decode-float**:

```
(multiple-value-bind (signif expon sign)
    (integer-decode-float f)
  (scale-float (float signif f) expon)) == (abs f)
```

## CLHS: Declaration DYNAMIC-EXTENT

### Examples:

```
; ; Note that since the purpose of this functionality is to expose
; ; details of the implementation, all of these examples are necessarily
; ; very implementation-dependent. Results may vary widely.
; ; Values shown here are chosen consistently from one particular implementation.
(decode-float .5) => 0.5, 0, 1.0
(decode-float 1.0) => 0.5, 1, 1.0
(scale-float 1.0 1) => 2.0
(scale-float 10.01 -2) => 2.5025
(scale-float 23.0 0) => 23.0
(float-radix 1.0) => 2
(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
(float-digits 1.0) => 24
(float-precision 1.0) => 24
(float-precision least-positive-single-float) => 1
(integer-decode-float 1.0) => 8388608, -23, 1
```

**Side Effects:** None.

### Affected By:

The implementation's representation for *floats*.

### Exceptional Situations:

The functions **decode-float**, **float-radix**, **float-digits**, **float-precision**, and **integer-decode-float** should signal an error if their only argument is not a *float*.

The *function scale-float* should signal an error if its first argument is not a *float* or if its second argument is not an *integer*.

The *function float-sign* should signal an error if its first argument is not a *float* or if its second argument is supplied but is not a *float*.

**See Also:** None.

### Notes:

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 *float*.

```
(multiple-value-bind (signif expon sign)
    (decode-float f)
    (scale-float signif expon))
== (abs f)
```

and

```
(multiple-value-bind (signif expon sign)
```

Function DECODE-FLOAT, SCALE-FLOAT, FLOAT-RADIX, FLOAT-SIGN, FLOAT-DIGITS, FLOAT-PR

## CLHS: Declaration DYNAMIC-EXTENT

```
(decode-float f)
(* (scale-float signif expon) sign))
== f
```

## Function DECODE-UNIVERSAL-TIME

### Syntax:

**decode-universal-time** *universal-time* &*optional time-zone*

=> *second, minute, hour, date, month, year, day, daylight-p, zone*

### Arguments and Values:

*universal-time*---a universal time.

*time-zone*---a time zone.

*second, minute, hour, date, month, year, day, daylight-p, zone*---a decoded time.

### Description:

Returns the decoded time represented by the given universal time.

If *time-zone* is not supplied, it defaults to the current time zone adjusted for daylight saving time. If *time-zone* is supplied, daylight saving time information is ignored. The daylight saving time flag is nil if *time-zone* is supplied.

### Examples:

```
(decode-universal-time 0 0) => 0, 0, 0, 1, 1, 1900, 0, false, 0
;; The next two examples assume Eastern Daylight Time.
(decode-universal-time 2414296800 5) => 0, 0, 1, 4, 7, 1976, 6, false, 5
(decode-universal-time 2414293200) => 0, 0, 1, 4, 7, 1976, 6, true, 5

;; This example assumes that the time zone is Eastern Daylight Time
;; (and that the time zone is constant throughout the example).
(let* ((here (nth 8 (multiple-value-list (get-decoded-time)))) ;Time zone
       (recently (get-universal-time))
       (a (nthcdr 7 (multiple-value-list (decode-universal-time recently))))
       (b (nthcdr 7 (multiple-value-list (decode-universal-time recently here)))))

  (list a b (equal a b))) => ((T 5) (NIL 5) NIL)
```

### Affected By:

Implementation-dependent mechanisms for calculating when or if daylight savings time is in effect for any given session.

**Exceptional Situations:** None.

### See Also:

[encode-universal-time](#), [get-universal-time](#), Section 25.1.4 (Time)

**Notes:** None.

## Function DELETE-FILE

### Syntax:

**delete-file** *filespec => t*

### Arguments and Values:

*filespec*—a pathname designator.

### Description:

Deletes the *file* specified by *filespec*.

If the *filespec designator* is an open *stream*, then *filespec* and the file associated with it are affected (if the file system permits), in which case *filespec* might be closed immediately, and the deletion might be immediate or delayed until *filespec* is explicitly closed, depending on the requirements of the file system.

It is implementation-dependent whether an attempt to delete a nonexistent file is considered to be successful.

**delete-file** returns *true* if it succeeds, or signals an error of type file-error if it does not.

The consequences are undefined if *filespec* has a wild component, or if *filespec* has a nil component and the file system does not permit a nil component.

### Examples:

```
(with-open-file (s "delete-me.text" :direction :output :if-exists :error))
=> NIL
(setq p (probe-file "delete-me.text")) => #P"R:>fred>delete-me.text.1"
(delete-file p) => T
(probe-file "delete-me.text") => false
(with-open-file (s "delete-me.text" :direction :output :if-exists :error)
  (delete-file s))
=> T
(probe-file "delete-me.text") => false
```

**Affected By:** None.

### Exceptional Situations:

If the deletion operation is not successful, an error of type file-error is signaled.

An error of type file-error might be signaled if *filespec* is wild.

### See Also:

[pathname, logical-pathname, Section 20.1 \(File System Concepts\), Section 19.1.2 \(Pathnames as Filenames\)](#)

**Notes:** None.

## Function DELETE-PACKAGE

**Syntax:**

**delete-package** *package* => *generalized-boolean*

**Arguments and Values:**

*package*—[a package designator](#).

*generalized-boolean*—[a generalized boolean](#).

**Description:**

**delete-package** deletes *package* from all package system data structures. If the operation is successful, **delete-package** returns true, otherwise **nil**. The effect of **delete-package** is that the name and nicknames of *package* cease to be recognized package names. The package *object* is still a [package](#) (i.e., **packagep** is *true* of it) but **package-name** returns **nil**. The consequences of deleting the COMMON-LISP package or the KEYWORD package are undefined. The consequences of invoking any other package operation on *package* once it has been deleted are unspecified. In particular, the consequences of invoking **find-symbol**, **intern** and other functions that look for a symbol name in a [package](#) are unspecified if they are called with **\*package\*** bound to the deleted *package* or with the deleted *package* as an argument.

If *package* is a [package object](#) that has already been deleted, **delete-package** immediately returns **nil**.

After this operation completes, the *home package* of any [symbol](#) whose *home package* had previously been *package* is [implementation-dependent](#). Except for this, [symbols accessible](#) in *package* are not modified in any other way; [symbols](#) whose *home package* is not *package* remain unchanged.

**Examples:**

```
(setq *foo-package* (make-package "FOO" :use nil))
(setq *foo-symbol* (intern "FOO" *foo-package*))
(export *foo-symbol* *foo-package*)

(setq *bar-package* (make-package "BAR" :use '("FOO")))
(setq *bar-symbol* (intern "BAR" *bar-package*))
(export *foo-symbol* *bar-package*)
(export *bar-symbol* *bar-package*)

(setq *baz-package* (make-package "BAZ" :use '("BAR")))

(symbol-package *foo-symbol*) => #<PACKAGE "FOO">
(symbol-package *bar-symbol*) => #<PACKAGE "BAR">

(prin1-to-string *foo-symbol*) => "FOO:FOO"
(prin1-to-string *bar-symbol*) => "BAR:BAR"

(find-symbol "FOO" *bar-package*) => FOO:FOO, :EXTERNAL
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(find-symbol "FOO" *baz-package*) => FOO:FOO, :INHERITED
(find-symbol "BAR" *baz-package*) => BAR:BAR, :INHERITED

(packagep *foo-package*) => true
(packagep *bar-package*) => true
(packagep *baz-package*) => true

(package-name *foo-package*) => "FOO"
(package-name *bar-package*) => "BAR"
(package-name *baz-package*) => "BAZ"

(package-use-list *foo-package*) => ()
(package-use-list *bar-package*) => (#<PACKAGE "FOO">)
(package-use-list *baz-package*) => (#<PACKAGE "BAR">)

(package-used-by-list *foo-package*) => (#<PACKAGE "BAR">)
(package-used-by-list *bar-package*) => (#<PACKAGE "BAZ">)
(package-used-by-list *baz-package*) => ()

(delete-package *bar-package*)
>> Error: Package BAZ uses package BAR.
>> If continued, BAZ will be made to unuse-package BAR,
>> and then BAR will be deleted.
>> Type :CONTINUE to continue.
>> Debug> :CONTINUE
=> T

(symbol-package *foo-symbol*) => #<PACKAGE "FOO">
(symbol-package *bar-symbol*) is unspecified

(prin1-to-string *foo-symbol*) => "FOO:FOO"
(prin1-to-string *bar-symbol*) is unspecified

(find-symbol "FOO" *bar-package*) is unspecified

(find-symbol "FOO" *baz-package*) => NIL, NIL
(find-symbol "BAR" *baz-package*) => NIL, NIL

(packagep *foo-package*) => T
(packagep *bar-package*) => T
(packagep *baz-package*) => T

(package-name *foo-package*) => "FOO"
(package-name *bar-package*) => NIL
(package-name *baz-package*) => "BAZ"

(package-use-list *foo-package*) => ()
(package-use-list *bar-package*) is unspecified
(package-use-list *baz-package*) => ()

(package-used-by-list *foo-package*) => ()
(package-used-by-list *bar-package*) is unspecified
(package-used-by-list *baz-package*) => ()
```

**Affected By:** None.

**Exceptional Situations:**

## CLHS: Declaration DYNAMIC-EXTENT

If the *package designator* is a *name* that does not currently name a *package*, a *correctable* error of *type package-error* is signaled. If correction is attempted, no deletion action is attempted; instead, *delete-package* immediately returns *nil*.

If *package* is used by other *packages*, a *correctable* error of *type package-error* is signaled. If correction is attempted, *unuse-package* is effectively called to remove any dependencies, causing *package's external symbols* to cease being *accessible* to those *packages* that use *package*. *delete-package* then deletes *package* just as it would have had there been no *packages* that used it.

**See Also:**

**unuse-package**

**Notes:** None.

## Function DEPOSIT-FIELD

**Syntax:**

**deposit-field** *newbyte bytespec integer => result-integer*

**Arguments and Values:**

*newbyte*—an *integer*.

*bytespec*—a *byte specifier*.

*integer*—an *integer*.

*result-integer*—an *integer*.

**Description:**

Replaces a field of bits within *integer*; specifically, returns an *integer* that contains the bits of *newbyte* within the *byte* specified by *bytespec*, and elsewhere contains the bits of *integer*.

**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
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**byte, dpb**

**Notes:**

```
(logbitp j (deposit-field m (byte s p) n))
== (if (and (>= j p) (< j (+ p s)))
      (logbitp j m)
      (logbitp j n))
```

**deposit-field** is to **mask-field** as **dpb** is to **ldb**.

## **Standard Generic Function DESCRIBE-OBJECT**

**Syntax:**

**describe-object** *object stream => implementation-dependent*

**Method Signatures:**

**describe-object** (*object standard-object*) *stream*

**Arguments and Values:**

*object*—an object.

*stream*—a stream.

**Description:**

The generic function **describe-object** prints a description of *object* to a *stream*. **describe-object** is called by **describe**; it must not be called by the user.

Each implementation is required to provide a method on the class standard-object and methods on enough other classes so as to ensure that there is always an applicable method. Implementations are free to add methods for other classes. Users can write methods for **describe-object** for their own classes if they do not wish to inherit an implementation-supplied method.

Methods on **describe-object** can recursively call **describe**. Indentation, depth limits, and circularity detection are all taken care of automatically, provided that each method handles exactly one level of structure and calls **describe** recursively if there are more structural levels. The consequences are undefined if this rule is not obeyed.

In some implementations the *stream* argument passed to a **describe-object** method is not the original *stream*, but is an intermediate stream that implements parts of **describe**. Methods should therefore not depend on the identity of this stream.

**Examples:**

```
(defclass spaceship ()
  ((captain :initarg :captain :accessor spaceship-captain)
   (serial# :initarg :serial-number :accessor spaceship-serial-number)))

(defclass federation-starship (spaceship) ())
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(defmethod describe-object ((s spaceship) stream)
  (with-slots (captain serial#) s
    (format stream "~&~S is a spaceship of type ~S,~"
           "~%with ~A at the helm ~"
           "and with serial number ~D.~%"'
           s (type-of s) captain serial#)))

(make-instance 'federation-starship
               :captain "Rachel Garrett"
               :serial-number "NCC-1701-C")
=> #<FEDERATION-STARSHIP 26312465>

(describe *)
>> #<FEDERATION-STARSHIP 26312465> is a spaceship of type FEDERATION-STARSHIP,
>> with Rachel Garrett at the helm and with serial number NCC-1701-C.
=> <no values>
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**describe**

**Notes:**

The same implementation techniques that are applicable to **print-object** are applicable to **describe-object**.

The reason for making the return values for **describe-object** unspecified is to avoid forcing users to include explicit (**values**) in all of their **methods**. **describe** takes care of that.

## Function DESCRIBE

**Syntax:**

**describe** *object* &*optional stream* => <no values>

**Arguments and Values:**

*object*---an *object*.

*stream*---an *output stream designator*. The default is *standard output*.

**Description:**

**describe** displays information about *object* to *stream*.

For example, **describe** of a *symbol* might show the *symbol*'s value, its definition, and each of its properties. **describe** of a *float* might show the number's internal representation in a way that is useful for tracking down round-off errors. In all cases, however, the nature and format of the output of **describe** is *implementation-dependent*.

**describe** can describe something that it finds inside the *object*; in such cases, a notational device such as increased indentation or positioning in a table is typically used in order to visually distinguish such recursive descriptions from descriptions of the argument *object*.

The actual act of describing the object is implemented by **describe-object**. **describe** exists as an interface primarily to manage argument defaulting (including conversion of arguments **t** and **nil** into *stream objects*) and to inhibit any return values from **describe-object**.

**describe** is not intended to be an interactive function. In a *conforming implementation*, **describe** must not, by default, prompt for user input. User-defined methods for **describe-object** are likewise restricted.

**Examples:** None.

**Side Effects:**

Output to *standard output* or *terminal I/O*.

**Affected By:**

\***standard-output**\* and \***terminal-io**\*, methods on **describe-object** and **print-object** for *objects* having user-defined *classes*.

**Exceptional Situations:** None.

**See Also:**

**inspect, describe-object**

**Notes:** None.

## **Function DIGIT-CHAR-P**

**Syntax:**

**digit-char-p** *char* &optional *radix* => *weight*

**Arguments and Values:**

*char*—a *character*.

*radix*—a *radix*. The default is 10.

*weight*—either a non-negative *integer* less than *radix*, or *false*.

**Description:**

Tests whether *char* is a digit in the specified *radix* (i.e., with a weight less than *radix*). If it is a digit in that *radix*, its weight is returned as an *integer*; otherwise **nil** is returned.

**Examples:**

## CLHS: Declaration DYNAMIC-EXTENT

```
(digit-char-p #\5)      => 5
(digit-char-p #\5 2)    => false
(digit-char-p #\A)      => false
(digit-char-p #\a)      => false
(digit-char-p #\A 11)   => 10
(digit-char-p #\a 11)   => 10
(mapcar #'(lambda (radix)
             (map 'list #'(lambda (x) (digit-char-p x radix))
                  "059AaFGZ"))
        '(2 8 10 16 36))
=> ((0 NIL NIL NIL NIL NIL NIL)
     (0 5 NIL NIL NIL NIL NIL NIL)
     (0 5 9 NIL NIL NIL NIL NIL)
     (0 5 9 10 10 15 NIL NIL)
     (0 5 9 10 10 15 16 35))
```

### Affected By:

None. (In particular, the results of this predicate are independent of any special syntax which might have been enabled in the current readable.)

### Exceptional Situations:

None.

### See Also:

alphanumericp

### Notes:

Digits are graphic characters.

## Function DIGIT-CHAR

### Syntax:

**digit-char** *weight* &*optional radix* => *char*

### Arguments and Values:

*weight*—a non-negative integer.

*radix*—a radix. The default is 10.

*char*—a character or false.

### Description:

If *weight* is less than *radix*, **digit-char** returns a character which has that *weight* when considered as a digit in the specified radix. If the resulting character is to be an alphabetic[1] character, it will be an uppercase character.

If *weight* is greater than or equal to *radix*, **digit-char** returns false.

**Examples:**

```
(digit-char 0) => #\0
(digit-char 10 11) => #\A
(digit-char 10 10) => false
(digit-char 7) => #\7
(digit-char 12) => false
(digit-char 12 16) => #\C ;not #\c
(digit-char 6 2) => false
(digit-char 1 2) => #\1
```

**Affected By:** None.**Exceptional Situations:** None.**See Also:**[digit-char-p, graphic-char-p, Section 2.1 \(Character Syntax\)](#)**Notes:*****Function DIRECTORY*****Syntax:****directory** *pathspec* &*key* => *pathnames***Arguments and Values:***pathspec*—a pathname designator, which may contain wild components.*pathnames*—a list of physical pathnames.**Description:**

Determines which, if any, files that are present in the file system have names matching *pathspec*, and returns a fresh list of pathnames corresponding to the truenames of those files.

An implementation may be extended to accept implementation-defined keyword arguments to **directory**.

**Examples:** None.**Affected By:**

The host computer's file system.

**Exceptional Situations:**

If the attempt to obtain a directory listing is not successful, an error of type file-error is signaled.

**See Also:**

pathname, logical-pathname, ensure-directories-exist, Section 20.1 (File System Concepts), Section 20.1.2 (File Operations on Open and Closed Streams), Section 19.1.2 (Pathnames as Filenames)

#### Notes:

If the *pathspec* is not wild, the resulting list will contain either zero or one elements.

Common Lisp specifies "&key" in the argument list to directory even though no standardized keyword arguments to directory are defined. ":allow-other-keys t" may be used in conforming programs in order to quietly ignore any additional keywords which are passed by the program but not supported by the implementation.

## Function DISASSEMBLE

#### Syntax:

**disassemble** *fn* => nil

#### Arguments and Values:

*fn*—an extended function designator or a lambda expression.

#### Description:

The function disassemble is a debugging aid that composes symbolic instructions or expressions in some implementation-dependent language which represent the code used to produce the function which is or is named by the argument *fn*. The result is displayed to standard output in an implementation-dependent format.

If *fn* is a lambda expression or interpreted function, it is compiled first and the result is disassembled.

If the *fn designator* is a function name, the function that it names is disassembled. (If that function is an interpreted function, it is first compiled but the result of this implicit compilation is not installed.)

#### Examples:

```
(defun f (a) (1+ a)) => F
(eq (symbol-function 'f)
     (progn (disassemble 'f)
            (symbol-function 'f))) => true
```

**Side Effects:** None.

#### Affected By:

**\*standard-output\***.

#### Exceptional Situations:

Should signal an error of type type-error if *fn* is not an extended function designator or a lambda expression.

**See Also:** None.

**Notes:** None.

## Standard Generic Function DOCUMENTATION, (SETF DOCUMENTATION)

**Syntax:**

**documentation** *x doc-type => documentation*

**(setf documentation)** *new-value x doc-type => new-value*

**Argument Precedence Order:**

*doc-type, object*

**Method Signatures:**

**Functions, Macros, and Special Forms:**

**documentation** (*x function*) (*doc-type (eql 't)*)

**documentation** (*x function*) (*doc-type (eql 'function)*)

**documentation** (*x list*) (*doc-type (eql 'function)*)

**documentation** (*x list*) (*doc-type (eql 'compiler-macro)*)

**documentation** (*x symbol*) (*doc-type (eql 'function)*)

**documentation** (*x symbol*) (*doc-type (eql 'compiler-macro)*)

**documentation** (*x symbol*) (*doc-type (eql 'setf)*)

**(setf documentation)** *new-value (x function) (doc-type (eql 't))*

**(setf documentation)** *new-value (x function) (doc-type (eql 'function))*

**(setf documentation)** *new-value (x list) (doc-type (eql 'function))*

**(setf documentation)** *new-value (x list) (doc-type (eql 'compiler-macro))*

## CLHS: Declaration DYNAMIC-EXTENT

(**setf documentation**) *new-value* (*x symbol*) (*doc-type* (**eql** 'function))

(**setf documentation**) *new-value* (*x symbol*) (*doc-type* (**eql** 'compiler-macro))

(**setf documentation**) *new-value* (*x symbol*) (*doc-type* (**eql** 'setf))

### Method Combinations:

**documentation** (*x method-combination*) (*doc-type* (**eql** 't))

**documentation** (*x method-combination*) (*doc-type* (**eql** 'method-combination))

**documentation** (*x symbol*) (*doc-type* (**eql** 'method-combination))

(**setf documentation**) *new-value* (*x method-combination*) (*doc-type* (**eql** 't))

(**setf documentation**) *new-value* (*x method-combination*) (*doc-type* (**eql** 'method-combination))

(**setf documentation**) *new-value* (*x symbol*) (*doc-type* (**eql** 'method-combination))

### Methods:

**documentation** (*x standard-method*) (*doc-type* (**eql** 't))

(**setf documentation**) *new-value* (*x standard-method*) (*doc-type* (**eql** 't))

### Packages:

**documentation** (*x package*) (*doc-type* (**eql** 't))

(**setf documentation**) *new-value* (*x package*) (*doc-type* (**eql** 't))

### Types, Classes, and Structure Names:

**documentation** (*x standard-class*) (*doc-type* (**eql** 't))

**documentation** (*x* standard-class) (*doc-type* (**eql** ' type))

**documentation** (*x* structure-class) (*doc-type* (**eql** ' t))

**documentation** (*x* structure-class) (*doc-type* (**eql** ' type))

**documentation** (*x* symbol) (*doc-type* (**eql** ' type))

**documentation** (*x* symbol) (*doc-type* (**eql** ' structure))

(**setf documentation**) *new-value* (*x* standard-class) (*doc-type* (**eql** ' t))

(**setf documentation**) *new-value* (*x* structure-class) (*doc-type* (**eql** ' type))

(**setf documentation**) *new-value* (*x* structure-class) (*doc-type* (**eql** ' t))

(**setf documentation**) *new-value* (*x* structure-class) (*doc-type* (**eql** ' type))

(**setf documentation**) *new-value* (*x* symbol) (*doc-type* (**eql** ' type))

(**setf documentation**) *new-value* (*x* symbol) (*doc-type* (**eql** ' structure))

#### Variables:

**documentation** (*x* symbol) (*doc-type* (**eql** ' variable))

(**setf documentation**) *new-value* (*x* symbol) (*doc-type* (**eql** ' variable))

#### Arguments and Values:

*x*---an object.

*doc-type*---a symbol.

*documentation*---a string, or nil.

*new-value*---a string.

**Description:**

The generic function documentation returns the documentation string associated with the given object if it is available; otherwise it returns nil.

The generic function (`(setf documentation)`) updates the documentation string associated with x to new-value. If x is a list, it must be of the form (`(setf symbol)`).

Documentation strings are made available for debugging purposes. Conforming programs are permitted to use documentation strings when they are present, but should not depend for their correct behavior on the presence of those documentation strings. An implementation is permitted to discard documentation strings at any time for implementation-defined reasons.

The nature of the documentation string returned depends on the doc-type, as follows:

**compiler-macro**

Returns the documentation string of the compiler macro whose name is the function name x.

**function**

If x is a function name, returns the documentation string of the function, macro, or special operator whose name is x.

If x is a function, returns the documentation string associated with x.

**method-combination**

If x is a symbol, returns the documentation string of the method combination whose name is x.

If x is a method combination, returns the documentation string associated with x.

**setf**

Returns the documentation string of the setf expander whose name is the symbol x.

**structure**

Returns the documentation string associated with the structure name x.

**t**

Returns a documentation string specialized on the class of the argument x itself. For example, if x is a function, the documentation string associated with the function x is returned.

**type**

If x is a symbol, returns the documentation string of the class whose name is the symbol x, if there is such a class. Otherwise, it returns the documentation string of the type which is the type specifier symbol x.

If x is a structure class or standard class, returns the documentation string associated with the class x.

**variable**

Returns the documentation string of the dynamic variable or constant variable whose name is the symbol x.

A conforming implementation or a conforming program may extend the set of symbols that are acceptable as the doc-type.

**Examples:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:** None.

**Notes:**

This standard prescribes no means to retrieve the documentation strings for individual slots specified in a **defclass** form, but implementations might still provide debugging tools and/or programming language extensions which manipulate this information. Implementors wishing to provide such support are encouraged to consult the Metaobject Protocol for suggestions about how this might be done.

## **Function DPB**

**Syntax:**

**dpb** *newbyte bytespec integer => result-integer*

**Pronunciation:**

[,duh'pib] or [,duh'puhb] or ['dee'pee'bee]

**Arguments and Values:**

*newbyte*—an integer.

*bytespec*—a byte specifier.

*integer*—an integer.

*result-integer*—an integer.

**Description:**

**dpb** (deposit byte) is used to replace a field of bits within *integer*. **dpb** returns an integer that is the same as *integer* except in the bits specified by *bytespec*.

Let *s* be the size specified by *bytespec*; then the low *s* bits of *newbyte* appear in the result in the byte specified by *bytespec*. *Newbyte* is interpreted as being right-justified, as if it were the result of **ldb**.

**Examples:**

```
(dpb 1 (byte 1 10) 0) => 1024
(dpdb -2 (byte 2 10) 0) => 2048
(dpdb 1 (byte 2 10) 2048) => 1024
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**byte, deposit-field, ldb****Notes:**

```
(logbitp j (dpb m (byte s p) n))
== (if (and (>= j p) (< j (+ p s)))
      (logbitp (- j p) m)
      (logbitp j n))
```

In general,

```
(dpb x (byte 0 y) z) => z
```

for all valid values of  $x$ ,  $y$ , and  $z$ .

Historically, the name "dpb" comes from a DEC PDP-10 assembly language instruction meaning "deposit byte."

**Function DRIBBLE****Syntax:**

**dribble** &*optional pathname* => *implementation-dependent*

**Arguments and Values:**

*pathname*—a *pathname designator*.

**Description:**

Either *binds \*standard-input\** and *\*standard-output\** or takes other appropriate action, so as to send a record of the input/output interaction to a file named by *pathname*. **dribble** is intended to create a readable record of an interactive session.

If *pathname* is a *logical pathname*, it is translated into a physical pathname as if by calling **translate-logical-pathname**.

(**dribble**) terminates the recording of input and output and closes the dribble file.

If **dribble** is called while a *stream* to a "dribble file" is still open from a previous *call* to **dribble**, the effect is *implementation-defined*. For example, the already-*open stream* might be *closed*, or dribbling might occur both to the old *stream* and to a new one, or the old *stream* might stay open but not receive any further output, or the new request might be ignored, or some other action might be taken.

**Examples:** None.

**Affected By:**

The *implementation*.

**Exceptional Situations:**

## CLHS: Declaration DYNAMIC-EXTENT

If a failure occurs when performing some operation on the *file system* while creating the dribble file, an error of type **file-error** is signaled.

An error of type **file-error** might be signaled if *pathname* is a *designator* for a *wild pathname*.

**See Also:**

[Section 19.1.2 \(Pathnames as Filenames\)](#)

**Notes:**

**dribble** can return before subsequent *forms* are executed. It also can enter a recursive interaction loop, returning only when (**dribble**) is done.

## **Function ECHO-STREAM-INPUT-STREAM, ECHO-STREAM-OUTPUT-STREAM**

**Syntax:**

**echo-stream-input-stream** *echo-stream* => *input-stream*

**echo-stream-output-stream** *echo-stream* => *output-stream*

**Arguments and Values:**

*echo-stream*—an *echo stream*.

*input-stream*—an *input stream*.

*output-stream*—an *output stream*.

**Description:**

**echo-stream-input-stream** returns the *input stream* from which *echo-stream* receives input.

**echo-stream-output-stream** returns the *output stream* to which *echo-stream* sends output.

**Examples:** None.

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:** None.

**Notes:** None.

## Function ED

### Syntax:

`ed &optional x => implementation-dependent`

### Arguments and Values:

*x*—nil, a pathname, a string, or a function name. The default is nil.

### Description:

ed invokes the editor if the implementation provides a resident editor.

If *x* is nil, the editor is entered. If the editor had been previously entered, its prior state is resumed, if possible.

If *x* is a pathname or string, it is taken as the pathname designator for a file to be edited.

If *x* is a function name, the text of its definition is edited. The means by which the function text is obtained is implementation-defined.

**Examples:** None.

**Affected By:** None.

### Exceptional Situations:

The consequences are undefined if the implementation does not provide a resident editor.

Might signal type-error if its argument is supplied but is not a symbol, a pathname, or nil.

If a failure occurs when performing some operation on the file system while attempting to edit a file, an error of type file-error is signaled.

An error of type file-error might be signaled if *x* is a designator for a wild pathname.

Implementation-dependent additional conditions might be signaled as well.

### See Also:

pathname, logical-pathname, compile-file, load, Section 19.1.2 (Pathnames as Filenames)

**Notes:** None.

## Accessor ELT

### Syntax:

`elt sequence index => object`

## CLHS: Declaration DYNAMIC-EXTENT

(**setf** (**elt** *sequence index*) *new-object*)

### Arguments and Values:

*sequence*—a *proper sequence*.

*index*—a *valid sequence index* for *sequence*.

*object*—an *object*.

*new-object*—an *object*.

### Description:

*Accesses* the *element* of *sequence* specified by *index*.

### Examples:

```
(setq str (copy-seq "0123456789")) => "0123456789"
(elt str 6) => #\6
(setf (elt str 0) #\#) => #\#
str => "#123456789"
```

**Side Effects:** None.

**Affected By:** None.

### Exceptional Situations:

Should be prepared to signal an error of *type type-error* if *sequence* is not a *proper sequence*. Should signal an error of *type type-error* if *index* is not a *valid sequence index* for *sequence*.

### See Also:

[\*\*aref\*\*, \*\*nth\*\*, Section 3.2.1 \(Compiler Terminology\)](#)

### Notes:

**aref** may be used to *access vector* elements that are beyond the *vector's fill pointer*.

## function ENCODE-UNIVERSAL-TIME

### Syntax:

```
encode-universal-time second minute hour date month year &optional time-zone
=> universal-time
```

### Arguments and Values:

*second*, *minute*, *hour*, *date*, *month*, *year*, *time-zone*—the corresponding parts of a *decoded time*. (Note that

## CLHS: Declaration DYNAMIC-EXTENT

some of the nine values in a full decoded time are redundant, and so are not used as inputs to this function.)

*universal-time*---a universal time.

### Description:

**encode-universal-time** converts a time from Decoded Time format to a universal time.

If *time-zone* is supplied, no adjustment for daylight savings time is performed.

### Examples:

```
(encode-universal-time 0 0 0 1 1 1900 0) => 0
(encode-universal-time 0 0 1 4 7 1976 5) => 2414296800
;; The next example assumes Eastern Daylight Time.
(encode-universal-time 0 0 1 4 7 1976) => 2414293200
```

**Affected By:** None.

**Exceptional Situations:** None.

### See Also:

decode-universal-time, get-decoded-time

**Notes:** None.

## Function ENDP

### Syntax:

**endp** *list* => *generalized-boolean*

### Arguments and Values:

*list*---a list, which might be a dotted list or a circular list.

*generalized-boolean*---a generalized boolean.

### Description:

Returns true if *list* is the empty list. Returns false if *list* is a cons.

### Examples:

```
(endp nil) => true
(endp '(1 2)) => false
(endp (cddr '(1 2))) => true
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:**

Should signal an error of type type-error if *list* is not a list.

**See Also:** None.

**Notes:**

The purpose of endp is to test for the end of *proper list*. Since endp does not descend into a cons, it is well-defined to pass it a dotted list. However, if shorter "lists" are iteratively produced by calling cdr on such a dotted list and those "lists" are tested with endp, a situation that has undefined consequences will eventually result when the non-nil atom (which is not in fact a list) finally becomes the argument to endp. Since this is the usual way in which endp is used, it is conservative programming style and consistent with the intent of endp to treat endp as simply a function on proper lists which happens not to enforce an argument type of proper list except when the argument is atomic.

## Function ENSURE-DIRECTORIES-EXIST

**Syntax:**

**ensure-directories-exist** *pathspec* &key *verbose* => *pathspec*, *created*

**Arguments and Values:**

*pathspec*—a pathname designator.

*verbose*—a generalized boolean.

*created*—a generalized boolean.

**Description:**

Tests whether the directories containing the specified file actually exist, and attempts to create them if they do not.

If the containing directories do not exist and if *verbose* is true, then the implementation is permitted (but not required) to perform output to standard output saying what directories were created. If the containing directories exist, or if *verbose* is false, this function performs no output.

The primary value is the given *pathspec* so that this operation can be straightforwardly composed with other file manipulation expressions. The secondary value, *created*, is true if any directories were created.

**Examples:** None.

**Affected By:**

The host computer's file system.

**Exceptional Situations:**

An error of type file-error is signaled if the host, device, or directory part of *pathspec* is wild.

## CLHS: Declaration DYNAMIC-EXTENT

If the directory creation attempt is not successful, an error of **type file-error** is signaled; if this occurs, it might be the case that none, some, or all of the requested creations have actually occurred within the **file system**.

### See Also:

**probe-file**, **open**, Section 19.1.2 (Pathnames as Filenames)

**Notes:** None.

## **Function ENSURE-GENERIC-FUNCTION**

### Syntax:

**ensure-generic-function** *function-name* &*key argument-precedence-order* *declare documentation environment generic-function-class lambda-list method-class method-combination*

=> *generic-function*

### Arguments and Values:

*function-name*—a **function name**.

The keyword arguments correspond to the *option* arguments of **defgeneric**, except that the :method-class and :generic-function-class arguments can be **class objects** as well as names.

Method-combination -- method combination object.

Environment -- the same as the &environment argument to macro expansion functions and is used to distinguish between compile-time and run-time environments.

*generic-function*—a **generic function object**.

### Description:

The **function ensure-generic-function** is used to define a globally named **generic function** with no **methods** or to specify or modify options and declarations that pertain to a globally named **generic function** as a whole.

If *function-name* is not **fbound** in the **global environment**, a new **generic function** is created. If (**fdefinition** *function-name*) is an **ordinary function**, a **macro**, or a **special operator**, an error is signaled.

If *function-name* is a **list**, it must be of the form (**setf symbol**). If *function-name* specifies a **generic function** that has a different value for any of the following arguments, the **generic function** is modified to have the new value: :argument-precedence-order, :declare, :documentation, :method-combination.

If *function-name* specifies a **generic function** that has a different value for the :lambda-list argument, and the new value is congruent with the **lambda lists** of all existing **methods** or there are no **methods**, the value is changed; otherwise an error is signaled.

## CLHS: Declaration DYNAMIC-EXTENT

If *function-name* specifies a generic function that has a different value for the :generic-function-class argument and if the new generic function class is compatible with the old, change-class is called to change the class of the generic function; otherwise an error is signaled.

If *function-name* specifies a generic function that has a different value for the :method-class argument, the value is changed, but any existing methods are not changed.

**Examples:** None.

**Affected By:**

Existing function binding of *function-name*.

**Exceptional Situations:**

If (fdefinition *function-name*) is an ordinary function, a macro, or a special operator, an error of type error is signaled.

If *function-name* specifies a generic function that has a different value for the :lambda-list argument, and the new value is not congruent with the lambda list of any existing method, an error of type error is signaled.

If *function-name* specifies a generic function that has a different value for the :generic-function-class argument and if the new generic function class not is compatible with the old, an error of type error is signaled.

**See Also:**

defgeneric

**Notes:** None.

## Function EQ

**Syntax:**

**eq** *x y => generalized-boolean*

**Arguments and Values:**

*x*—an object.

*y*—an object.

*generalized-boolean*—a generalized boolean.

**Description:**

Returns true if its arguments are the same, identical object; otherwise, returns false.

**Examples:**

## CLHS: Declaration DYNAMIC-EXTENT

```
(eq 'a 'b) => false
(eq 'a 'a) => true
(eq 3 3)
=> true
OR=> false
(eq 3 3.0) => false
(eq 3.0 3.0)
=> true
OR=> false
(eq #c(3 -4) #c(3 -4))
=> true
OR=> false
(eq #c(3 -4.0) #c(3 -4)) => false
(eq (cons 'a 'b) (cons 'a 'c)) => false
(eq (cons 'a 'b) (cons 'a 'b)) => false
(eq '(a . b) '(a . b))
=> true
OR=> false
(progn (setq x (cons 'a 'b)) (eq x x)) => true
(progn (setq x '(a . b)) (eq x x)) => true
(eq #\A #\A)
=> true
OR=> false
(let ((x "Foo")) (eq x x)) => true
(eq "Foo" "Foo")
=> true
OR=> false
(eq "Foo" (copy-seq "Foo")) => false
(eq "FOO" "foo") => false
(eq "string-seq" (copy-seq "string-seq")) => false
(let ((x 5)) (eq x x))
=> true
OR=> false
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[eql](#), [equal](#), [equalp](#), [=](#), [Section 3.2 \(Compilation\)](#)

**Notes:**

*Objects* that appear the same when printed are not necessarily eq to each other. *Symbols* that print the same usually are eq to each other because of the use of the intern function. However, *numbers* with the same value need not be eq, and two similar *lists* are usually not identical.

An implementation is permitted to make "copies" of *characters* and *numbers* at any time. The effect is that Common Lisp makes no guarantee that eq is true even when both its arguments are "the same thing" if that thing is a *character* or *number*.

Most Common Lisp *operators* use eql rather than eq to compare objects, or else they default to eql and only use eq if specifically requested to do so. However, the following *operators* are defined to use eq rather than

**eq** in a way that cannot be overridden by the *code* which employs them:

<u>catch</u>	<u>aetf</u>	<u>throw</u>
<u>get</u>	<u>remf</u>	
<u>get-properties</u>	<u>remprop</u>	

**Figure 5–11. Operators that always prefer EQ over EQL**

### **Function =, /=, <, >, <=, >=**

#### **Syntax:**

```
= &rest numbers+ => generalized-boolean
/= &rest numbers+ => generalized-boolean
< &rest numbers+ => generalized-boolean
> &rest numbers+ => generalized-boolean
<= &rest numbers+ => generalized-boolean
>= &rest numbers+ => generalized-boolean
```

#### **Arguments and Values:**

*number*—*for*  $\leq$ ,  $\geq$ ,  $\leq=$ ,  $\geq=$ : a *real*; *for*  $\equiv$ ,  $/\equiv$ : a *number*.

*generalized-boolean*—*a generalized boolean*.

#### **Description:**

$\equiv$ ,  $/\equiv$ ,  $\leq$ ,  $\geq$ ,  $\leq=$ , and  $\geq=$  perform arithmetic comparisons on their arguments as follows:

$\equiv$

The value of  $\equiv$  is *true* if all *numbers* are the same in value; otherwise it is *false*. Two *complexes* are considered equal by  $\equiv$  if their real and imaginary parts are equal according to  $\equiv$ .

$/\equiv$

The value of  $/\equiv$  is *true* if no two *numbers* are the same in value; otherwise it is *false*.

$\leq$

The value of  $\leq$  is *true* if the *numbers* are in monotonically increasing order; otherwise it is *false*.

$\geq$

The value of  $\geq$  is *true* if the *numbers* are in monotonically decreasing order; otherwise it is *false*.

$\leq=$

The value of  $\leq=$  is *true* if the *numbers* are in monotonically nondecreasing order; otherwise it is *false*.

$\geq=$

## CLHS: Declaration DYNAMIC-EXTENT

The value of `>=` is *true* if the *numbers* are in monotonically nonincreasing order; otherwise it is *false*.

`≡, /≡, ≤, ≥, <=, and >=` perform necessary type conversions.

### Examples:

The uses of these functions are illustrated in the next figure.

<code>(= 3 3) is true.</code>	<code>(/= 3 3) is false.</code>
<code>(= 3 5) is false.</code>	<code>(/= 3 5) is true.</code>
<code>(= 3 3 3 3) is true.</code>	<code>(/= 3 3 3 3) is false.</code>
<code>(= 3 3 5 3) is false.</code>	<code>(/= 3 3 5 3) is false.</code>
<code>(= 3 6 5 2) is false.</code>	<code>(/= 3 6 5 2) is true.</code>
<code>(= 3 2 3) is false.</code>	<code>(/= 3 2 3) is false.</code>
<code>(&lt; 3 5) is true.</code>	<code>(&lt;= 3 5) is true.</code>
<code>(&lt; 3 -5) is false.</code>	<code>(&lt;= 3 -5) is false.</code>
<code>(&lt; 3 3) is false.</code>	<code>(&lt;= 3 3) is true.</code>
<code>(&lt; 0 3 4 6 7) is true.</code>	<code>(&lt;= 0 3 4 6 7) is true.</code>
<code>(&lt; 0 3 4 4 6) is false.</code>	<code>(&lt;= 0 3 4 4 6) is true.</code>
<code>(&gt; 4 3) is true.</code>	<code>(&gt;= 4 3) is true.</code>
<code>(&gt; 4 3 2 1 0) is true.</code>	<code>(&gt;= 4 3 2 1 0) is true.</code>
<code>(&gt; 4 3 3 2 0) is false.</code>	<code>(&gt;= 4 3 3 2 0) is true.</code>
<code>(&gt; 4 3 1 2 0) is false.</code>	<code>(&gt;= 4 3 1 2 0) is false.</code>
<code>(= 3) is true.</code>	<code>(/= 3) is true.</code>
<code>(&lt; 3) is true.</code>	<code>(&lt;= 3) is true.</code>
<code>(= 3.0 #c(3.0 0.0)) is true.</code>	<code>(/= 3.0 #c(3.0 1.0)) is true.</code>
<code>(= 3 3.0) is true.</code>	<code>(= 3.0s0 3.0d0) is true.</code>
<code>(= 0.0 -0.0) is true.</code>	<code>(= 5/2 2.5) is true.</code>
<code>(&gt; 0.0 -0.0) is false.</code>	<code>(= 0 -0.0) is true.</code>
<code>(&lt;= 0 x 9) is true if x is between 0 and 9, inclusive</code>	
<code>(&lt; 0.0 x 1.0) is true if x is between 0.0 and 1.0, exclusive</code>	
<code>(&lt; -1 j (length v)) is true if j is a valid array index for a vector v</code>	

**Figure 12–13. Uses of `/=`, `=`, `<`, `>`, `<=`, and `>=`**

**Affected By:** None.

### Exceptional Situations:

Might signal **type-error** if some *argument* is not a *real*. Might signal **arithmetic-error** if otherwise unable to fulfill its contract.

**See Also:** None.

### Notes:

`≡` differs from `eql` in that `(= 0.0 -0.0)` is always true, because `≡` compares the mathematical values of its operands, whereas `eql` compares the representational values, so to speak.

## Function EQL

### Syntax:

`eql x y => generalized-boolean`

**Arguments and Values:**

*x*—an object.

*y*—an object.

*generalized-boolean*—a generalized boolean.

**Description:**

The value of **eql** is true of two objects, *x* and *y*, in the following cases:

1. If *x* and *y* are eq.
2. If *x* and *y* are both numbers of the same type and the same value.
3. If they are both characters that represent the same character.

Otherwise the value of **eql** is false.

If an implementation supports positive and negative zeros as distinct values, then (**eql** 0.0 -0.0) returns false. Otherwise, when the syntax -0.0 is read it is interpreted as the value 0.0, and so (**eql** 0.0 -0.0) returns true.

**Examples:**

```
(eql 'a 'b) => false
(eql 'a 'a) => true
(eql 3 3) => true
(eql 3 3.0) => false
(eql 3.0 3.0) => true
(eql #c(3 -4) #c(3 -4)) => true
(eql #c(3 -4.0) #c(3 -4)) => false
(eql (cons 'a 'b) (cons 'a 'c)) => false
(eql (cons 'a 'b) (cons 'a 'b)) => false
(eql '(a . b) '(a . b))
=> true
OR=> false
(progn (setq x (cons 'a 'b)) (eql x x)) => true
(progn (setq x '(a . b)) (eql x x)) => true
(eql #\A #\A) => true
(eql "Foo" "Foo")
=> true
OR=> false
(eql "Foo" (copy-seq "Foo")) => false
(eql "FOO" "foo") => false
```

Normally (**eql** 1.0s0 1.0d0) is false, under the assumption that 1.0s0 and 1.0d0 are of distinct data types. However, implementations that do not provide four distinct floating-point formats are permitted to "collapse" the four formats into some smaller number of them; in such an implementation (**eql** 1.0s0 1.0d0) might be true.

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[eq](#), [equal](#), [equalp](#), [≡](#), [char=](#)

**Notes:**

[eql](#) is the same as [eq](#), except that if the arguments are *characters* or *numbers* of the same type then their values are compared. Thus [eql](#) tells whether two *objects* are conceptually the same, whereas [eq](#) tells whether two *objects* are implementationally identical. It is for this reason that [eql](#), not [eq](#), is the default comparison predicate for *operators* that take *sequences* as arguments.

[eql](#) may not be true of two *floats* even when they represent the same value. [≡](#) is used to compare mathematical values.

Two *complex* numbers are considered to be [eql](#) if their real parts are [eql](#) and their imaginary parts are [eql](#). For example, ([eql](#) #C(4 5) #C(4 5)) is *true* and ([eql](#) #C(4 5) #C(4.0 5.0)) is *false*. Note that while ([eql](#) #C(5.0 0.0) 5.0) is *false*, ([eql](#) #C(5 0) 5) is *true*. In the case of ([eql](#) #C(5.0 0.0) 5.0) the two arguments are of different types, and so cannot satisfy [eql](#). In the case of ([eql](#) #C(5 0) 5), #C(5 0) is not a *complex* number, but is automatically reduced to the *integer* 5.

## Function EQUAL

**Syntax:**

`equal x y => generalized-boolean`

**Arguments and Values:**

*x*—an *object*.

*y*—an *object*.

*generalized-boolean*—a *generalized boolean*.

**Description:**

Returns *true* if *x* and *y* are structurally similar (isomorphic) *objects*. *Objects* are treated as follows by [equal](#).

### *Symbols, Numbers, and Characters*

[equal](#) is *true* of two *objects* if they are *symbols* that are [eq](#), if they are *numbers* that are [eql](#), or if they are *characters* that are [eql](#).

### *Conses*

For *conses*, [equal](#) is defined recursively as the two *cars* being [equal](#) and the two *cdrs* being [equal](#).

### *Arrays*

Two *arrays* are [equal](#) only if they are [eq](#), with one exception: *strings* and *bit vectors* are compared element-by-element (using [eql](#)). If either *x* or *y* has a *fill pointer*, the *fill pointer* limits the number of elements examined by [equal](#). Uppercase and lowercase letters in *strings* are considered by [equal](#) to be different.

### *Pathnames*

## CLHS: Declaration DYNAMIC-EXTENT

Two pathnames are equal if and only if all the corresponding components (host, device, and so on) are equivalent. Whether or not uppercase and lowercase letters are considered equivalent in strings appearing in components is implementation-dependent. pathnames that are equal should be functionally equivalent.

**Other (Structures, hash-tables, instances, ...)**

Two other objects are equal only if they are eq.

equal does not descend any objects other than the ones explicitly specified above. The next figure summarizes the information given in the previous list. In addition, the figure specifies the priority of the behavior of equal, with upper entries taking priority over lower ones.

Type	Behavior
<u>number</u>	uses <u>eql</u>
<u>character</u>	uses <u>eql</u>
<u>cons</u>	descends
<u>bit vector</u>	descends
<u>string</u>	descends
<u>pathname</u>	"functionally equivalent"
<u>structure</u>	uses <u>eq</u>
Other <u>array</u>	uses <u>eq</u>
<u>hash table</u>	uses <u>eq</u>
Other <u>object</u>	uses <u>eq</u>

**Figure 5–12. Summary and priorities of behavior of equal**

Any two objects that are eql are also equal.

equal may fail to terminate if  $x$  or  $y$  is circular.

### Examples:

```
(equal 'a 'b) => false
(equal 'a 'a) => true
(equal 3 3) => true
(equal 3 3.0) => false
(equal 3.0 3.0) => true
(equal #c(3 -4) #c(3 -4)) => true
(equal #c(3 -4.0) #c(3 -4)) => false
(equal (cons 'a 'b) (cons 'a 'c)) => false
(equal (cons 'a 'b) (cons 'a 'b)) => true
(equal #\A #\A) => true
(equal #\A #\a) => false
(equal "Foo" "Foo") => true
(equal "Foo" (copy-seq "Foo")) => true
(equal "FOO" "foo") => false
(equal "This-string" "This-string") => true
(equal "This-string" "this-string") => false
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**eq, eql, equalp, =, string=, string-equal, char=, char-equal, tree-equal****Notes:**

*Object* equality is not a concept for which there is a uniquely determined correct algorithm. The appropriateness of an equality predicate can be judged only in the context of the needs of some particular program. Although these functions take any type of argument and their names sound very generic, equal and equalp are not appropriate for every application.

A rough rule of thumb is that two *objects* are equal if and only if their printed representations are the same.

**Function EQUALP****Syntax:**

**equalp** *x y => generalized-boolean*

**Arguments and Values:**

*x*—an object.

*y*—an object.

*generalized-boolean*—a generalized boolean.

**Description:**

Returns true if *x* and *y* are equal, or if they have components that are of the same type as each other and if those components are equalp; specifically, equalp returns true in the following cases:

Characters

If two characters are char-equal.

Numbers

If two numbers are the same under =.

Conses

If the two cars in the conses are equalp and the two cdrs in the conses are equalp.

Arrays

If two arrays have the same number of dimensions, the dimensions match, and the corresponding active elements are equalp. The types for which the arrays are specialized need not match; for example, a string and a general array that happens to contain the same characters are equalp.

Because equalp performs element-by-element comparisons of strings and ignores the case of characters, case distinctions are ignored when equalp compares strings.

Structures

If two structures S1 and S2 have the same class and the value of each slot in S1 is the same under equalp as the value of the corresponding slot in S2.

Hash Tables

equalp descends hash-tables by first comparing the count of entries and the :test function; if those are the same, it compares the keys of the tables using the :test function and then the values of the matching keys using equalp recursively.

## CLHS: Declaration DYNAMIC-EXTENT

**equalp** does not descend any *objects* other than the ones explicitly specified above. The next figure summarizes the information given in the previous list. In addition, the figure specifies the priority of the behavior of **equalp**, with upper entries taking priority over lower ones.

Type	Behavior
<u>number</u>	uses <u>=</u>
<u>character</u>	uses <u>char-equal</u>
<u>cons</u>	descends
<u>bit vector</u>	descends
<u>string</u>	descends
<u>pathname</u>	same as <u>equal</u>
<u>structure</u>	descends, as described above
Other <u>array</u>	descends
<u>hash table</u>	descends, as described above
Other <u>object</u>	uses <u>eq</u>

**Figure 5–13. Summary and priorities of behavior of equalp**

**Examples:**

```
(equalp 'a 'b) => false
(equalp 'a 'a) => true
(equalp 3 3) => true
(equalp 3 3.0) => true
(equalp 3.0 3.0) => true
(equalp #c(3 -4) #c(3 -4)) => true
(equalp #c(3 -4.0) #c(3 -4)) => true
(equalp (cons 'a 'b) (cons 'a 'c)) => false
(equalp (cons 'a 'b) (cons 'a 'b)) => true
(equalp #\A #\A) => true
(equalp #\A #\a) => true
(equalp "Foo" "Foo") => true
(equalp "Foo" (copy-seq "Foo")) => true
(equalp "FOO" "foo") => true

(setq array1 (make-array 6 :element-type 'integer
                         :initial-contents '(1 1 1 3 5 7)))
=> #(1 1 1 3 5 7)
(setq array2 (make-array 8 :element-type 'integer
                         :initial-contents '(1 1 1 3 5 7 2 6)
                         :fill-pointer 6))
=> #(1 1 1 3 5 7)
(equalp array1 array2) => true
(setq vector1 (vector 1 1 1 3 5 7)) => #(1 1 1 3 5 7)
(equalp array1 vector1) => true
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**eq, eql, equal, =, string=, string-equal, char=, char-equal**

**Notes:**

## CLHS: Declaration DYNAMIC-EXTENT

*Object* equality is not a concept for which there is a uniquely determined correct algorithm. The appropriateness of an equality predicate can be judged only in the context of the needs of some particular program. Although these functions take any type of argument and their names sound very generic, **equal** and **equalp** are not appropriate for every application.

## Function ERROR

### Syntax:

```
error datum &rest arguments =>
```

### Arguments and Values:

*datum, arguments*—*designators* for a *condition* of default type **simple-error**.

### Description:

**error** effectively invokes **signal** on the denoted *condition*.

If the *condition* is not handled, (*invoke-debugger condition*) is done. As a consequence of calling **invoke-debugger**, **error** cannot directly return; the only exit from **error** can come by non-local transfer of control in a handler or by use of an interactive debugging command.

### Examples:

```
(defun factorial (x)
  (cond ((or (not (typep x 'integer)) (minusp x))
          (error "~S is not a valid argument to FACTORIAL." x))
        ((zerop x) 1)
        (t (* x (factorial (- x 1))))))
=> FACTORIAL
(factorial 20)
=> 2432902008176640000
(factorial -1)
>> Error: -1 is not a valid argument to FACTORIAL.
>> To continue, type :CONTINUE followed by an option number:
>>   1: Return to Lisp Toplevel.
>> Debug>

      (setq a 'fred)
=> FRED
      (if (numberp a) (1+ a) (error "~S is not a number." A))
>> Error: FRED is not a number.
>> To continue, type :CONTINUE followed by an option number:
>>   1: Return to Lisp Toplevel.
>> Debug> :Continue 1
>> Return to Lisp Toplevel.

(define-condition not-a-number (error)
  ((argument :reader not-a-number-argument :initarg :argument))
  (:report (lambda (condition stream)
    (format stream "~S is not a number."
            (not-a-number-argument condition)))))
=> NOT-A-NUMBER
      (if (numberp a) (1+ a) (error 'not-a-number :argument a))
```

## CLHS: Declaration DYNAMIC-EXTENT

```
>> Error: FRED is not a number.  
>> To continue, type :CONTINUE followed by an option number:  
>> 1: Return to Lisp Toplevel.  
>> Debug> :Continue 1  
>> Return to Lisp Toplevel.
```

### Side Effects:

Handlers for the specified condition, if any, are invoked and might have side effects. Program execution might stop, and the debugger might be entered.

### Affected By:

Existing handler bindings.

### \*break-on-signals\*

**Exceptional Situations:** None.

Signals an error of type type-error if *datum* and *arguments* are not designators for a condition.

### See Also:

cerror, signal, format, ignore-errors, \*break-on-signals\*, handler-bind, Section 9.1 (Condition System Concepts)

### Notes:

Some implementations may provide debugger commands for interactively returning from individual stack frames. However, it should be possible for the programmer to feel confident about writing code like:

```
(defun wargames:no-win-scenario ()  
  (if (error "pushing the button would be stupid.")  
    (push-the-button))
```

In this scenario, there should be no chance that error will return and the button will get pushed.

While the meaning of this program is clear and it might be proven `safe' by a formal theorem prover, such a proof is no guarantee that the program is safe to execute. Compilers have been known to have bugs, computers to have signal glitches, and human beings to manually intervene in ways that are not always possible to predict. Those kinds of errors, while beyond the scope of the condition system to formally model, are not beyond the scope of things that should seriously be considered when writing code that could have the kinds of sweeping effects hinted at by this example.

## Function EVAL

### Syntax:

**eval** *form => result\**

### Arguments and Values:

## CLHS: Declaration DYNAMIC-EXTENT

*form*—a form.

*results*—the values yielded by the evaluation of form.

### Description:

Evaluates *form* in the current dynamic environment and the null lexical environment.

eval is a user interface to the evaluator.

The evaluator expands macro calls as if through the use of macroexpand-1.

Constants appearing in code processed by eval are not copied nor coalesced. The code resulting from the execution of eval references objects that are eql to the corresponding objects in the source code.

### 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
```

**Affected By:** None.

**Exceptional Situations:** None.

### See Also:

macroexpand-1, Section 3.1.2 (The Evaluation Model)

### Notes:

To obtain the current dynamic value of a symbol, use of symbol-value is equivalent (and usually preferable) to use of eval.

Note that an eval form involves two levels of evaluation for its argument. First, *form* is *evaluated* by the normal argument evaluation mechanism as would occur with any call. The object that results from this normal argument evaluation becomes the value of the form parameter, and is then *evaluated* as part of the eval form. For example:

```
(eval (list 'cdr (car '((quote (a . b)) c)))) => b
```

The argument form (list 'cdr (car '((quote (a . b)) c))) is evaluated in the usual way to produce the argument (cdr (quote (a . b))); eval then evaluates its argument, (cdr (quote (a . b))), to produce b. Since a single evaluation already occurs for any argument form in any function form, eval is sometimes said to perform "an extra level of evaluation."

**Function EVENP, ODDP****Syntax:**

**evenp** *integer* => *generalized-boolean*

**oddp** *integer* => *generalized-boolean*

**Arguments and Values:**

*integer*—an integer.

*generalized-boolean*—a generalized boolean.

**Description:**

**evenp** returns true if *integer* is even (divisible by two); otherwise, returns false.

**oddp** returns true if *integer* is odd (not divisible by two); otherwise, returns false.

**Examples:**

```
(evenp 0) => true
(oddp 1000000000000000000000000000000) => false
(oddp -1) => true
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:**

Should signal an error of type type-error if *integer* is not an integer.

**See Also:** None.

**Notes:**

```
(evenp integer) == (not (oddp integer))
(oddp integer) == (not (evenp integer))
```

**Function EVERY, SOME, NOTEVERY, NOTANY****Syntax:**

**every** *predicate &rest sequences+* => *generalized-boolean*

**some** *predicate &rest sequences+* => *result*

## CLHS: Declaration DYNAMIC-EXTENT

**notevery** *predicate &rest sequences+ => generalized-boolean*

**notany** *predicate &rest sequences+ => generalized-boolean*

### Arguments and Values:

*predicate*—a designator for a function of as many arguments as there are *sequences*.

*sequence*—a sequence.

*result*—an object.

*generalized-boolean*—a generalized boolean.

### Description:

**every**, **some**, **notevery**, and **notany** test elements of *sequences* for satisfaction of a given *predicate*. The first argument to *predicate* is an element of the first *sequence*; each succeeding argument is an element of a succeeding *sequence*.

*Predicate* is first applied to the elements with index 0 in each of the *sequences*, and possibly then to the elements with index 1, and so on, until a termination criterion is met or the end of the shortest of the *sequences* is reached.

**every** returns false as soon as any invocation of *predicate* returns false. If the end of a *sequence* is reached, **every** returns true. Thus, **every** returns true if and only if every invocation of *predicate* returns true.

**some** returns the first non-nil value which is returned by an invocation of *predicate*. If the end of a *sequence* is reached without any invocation of the *predicate* returning true, **some** returns false. Thus, **some** returns true if and only if some invocation of *predicate* returns true.

**notany** returns false as soon as any invocation of *predicate* returns true. If the end of a *sequence* is reached, **notany** returns true. Thus, **notany** returns true if and only if it is not the case that any invocation of *predicate* returns true.

**notevery** returns true as soon as any invocation of *predicate* returns false. If the end of a *sequence* is reached, **notevery** returns false. Thus, **notevery** returns true if and only if it is not the case that every invocation of *predicate* returns true.

### Examples:

```
(every #'characterp "abc") => true
(some #'= '(1 2 3 4 5) '(5 4 3 2 1)) => true
(notevery #'< '(1 2 3 4) '(5 6 7 8) '(9 10 11 12)) => false
(notany #'> '(1 2 3 4) '(5 6 7 8) '(9 10 11 12)) => true
```

**Affected By:** None.

### Exceptional Situations:

Should signal type-error if its first argument is neither a symbol nor a function or if any subsequent argument

is not a *proper sequence*.

Other exceptional situations are possible, depending on the nature of the *predicate*.

#### See Also:

[and, or](#), [Section 3.6 \(Traversal Rules and Side Effects\)](#)

#### Notes:

```
(notany predicate sequence*) == (not (some predicate sequence*))  
(notevery predicate sequence*) == (not (every predicate sequence*))
```

## Function EXP, EXPT

#### Syntax:

**exp** *number* => *result*

**expt** *base-number* *power-number* => *result*

#### Arguments and Values:

*number*—a *number*.

*base-number*—a *number*.

*power-number*—a *number*.

*result*—a *number*.

#### Description:

**exp** and **expt** perform exponentiation.

**exp** returns *e* raised to the power *number*, where *e* is the base of the natural logarithms. **exp** has no branch cut.

**expt** returns *base-number* raised to the power *power-number*. If the *base-number* is a *rational* and *power-number* is an *integer*, the calculation is exact and the result will be of type *rational*; otherwise a floating-point approximation might result. For **expt** of a *complex rational* to an *integer* power, the calculation must be exact and the result is of type (or *rational* (*complex rational*)).

The result of **expt** can be a *complex*, even when neither argument is a *complex*, if *base-number* is negative and *power-number* is not an *integer*. The result is always the *principal complex value*. For example, (**expt** -8 1/3) is not permitted to return -2, even though -2 is one of the cube roots of -8. The *principal* cube root is a *complex* approximately equal to #C(1.0 1.73205), not -2.

**expt** is defined as  $b^x = e^x \log b$ . This defines the *principal values* precisely. The range of **expt** is the entire complex plane. Regarded as a function of *x*, with *b* fixed, there is no branch cut. Regarded as a function of *b*, with *x* fixed, there is in general a branch cut along the negative real axis, continuous with quadrant II. The

## CLHS: Declaration DYNAMIC-EXTENT

domain excludes the origin. By definition,  $0^0=1$ . If  $b=0$  and the real part of  $x$  is strictly positive, then  $b^x=0$ . For all other values of  $x$ ,  $0^x$  is an error.

When *power-number* is an integer 0, then the result is always the value one in the type of *base-number*, even if the *base-number* is zero (of any type). That is:

```
(expt x 0) == (coerce 1 (type-of x))
```

If *power-number* is a zero of any other type, then the result is also the value one, in the type of the arguments after the application of the contagion rules in Section 12.1.1.2 (Contagion in Numeric Operations), with one exception: the consequences are undefined if *base-number* is zero when *power-number* is zero and not of type integer.

### Examples:

```
(exp 0) => 1.0
(exp 1) => 2.718282
(exp (log 5)) => 5.0
(expt 2 8) => 256
(expt 4 .5) => 2.0
(expt #c(0 1) 2) => -1
(expt #c(2 2) 3) => #C(-16 16)
(expt #c(2 2) 4) => -64
```

**Affected By:** None.

**Exceptional Situations:** None.

### See Also:

[log](#), [Section 12.1.3.3 \(Rule of Float Substitutability\)](#)

### Notes:

Implementations of expt are permitted to use different algorithms for the cases of a *power-number* of type rational and a *power-number* of type float.

Note that by the following logic, (sqrt (expt *x* 3)) is not equivalent to (expt *x* 3/2).

```
(setq x (exp (/ (* 2 pi #c(0 1)) 3))) ;exp(2.pi.i/3)
(expt x 3) => 1 ;except for round-off error
(sqrt (expt x 3)) => 1 ;except for round-off error
(expt x 3/2) => -1 ;except for round-off error
```

## Function EXPORT

### Syntax:

**export** *symbols* &*optional package* => *t*

### Arguments and Values:

*symbols*—a designator for a list of symbols.

*package*---a package designator. The default is the current package.

### Description:

**export** makes one or more *symbols* that are accessible in *package* (whether directly or by inheritance) be external symbols of that *package*.

If any of the *symbols* is already accessible as an external symbol of *package*, **export** has no effect on that symbol. If the symbol is present in *package* as an internal symbol, it is simply changed to external status. If it is accessible as an internal symbol via use-package, it is first *imported* into *package*, then exported. (The symbol is then present in the *package* whether or not *package* continues to use the package through which the symbol was originally inherited.)

**export** makes each *symbol accessible* to all the packages that use *package*. All of these packages are checked for name conflicts: (*export s p*) does (*find-symbol (symbol-name s) q*) for each package *q* in (*package-used-by-list p*). Note that in the usual case of an **export** during the initial definition of a package, the result of package-used-by-list is nil and the name-conflict checking takes negligible time. When multiple changes are to be made, for example when **export** is given a *list of symbols*, it is permissible for the implementation to process each change separately, so that aborting from a name conflict caused by any but the first *symbol* in the *list* does not unexport the first *symbol* in the *list*. However, aborting from a name-conflict error caused by **export** of one of *symbols* does not leave that *symbol accessible* to some packages and inaccessible to others; with respect to each of *symbols* processed, **export** behaves as if it were as an atomic operation.

A name conflict in **export** between one of *symbols* being exported and a *symbol* already present in a package that would inherit the newly-exported *symbol* may be resolved in favor of the exported *symbol* by uninterning the other one, or in favor of the already-present *symbol* by making it a shadowing symbol.

### Examples:

```
(make-package 'temp :use nil) => #<PACKAGE "TEMP">
(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
```

### Side Effects:

The package system is modified.

### Affected By:

Accessible symbols.

### Exceptional Situations:

If any of the *symbols* is not accessible at all in *package*, an error of type package-error is signaled that is correctable by permitting the user to interactively specify whether that *symbol* should be *imported*.

### See Also:

**import, unexport, Section 11.1 (Package Concepts)**

**Notes:** None.

**Function FBOUNDP**

**Syntax:**

**fboundp** *name* => *generalized-boolean*

**Pronunciation:**

[,ef'bandpee]

**Arguments and Values:**

*name*—*a function name*.

*generalized-boolean*—*a generalized boolean*.

**Description:**

Returns *true* if *name* is *fbound*; otherwise, returns *false*.

**Examples:**

```
(fboundp 'car) => true
(fboundp 'nth-value) => false
(fboundp 'with-open-file) => true
(fboundp 'unwind-protect) => true
(defun my-function (x) x) => MY-FUNCTION
(fboundp 'my-function) => true
(let ((saved-definition (symbol-function 'my-function)))
  (unwind-protect (progn (fmakunbound 'my-function)
                         (fboundp 'my-function))
    (setf (symbol-function 'my-function) saved-definition)))
=> false
(fboundp 'my-function) => true
(defmacro my-macro (x) `',x) => MY-MACRO
(fboundp 'my-macro) => true
(fmakunbound 'my-function) => MY-FUNCTION
(fboundp 'my-function) => false
(flet ((my-function (x) x))
  (fboundp 'my-function)) => false
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:**

Should signal an error of *type type-error* if *name* is not a *function name*.

**See Also:**

Function FBOUNDP

**symbol-function, fmakunbound, fdefinition****Notes:**

It is permissible to call **symbol-function** on any symbol that is fbound.

**fboundp** is sometimes used to "guard" an access to the function cell, as in:

```
(if (fboundp x) (symbol-function x))
```

Defining a setf expander *F* does not cause the setf function (**setf** *F*) to become defined.

**Accessor FDEFINITION****Syntax:**

**fdefinition** *function-name* => *definition*

```
(setf (fdefinition function-name) new-definition)
```

**Arguments and Values:**

*function-name*—a function name. In the non-**setf** case, the name must be fbound in the global environment.

*definition*—Current global function definition named by *function-name*.

*new-definition*—a function.

**Description:**

**fdefinition** *accesses* the current global function definition named by *function-name*. The definition may be a function or may be an object representing a special form or macro. The value returned by **fdefinition** when **fboundp** returns true but the *function-name* denotes a macro or special form is not well-defined, but **fdefinition** does not signal an error.

**Examples:** None.

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:**

Should signal an error of type type-error if *function-name* is not a function name.

An error of type undefined-function is signaled in the non-**setf** case if *function-name* is not fbound.

**See Also:**

**fboundp, fmakunbound, macro-function, special-operator-p, symbol-function**

**Notes:**

**fdefinition** cannot access the value of a lexical function name produced by **flet** or **labels**; it can access only the global function value.

**setf** can be used with **fdefinition** to replace a global function definition when the *function-name*'s function definition does not represent a special form. **setf** of **fdefinition** requires a function as the new value. It is an error to set the **fdefinition** of a *function-name* to a symbol, a list, or the value returned by **fdefinition** on the name of a macro or special form.

## Function FILE-AUTHOR

**Syntax:**

**file-author** *pathspec* => *author*

**Arguments and Values:**

*pathspec*—a pathname designator.

*author*—a string or nil.

**Description:**

Returns a string naming the author of the *file* specified by *pathspec*, or nil if the author's name cannot be determined.

**Examples:**

```
(with-open-file (stream ">relativity>general.text")
  (file-author s))
=> "albert"
```

**Affected By:**

The host computer's file system.

Other users of the *file* named by *pathspec*.

**Exceptional Situations:**

An error of type **file-error** is signaled if *pathspec* is wild.

An error of type **file-error** is signaled if the file system cannot perform the requested operation.

**See Also:**

[pathname](#), [logical-pathname](#), [Section 20.1 \(File System Concepts\)](#), [Section 19.1.2 \(Pathnames as Filenames\)](#)

**Notes:** None.

**Function FILE-ERROR-PATHNAME****Syntax:****file-error-pathname** *condition => pathspec***Arguments and Values:***condition*—a *condition* of type file-error.*pathspec*—a *pathname designator*.**Description:**Returns the "offending pathname" of a *condition* of type file-error.**Examples:** None.**Side Effects:** None.**Affected By:** None.**Exceptional Situations:****See Also:****file-error**, Section 9 (Conditions)**Notes:** None.**Function FILE-LENGTH****Syntax:****file-length** *stream => length***Arguments and Values:***stream*—a *stream associated with a file*.*length*—a non-negative *integer* or *nil*.**Description:****file-length** returns the length of *stream*, or *nil* if the length cannot be determined.For a binary file, the length is measured in units of the *element type* of the *stream*.**Examples:**

(with-open-file (s "decimal-digits.text"

## CLHS: Declaration DYNAMIC-EXTENT

```
:direction :output :if-exists :error)
(princ "0123456789" s)
(truename s))
=> #P"A:>Joe>decimal-digits.text.1"
(with-open-file (s "decimal-digits.text")
  (file-length s))
=> 10
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:**

Should signal an error of type type-error if *stream* is not a *stream associated with a file*.

**See Also:**

open

**Notes:** None.

## Function FILE-POSITION

**Syntax:**

**file-position** *stream* => *position*

**file-position** *stream* *position-spec* => *success-p*

**Arguments and Values:**

*stream*—a stream.

*position-spec*—a file position designator.

*position*—a file position or nil.

*success-p*—a generalized boolean.

**Description:**

Returns or changes the current position within a *stream*.

When *position-spec* is not supplied, **file-position** returns the current file position in the *stream*, or nil if this cannot be determined.

When *position-spec* is supplied, the file position in *stream* is set to that file position (if possible). **file-position** returns true if the repositioning is performed successfully, or false if it is not.

An integer returned by **file-position** of one argument should be acceptable as *position-spec* for use with the same file.

## CLHS: Declaration DYNAMIC-EXTENT

For a character file, performing a single **read-char** or **write-char** operation may cause the file position to be increased by more than 1 because of character-set translations (such as translating between the Common Lisp #\Newline character and an external ASCII carriage-return/line-feed sequence) and other aspects of the implementation. For a binary file, every **read-byte** or **write-byte** operation increases the file position by 1.

### Examples:

```
(defun tester ()
  (let ((noticed '()) file-written)
    (flet ((notice (x) (push x noticed) x))
      (with-open-file (s "test.bin"
                         :element-type '(unsigned-byte 8)
                         :direction :output
                         :if-exists :error)
        (notice (file-position s)) ;1
        (write-byte 5 s)
        (write-byte 6 s)
        (let ((p (file-position s)))
          (notice p) ;2
          (notice (when p (file-position s (1- p)))));3
          (write-byte 7 s)
          (notice (file-position s)) ;4
          (setq file-written (truename s)))
        (with-open-file (s file-written
                           :element-type '(unsigned-byte 8)
                           :direction :input)
          (notice (file-position s)) ;5
          (let ((length (file-length s)))
            (notice length) ;6
            (when length
              (dotimes (i length)
                (notice (read-byte s))))));7, ...
          (nreverse noticed))))
  => tester
  (tester)
=> (0 2 T 2 0 2 5 7)
OR=> (0 2 NIL 3 0 3 5 6 7)
OR=> (NIL NIL NIL NIL NIL NIL)
```

### Side Effects:

When the *position-spec* argument is supplied, the **file position** in the *stream* might be moved.

### Affected By:

The value returned by **file-position** increases monotonically as input or output operations are performed.

### Exceptional Situations:

If *position-spec* is supplied, but is too large or otherwise inappropriate, an error is signaled.

### See Also:

**file-length**, **file-string-length**, **open**

### Notes:

## CLHS: Declaration DYNAMIC-EXTENT

Implementations that have character files represented as a sequence of records of bounded size might choose to encode the file position as, for example,

`<<record-number>>*<<max-record-size>>+<<character-within-record>>`. This is a valid encoding because it increases monotonically as each character is read or written, though not necessarily by 1 at each step. An integer might then be considered "inappropriate" as *position-spec* to file-position if, when decoded into record number and character number, it turned out that the supplied record was too short for the specified character number.

## Function FILE-STRING-LENGTH

### Syntax:

**file-string-length** *stream object => length*

### Arguments and Values:

*stream*—an output character file stream.

*object*—a string or a character.

*length*—a non-negative integer, or nil.

### Description:

**file-string-length** returns the difference between what (`file-position stream`) would be after writing *object* and its current value, or nil if this cannot be determined.

The returned value corresponds to the current state of *stream* at the time of the call and might not be the same if it is called again when the state of the stream has changed.

**Examples:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:** None.

**Notes:** None.

## Function FILE-WRITE-DATE

### Syntax:

**file-write-date** *pathspec => date*

### Arguments and Values:

*pathspec*—a pathname designator.

*date*---a universal time or nil.

#### Description:

Returns a universal time representing the time at which the file specified by *pathspec* was last written (or created), or returns nil if such a time cannot be determined.

#### Examples:

```
(with-open-file (s "noel.text"
                    :direction :output :if-exists :error)
  (format s "~&Dear Santa,~2%I was good this year. ~
            Please leave lots of toys.~2%Love, Sue~
            ~2%attachments: milk, cookies~%")
  (truename s))
=> #P"CUPID:/susan/noel.text"
(with-open-file (s "noel.text")
  (file-write-date s))
=> 2902600800
```

#### Affected By:

The host computer's file system.

#### Exceptional Situations:

An error of type file-error is signaled if *pathspec* is wild.

An error of type file-error is signaled if the file system cannot perform the requested operation.

#### See Also:

**Notes:** None.

## Function FILL

#### Syntax:

`fill sequence item &key start end => sequence`

#### Arguments and Values:

*sequence*---a proper sequence.

*item*---a sequence.

*start*, *end*---bounding index designators of *sequence*. The defaults for *start* and *end* are 0 and nil, respectively.

#### Description:

Replaces the elements of *sequence* bounded by *start* and *end* with *item*.

**Examples:**

```
(fill (list 0 1 2 3 4 5) '(444)) => ((444) (444) (444) (444) (444) (444))
(fill (copy-seq "01234") #\e :start 3) => "012ee"
(setq x (vector 'a 'b 'c 'd 'e)) => #(A B C D E)
(fill x 'z :start 1 :end 3) => #(A Z Z D E)
x => #(A Z Z D E)
(fill x 'p) => #(P P P P P)
x => #(P P P P P)
```

**Side Effects:**

*Sequence* is destructively modified.

**Affected By:** None.

**Exceptional Situations:**

Should be prepared to signal an error of type type-error if *sequence* is not a proper sequence. Should signal an error of type type-error if *start* is not a non-negative integer. Should signal an error of type type-error if *end* is not a non-negative integer or nil.

**See Also:**

replace, nsubstitute

**Notes:**

```
(fill sequence item) == (nsubstitute-if item (constantly t) sequence)
```

**Accessor FILL-POINTER****Syntax:**

**fill-pointer** *vector* => *fill-pointer*

```
(setf (fill-pointer vector) new-fill-pointer)
```

**Arguments and Values:**

*vector*—a vector with a fill pointer.

*fill-pointer*, *new-fill-pointer*—a valid fill pointer for the *vector*.

**Description:**

Accesses the fill pointer of *vector*.

**Examples:**

```
(setq a (make-array 8 :fill-pointer 4)) => #(NIL NIL NIL NIL)
(fill-pointer a) => 4
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(dotimes (i (length a)) (setf (aref a i) (* i i))) => NIL
a => #(0 1 4 9)
(setf (fill-pointer a) 3) => 3
(fill-pointer a) => 3
a => #(0 1 4)
(setf (fill-pointer a) 8) => 8
a => #(0 1 4 9 NIL NIL NIL NIL)
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:**

Should signal an error of type type-error if vector is not a vector with a fill pointer.

**See Also:**

make-array, length

**Notes:**

There is no operator that will remove a vector's fill pointer.

## Function FIND, FIND-IF, FIND-IF-NOT

**Syntax:**

**find** *item sequence &key from-end test test-not start end key => element*

**find-if** *predicate sequence &key from-end start end key => element*

**find-if-not** *predicate sequence &key from-end start end key => element*

**Arguments and Values:**

*item*—an object.

*sequence*—a proper sequence.

*predicate*—a designator for a function of one argument that returns a generalized boolean.

*from-end*—a generalized boolean. The default is false.

*test*—a designator for a function of two arguments that returns a generalized boolean.

*test-not*—a designator for a function of two arguments that returns a generalized boolean.

*start, end*—bounding index designators of *sequence*. The defaults for *start* and *end* are 0 and nil, respectively.

*key*—a designator for a function of one argument, or nil.

*element*—an element of the sequence, or nil.

### Description:

**find**, **find-if**, and **find-if-not** each search for an element of the sequence bounded by *start* and *end* that *satisfies the predicate predicate* or that *satisfies the test test* or *test-not*, as appropriate.

If *from-end* is true, then the result is the rightmost element that *satisfies the test*.

If the sequence contains an element that *satisfies the test*, then the leftmost or rightmost sequence element, depending on *from-end*, is returned; otherwise nil is returned.

### 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
(find-if-not #'complexp
             '#(3.5 2 #C(1.0 0.0) #C(0.0 1.0))
             :start 2) => NIL
```

**Side Effects:** None.

**Affected By:** None.

### Exceptional Situations:

Should be prepared to signal an error of type type-error if sequence is not a proper sequence.

### See Also:

position, Section 17.2 (Rules about Test Functions), Section 3.6 (Traversal Rules and Side Effects)

### Notes:

The :test-not argument is deprecated.

The function find-if-not is deprecated.

## Function FIND-ALL-SYMBOLS

### Syntax:

**find-all-symbols** *string* => *symbols*

### Arguments and Values:

*string*—a string designator.

*symbols*—a list of symbols.

**Description:**

**find-all-symbols** searches every *registered package* for *symbols* that have a *name* that is the *same* (under *string=*) as *string*. A *list* of all such *symbols* is returned. Whether or how the *list* is ordered is *implementation-dependent*.

**Examples:**

```
(find-all-symbols 'car)
=> (CAR)
OR=> (CAR VEHICLES:CAR)
OR=> (VEHICLES:CAR CAR)
(intern "CAR" (make-package 'temp :use nil)) => TEMP::CAR, NIL
(find-all-symbols 'car)
=> (TEMP::CAR CAR)
OR=> (CAR TEMP::CAR)
OR=> (TEMP::CAR CAR VEHICLES:CAR)
OR=> (CAR TEMP::CAR VEHICLES:CAR)
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:****find-symbol**

**Notes:** None.

**Accessor FIND-CLASS****Syntax:**

**find-class** *symbol* &*optional errorp environment* => *class*

(**setf** (**find-class** *symbol* &*optional errorp environment*) *new-class*)

**Arguments and Values:**

*symbol*---a *symbol*.

*errorp*---a *generalized boolean*. The default is *true*.

*environment* -- same as the &*environment* argument to macro expansion functions and is used to distinguish between compile-time and run-time environments. The &*environment* argument has *dynamic extent*; the consequences are undefined if the &*environment* argument is referred to outside the *dynamic extent* of the macro expansion function.

*class*---a *class object*, or *nil*.

**Description:**

Returns the *class object* named by the *symbol* in the *environment*. If there is no such *class*, *nil* is returned if *errorp* is *false*; otherwise, if *errorp* is *true*, an error is signaled.

The *class* associated with a particular *symbol* can be changed by using *setf* with **find-class**; or, if the new *class* given to *setf* is *nil*, the *class* association is removed (but the *class object* itself is not affected). The results are undefined if the user attempts to change or remove the *class* associated with a *symbol* that is defined as a *type specifier* in this standard. See [Section 4.3.7 \(Integrating Types and Classes\)](#).

When using *setf* of **find-class**, any *errorp* argument is *evaluated* for effect, but any *values* it returns are ignored; the *errorp parameter* is permitted primarily so that the *environment parameter* can be used.

The *environment* might be used to distinguish between a compile-time and a run-time environment.

**Examples:** None.

**Affected By:** None.

**Exceptional Situations:**

If there is no such *class* and *errorp* is *true*, **find-class** signals an error of **type error**.

**See Also:**

[defmacro](#), [Section 4.3.7 \(Integrating Types and Classes\)](#)

**Notes:** None.

**Standard Generic Function FIND-METHOD****Syntax:**

**find-method** *generic-function* *method-qualifiers* *specializers* &*optional errorp*

=> *method*

**Method Signatures:**

**find-method** (*generic-function* [standard-generic-function](#)) *method-qualifiers* *specializers* &*optional errorp*

**Arguments and Values:**

*generic-function*—a *generic function*.

*method-qualifiers*—a *list*.

*specializers*—a *list*.

*errorp*—a *generalized boolean*. The default is *true*.

*method*—a method object, or nil.

#### Description:

The generic function find-method takes a generic function and returns the method object that agrees on qualifiers and parameter specializers with the method-qualifiers and specializers arguments of find-method. Method-qualifiers contains the method qualifiers for the method. The order of the method qualifiers is significant. For a definition of agreement in this context, see Section 7.6.3 (Agreement on Parameter Specializers and Qualifiers).

The specializers argument contains the parameter specializers for the method. It must correspond in length to the number of required arguments of the generic function, or an error is signaled. This means that to obtain the default method on a given generic-function, a list whose elements are the class t must be given.

If there is no such method and errorp is true, find-method signals an error. If there is no such method and errorp is false, find-method returns nil.

#### Examples:

```
(defmethod some-operation ((a integer) (b float)) (list a b))
=> #<STANDARD-METHOD SOME-OPERATION (INTEGER FLOAT) 26723357>
  (find-method #'some-operation '() (mapcar #'find-class '(integer float)))
=> #<STANDARD-METHOD SOME-OPERATION (INTEGER FLOAT) 26723357>
  (find-method #'some-operation '() (mapcar #'find-class '(integer integer)))
>> Error: No matching method
  (find-method #'some-operation '() (mapcar #'find-class '(integer integer)) nil)
=> NIL
```

#### Affected By:

add-method, defclass, defgeneric, defmethod

#### Exceptional Situations:

If the specializers argument does not correspond in length to the number of required arguments of the generic-function, an an error of type error is signaled.

If there is no such method and errorp is true, find-method signals an error of type error.

#### See Also:

**Notes:** None.

## Function FIND-PACKAGE

#### Syntax:

**find-package** *name* => *package*

#### Arguments and Values:

*name*—a string designator or a package object.

*package*---a package object or nil.

**Description:**

If *name* is a string designator, **find-package** locates and returns the package whose name or nickname is *name*. This search is case sensitive. If there is no such package, **find-package** returns nil.

If *name* is a package object, that package object is returned.

**Examples:**

```
(find-package 'common-lisp) => #<PACKAGE "COMMON-LISP">
(find-package "COMMON-LISP-USER") => #<PACKAGE "COMMON-LISP-USER">
(find-package 'not-there) => NIL
```

**Side Effects:** None.

**Affected By:**

The set of packages created by the implementation.

defpackage, delete-package, make-package, rename-package

**Exceptional Situations:** None.

**See Also:**

make-package

**Notes:** None.

## **Function FIND-RESTART**

**Syntax:**

**find-restart** *identifier* &*optional condition*

restart

**Arguments and Values:**

*identifier*---a non-nil symbol, or a restart.

*condition*---a condition object, or nil.

*restart*---a restart or nil.

**Description:**

**find-restart** searches for a particular restart in the current dynamic environment.

## CLHS: Declaration DYNAMIC-EXTENT

When *condition* is *non-nil*, only those *restarts* are considered that are either explicitly associated with that *condition*, or not associated with any *condition*; that is, the excluded *restarts* are those that are associated with a non-empty set of *conditions* of which the given *condition* is not an *element*. If *condition* is *nil*, all *restarts* are considered.

If *identifier* is a *symbol*, then the innermost (most recently established) *applicable restart* with that *name* is returned. *nil* is returned if no such restart is found.

If *identifier* is a currently active restart, then it is returned. Otherwise, *nil* is returned.

### Examples:

```
(restart-case
  (let ((r (find-restart 'my-restart)))
    (format t "~S is named ~S" r (restart-name r)))
  (my-restart () nil))
=> #<RESTART 32307325> is named MY-RESTART
=> NIL
(find-restart 'my-restart)
=> NIL
```

**Side Effects:** None.

### Affected By:

Existing restarts.

restart-case, restart-bind, with-condition-restarts.

**Exceptional Situations:** None.

### See Also:

compute-restarts

### Notes:

```
(find-restart identifier)
== (find identifier (compute-restarts) :key :restart-name)
```

Although anonymous restarts have a name of *nil*, the consequences are unspecified if *nil* is given as an *identifier*. Occasionally, programmers lament that *nil* is not permissible as an *identifier* argument. In most such cases, compute-restarts can probably be used to simulate the desired effect.

## Function FIND-SYMBOL

### Syntax:

**find-symbol** *string &optional package => symbol, status*

### Arguments and Values:

*string*---a string.

*package*---a package designator. The default is the current package.

*symbol*---a symbol accessible in the *package*, or nil.

*status*---one of :inherited, :external, :internal, or nil.

### Description:

**find-symbol** locates a symbol whose name is *string* in a package. If a symbol named *string* is found in *package*, directly or by inheritance, the symbol found is returned as the first value; the second value is as follows:

- :internal  
If the symbol is present in *package* as an internal symbol.
- :external  
If the symbol is present in *package* as an external symbol.
- :inherited  
If the symbol is inherited by *package* through use-package, but is not present in *package*.

If no such symbol is accessible in *package*, both values are nil.

### Examples:

```
(find-symbol "NEVER-BEFORE-USED") => NIL, NIL
(find-symbol "NEVER-BEFORE-USED") => NIL, NIL
(intern "NEVER-BEFORE-USED") => NEVER-BEFORE-USED, NIL
(intern "NEVER-BEFORE-USED") => NEVER-BEFORE-USED, :INTERNAL
(find-symbol "NEVER-BEFORE-USED") => NEVER-BEFORE-USED, :INTERNAL
(find-symbol "never-before-used") => NIL, NIL
(find-symbol "CAR" 'common-lisp-user) => CAR, :INHERITED
(find-symbol "CAR" 'common-lisp) => CAR, :EXTERNAL
(find-symbol "NIL" 'common-lisp-user) => NIL, :INHERITED
(find-symbol "NIL" 'common-lisp) => NIL, :EXTERNAL
(find-symbol "NIL" (prog1 (make-package "JUST-TESTING" :use '())
                           (intern "NIL" "JUST-TESTING"))))
=> JUST-TESTING::NIL, :INTERNAL
(export 'just-testing::nil 'just-testing)
(find-symbol "NIL" 'just-testing) => JUST-TESTING:Nil, :EXTERNAL
(find-symbol "NIL" "KEYWORD")
=> NIL, NIL
OR=> :NIL, :EXTERNAL
(find-symbol (symbol-name :nil) "KEYWORD") => :NIL, :EXTERNAL
```

**Side Effects:** None.

**Affected By:**

intern, import, export, use-package, unintern, unexport, unuse-package

**Exceptional Situations:** None.

**See Also:**

**intern, find-all-symbols****Notes:**

**find-symbol** is operationally equivalent to **intern**, except that it never creates a new *symbol*.

**Function FINISH-OUTPUT, FORCE-OUTPUT, CLEAR-OUTPUT****Syntax:**

**finish-output** &*optional output-stream => nil*

**force-output** &*optional output-stream => nil*

**clear-output** &*optional output-stream => nil*

**Arguments and Values:**

*output-stream*—an *output stream designator*. The default is *standard output*.

**Description:**

**finish-output**, **force-output**, and **clear-output** exercise control over the internal handling of buffered stream output.

**finish-output** attempts to ensure that any buffered output sent to *output-stream* has reached its destination, and then returns.

**force-output** initiates the emptying of any internal buffers but does not wait for completion or acknowledgment to return.

**clear-output** attempts to abort any outstanding output operation in progress in order to allow as little output as possible to continue to the destination.

If any of these operations does not make sense for *output-stream*, then it does nothing. The precise actions of these *functions* are *implementation-dependent*.

**Examples:**

```
; ; Implementation A
(progn (princ "am i seen?") (clear-output))
=> NIL

; ; Implementation B
(progn (princ "am i seen?") (clear-output))
>> am i seen?
=> NIL
```

**Side Effects:** None.

**Affected By:**

**\*standard-output\***

**Exceptional Situations:**

Should signal an error of **type-type-error** if *output-stream* is not a **stream designator**.

**See Also:**

**clear-input**

**Notes:** None.

## **Accessor FIRST, SECOND, THIRD, FOURTH, FIFTH, SIXTH, SEVENTH, EIGHTH, NINTH, TENTH**

**Syntax:**

**first** *list* => *object*

**second** *list* => *object*

**third** *list* => *object*

**fourth** *list* => *object*

**fifth** *list* => *object*

**sixth** *list* => *object*

**seventh** *list* => *object*

**eighth** *list* => *object*

**ninth** *list* => *object*

**tenth** *list* => *object*

(**setf** (**first** *list*) *new-object*)

## CLHS: Declaration DYNAMIC-EXTENT

```
(setf (second list) new-object)  
  
(setf (third list) new-object)  
  
(setf (fourth list) new-object)  
  
(setf (fifth list) new-object)  
  
(setf (sixth list) new-object)  
  
(setf (seventh list) new-object)  
  
(setf (eighth list) new-object)  
  
(setf (ninth list) new-object)  
  
(setf (tenth list) new-object)
```

### Arguments and Values:

*list*—a list, which might be a dotted list or a circular list.

*object, new-object*—an *object*.

### Description:

The functions **first**, **second**, **third**, **fourth**, **fifth**, **sixth**, **seventh**, **eighth**, **ninth**, and **tenth** access the first, second, third, fourth, fifth, sixth, seventh, eighth, ninth, and tenth elements of *list*, respectively. Specifically,

```
(first list)    ==  (car list)  
(second list)  ==  (car (cdr list))  
(third list)   ==  (car (caddr list))  
(fourth list)  ==  (car (cdddr list))  
(fifth list)   ==  (car (cddddr list))  
(sixth list)   ==  (car (cdr (cddddr list)))  
(seventh list) ==  (car (caddr (cddddr list)))  
(eighth list)  ==  (car (cdddr (cddddr list)))  
(ninth list)   ==  (car (cddddr (cddddr list)))  
(tenth list)   ==  (car (cdr (cddddr (cddddr list)))))
```

**setf** can also be used with any of these functions to change an existing component. The same equivalences apply. For example:

```
(setf (fifth list) new-object) == (setf (car (cddddr list)) new-object)
```

### 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
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(setf (fourth lst) "four") => "four"  
lst => (1 2 3 "four" ((V)) VI 7 8 9 10)
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[car](#), [nth](#)

**Notes:**

[first](#) is functionally equivalent to [car](#), [second](#) is functionally equivalent to [cadr](#), [third](#) is functionally equivalent to [caddr](#), and [fourth](#) is functionally equivalent to [cadddr](#).

The ordinal numbering used here is one-origin, as opposed to the zero-origin numbering used by [nth](#):

```
(fifth x) == (nth 4 x)
```

## Function FLOAT

**Syntax:**

**float** *number &optional prototype => float*

**Arguments and Values:**

*number*—a [real](#).

*prototype*—a [float](#).

*float*—a [float](#).

**Description:**

[float](#) converts a [real](#) number to a [float](#).

If a *prototype* is supplied, a [float](#) is returned that is mathematically equal to *number* but has the same format as *prototype*.

If *prototype* is not supplied, then if the *number* is already a [float](#), it is returned; otherwise, a [float](#) is returned that is mathematically equal to *number* but is a [single float](#).

**Examples:**

```
(float 0) => 0.0  
(float 1 .5) => 1.0  
(float 1.0) => 1.0  
(float 1/2) => 0.5
```

```
=> 1.0d0
OR=> 1.0
(eql (float 1.0 1.0d0) 1.0d0) => true
```

**Side Effects:** None.**Affected By:** None.**Exceptional Situations:** None.**See Also:**[coerce](#)**Notes:** None.

## **Function FLOATP**

**Syntax:****floatp** *object*

generalized-boolean

**Arguments and Values:***object*—an [object](#).generalized-boolean—a generalized boolean.**Description:**Returns true if *object* is of type float; otherwise, returns false.**Examples:**

```
(floatp 1.2d2) => true
(floatp 1.212) => true
(floatp 1.2s2) => true
(floatp (expt 2 130)) => false
```

**Side Effects:** None.**Affected By:** None.**Exceptional Situations:** None.**See Also:** None.**Notes:**(floatp *object*) == (typep *object* 'float)

## **Function FLOOR, FFLOOR, CEILING, FCEILING, TRUNCATE, FTRUNCATE, ROUND, FROUND**

### Syntax:

**floor** *number* &optional *divisor* => *quotient, remainder*  
**ffloor** *number* &optional *divisor* => *quotient, remainder*  
**ceiling** *number* &optional *divisor* => *quotient, remainder*  
**fceiling** *number* &optional *divisor* => *quotient, remainder*  
**truncate** *number* &optional *divisor* => *quotient, remainder*  
**ftruncate** *number* &optional *divisor* => *quotient, remainder*  
**round** *number* &optional *divisor* => *quotient, remainder*  
**fround** *number* &optional *divisor* => *quotient, remainder*

### Arguments and Values:

*number*---a real.

*divisor*---a non-zero real. The default is the integer 1.

*quotient*---for **floor**, **ceiling**, **truncate**, and **round**: an integer; for **ffloor**, **fceiling**, **ftruncate**, and **fround**: a float.

*remainder*---a real.

### Description:

These functions divide *number* by *divisor*, returning a *quotient* and *remainder*, such that

$$\textit{quotient} * \textit{divisor} + \textit{remainder} = \textit{number}$$

The *quotient* always represents a mathematical integer. When more than one mathematical integer might be possible (i.e., when the remainder is not zero), the kind of rounding or truncation depends on the operator:

#### **floor**, **ffloor**

**floor** and **ffloor** produce a *quotient* that has been truncated toward negative infinity; that is, the *quotient* represents the largest mathematical integer that is not larger than the mathematical quotient.

#### **ceiling**, **fceiling**

**ceiling** and **fceiling** produce a *quotient* that has been truncated toward positive infinity; that is, the *quotient* represents the smallest mathematical integer that is not smaller than the mathematical result.

#### **truncate**, **ftruncate**

**truncate** and **ftruncate** produce a *quotient* that has been truncated towards zero; that is, the *quotient* represents the mathematical integer of the same sign as the mathematical quotient, and that has the

## CLHS: Declaration DYNAMIC-EXTENT

greatest integral magnitude not greater than that of the mathematical quotient.

### **round, fround**

**round** and **fround** produce a *quotient* that has been rounded to the nearest mathematical integer; if the mathematical quotient is exactly halfway between two integers, (that is, it has the form *integer*+1/2), then the *quotient* has been rounded to the even (divisible by two) integer.

All of these functions perform type conversion operations on *numbers*.

The *remainder* is an integer if both *x* and *y* are integers, is a rational if both *x* and *y* are rationals, and is a float if either *x* or *y* is a float.

**ffloor**, **fceiling**, **ftruncate**, and **fround** handle arguments of different *types* in the following way: If *number* is a float, and *divisor* is not a float of longer format, then the first result is a float of the same *type* as *number*. Otherwise, the first result is of the *type* determined by contagion rules; see [Section 12.1.1.2 \(Contagion in Numeric Operations\)](#).

### Examples:

```
(floor 3/2) => 1, 1/2
(ceiling 3 2) => 2, -1
(ffloor 3 2) => 1.0, 1
(ffloor -4.7) => -5.0, 0.3
(ffloor 3.5d0) => 3.0d0, 0.5d0
(fceiling 3/2) => 2.0, -1/2
(ftruncate 1) => 1, 0
(ftruncate .5) => 0, 0.5
(round .5) => 0, 0.5
(ftruncate -7 2) => -3.0, -1
(fround -7 2) => -4.0, 1
(dolist (n '(2.6 2.5 2.4 0.7 0.3 -0.3 -0.7 -2.4 -2.5 -2.6))
  (format t "~&~4,1@F ~2,' D ~2,' D ~2,' D"
    n (floor n) (ceiling n) (truncate n) (round n)))
=> +2.6 2 3 2 3
=> +2.5 2 3 2 2
=> +2.4 2 3 2 2
=> +0.7 0 1 0 1
=> +0.3 0 1 0 0
=> -0.3 -1 0 0 0
=> -0.7 -1 0 0 -1
=> -2.4 -3 -2 -2 -2
=> -2.5 -3 -2 -2 -2
=> -2.6 -3 -2 -2 -3
=> NIL
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:** None.

### Notes:

## CLHS: Declaration DYNAMIC-EXTENT

When only *number* is given, the two results are exact; the mathematical sum of the two results is always equal to the mathematical value of *number*.

(*function number divisor*) and (*function (/ number divisor)*) (where *function* is any of one of **floor**, **ceiling**, **ffloor**, **fceiling**, **truncate**, **round**, **ftruncate**, and **fround**) return the same first value, but they return different remainders as the second value. For example:

```
(floor 5 2) => 2, 1
(floor (/ 5 2)) => 2, 1/2
```

If an effect is desired that is similar to **round**, but that always rounds up or down (rather than toward the nearest even integer) if the mathematical quotient is exactly halfway between two integers, the programmer should consider a construction such as (*floor (+ x 1/2)*) or (*ceiling (- x 1/2)*).

## Function FMAKUNBOUND

**Syntax:**

**fmakunbound** *name* => *name*

**Pronunciation:**

[,ef'makuhn,band] or [,ef'maykuhn,band]

**Arguments and Values:**

*name*—a *function name*.

**Description:**

Removes the *function* or *macro* definition, if any, of *name* in the *global environment*.

**Examples:**

```
(defun add-some (x) (+ x 19)) => ADD-SOME
(fboundp 'add-some) => true
(flet ((add-some (x) (+ x 37)))
  (fmakunbound 'add-some)
  (add-some 1)) => 38
(fboundp 'add-some) => false
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:**

Should signal an error of *type type-error* if *name* is not a *function name*.

The consequences are undefined if *name* is a *special operator*.

**See Also:**

**fboundp, makunbound**

**Notes:** None.

## **Standard Generic Function FUNCTION-KEYWORDS**

### Syntax:

**function-keywords** *method* => *keys, allow-other-keys-p*

### Method Signatures:

**function-keywords** (*method* standard-method)

### Arguments and Values:

*method*—a method.

*keys*—a list.

*allow-other-keys-p*—a generalized boolean.

### Description:

Returns the keyword parameter specifiers for a *method*.

Two values are returned: a list of the explicitly named keywords and a generalized boolean that states whether &*allow-other-keys* had been specified in the *method* definition.

### Examples:

```
(defmethod gf1 ((a integer) &optional (b 2)
              &key (c 3) ((:dee d) 4) e ((eff f)))
  (list a b c d e f))
=> #<STANDARD-METHOD GF1 (INTEGER) 36324653>
(find-method #'gf1 '() (list (find-class 'integer)))
=> #<STANDARD-METHOD GF1 (INTEGER) 36324653>
(function-keywords *)
=> (:C :DEE :E EFF), false
(defmethod gf2 ((a integer))
  (list a b c d e f))
=> #<STANDARD-METHOD GF2 (INTEGER) 42701775>
(function-keywords (find-method #'gf1 '() (list (find-class 'integer))))
=> (), false
(defmethod gf3 ((a integer) &key b c d &allow-other-keys)
  (list a b c d e f))
(function-keywords *)
=> (:B :C :D), true
```

### Affected By:

#### **defmethod**

**Exceptional Situations:** None.

**See Also:**

**defmethod**

**Notes:** None.

## **Function FUNCTION-LAMBDA-EXPRESSION**

**Syntax:**

**function-lambda-expression** *function*

=> *lambda-expression*, *closure-p*, *name*

**Arguments and Values:**

*function*—a *function*.

*lambda-expression*—a *lambda expression* or **nil**.

*closure-p*—a *generalized boolean*.

*name*—an *object*.

**Description:**

Returns information about *function* as follows:

The primary value, *lambda-expression*, is *function*'s defining *lambda expression*, or **nil** if the information is not available. The lambda expression may have been pre-processed in some ways, but it should remain a suitable argument to **compile** or **function**. Any implementation may legitimately return **nil** as the *lambda-expression* of any *function*.

The secondary value, *closure-p*, is **nil** if *function*'s definition was enclosed in the null lexical environment or something non-nil if *function*'s definition might have been enclosed in some non-null lexical environment. Any implementation may legitimately return **true** as the *closure-p* of any *function*.

The tertiary value, *name*, is the "name" of *function*. The name is intended for debugging only and is not necessarily one that would be valid for use as a name in **defun** or **function**, for example. By convention, **nil** is used to mean that *function* has no name. Any implementation may legitimately return **nil** as the *name* of any *function*.

**Examples:**

The following examples illustrate some possible return values, but are not intended to be exhaustive:

```
(function-lambda-expression #'(lambda (x) x))
=> NIL, false, NIL
OR=> NIL, true, NIL
```

## CLHS: Declaration DYNAMIC-EXTENT

```
OR=> (LAMBDA (X) X), true, NIL
OR=> (LAMBDA (X) X), false, NIL

(function-lambda-expression
  (funcall #'(lambda () #'(lambda (x) x))))
=> NIL, false, NIL
OR=> NIL, true, NIL
OR=> (LAMBDA (X) X), true, NIL
OR=> (LAMBDA (X) X), false, NIL

(function-lambda-expression
  (funcall #'(lambda (x) #'(lambda () x)) nil))
=> NIL, true, NIL
OR=> (LAMBDA () X), true, NIL
NOT=> NIL, false, NIL
NOT=> (LAMBDA () X), false, NIL

(flet ((foo (x) x))
  (setf (symbol-function 'bar) #'foo)
  (function-lambda-expression #'bar))
=> NIL, false, NIL
OR=> NIL, true, NIL
OR=> (LAMBDA (X) (BLOCK FOO X)), true, NIL
OR=> (LAMBDA (X) (BLOCK FOO X)), false, FOO
OR=> (SI::BLOCK-LAMBDA FOO (X) X), false, FOO

(defun foo ()
  (flet ((bar (x) x))
    #'bar))
  (function-lambda-expression (foo)))
=> NIL, false, NIL
OR=> NIL, true, NIL
OR=> (LAMBDA (X) (BLOCK BAR X)), true, NIL
OR=> (LAMBDA (X) (BLOCK BAR X)), true, (:INTERNAL FOO 0 BAR)
OR=> (LAMBDA (X) (BLOCK BAR X)), false, "BAR in FOO"
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:** None.

**Notes:**

Although *implementations* are free to return "nil, true, nil" in all cases, they are encouraged to return a *lambda expression* as the *primary value* in the case where the argument was created by a call to compile or eval (as opposed to being created by *loading a compiled file*).

## Function FUNCTIONP

**Syntax:**

**functionp** *object* => *generalized-boolean*

**Arguments and Values:**

*object*—an object.

*generalized-boolean*—a generalized boolean.

**Description:**

Returns true if *object* is of type function; otherwise, returns false.

**Examples:**

```
(functionp 'append) => false
(functionp #'append) => true
(functionp (symbol-function 'append)) => true
(flet ((f () 1)) (functionp #'f)) => true
(functionp (compile nil '(lambda () 259))) => true
(functionp nil) => false
(functionp 12) => false
(functionp '(* x x)) => false
(functionp #'(lambda (x) (* x x))) => true
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:** None.

**Notes:**

```
(functionp object) == (typep object 'function)
```

**Function FORMAT****Syntax:**

**format** *destination control-string &rest args => result*

**Arguments and Values:**

*destination*—nil, t, a stream, or a string with a fill pointer.

*control-string*—a format control.

*args*—format arguments for *control-string*.

*result*—if *destination* is non-nil, then nil; otherwise, a string.

**Description:**

## CLHS: Declaration DYNAMIC-EXTENT

**format** produces formatted output by outputting the characters of *control-string* and observing that a *tilde* introduces a directive. The character after the tilde, possibly preceded by prefix parameters and modifiers, specifies what kind of formatting is desired. Most directives use one or more elements of *args* to create their output.

If *destination* is a *string*, a *stream*, or *t*, then the *result* is *nil*. Otherwise, the *result* is a *string* containing the `output.'

**format** is useful for producing nicely formatted text, producing good-looking messages, and so on. **format** can generate and return a *string* or output to *destination*.

For details on how the *control-string* is interpreted, see [Section 22.3 \(Formatted Output\)](#).

**Examples:** None.

**Affected By:**

[\*\*\\*standard-output\\*\*\*](#), [\*\*\\*print-escape\\*\*\*](#), [\*\*\\*print-radix\\*\*\*](#), [\*\*\\*print-base\\*\*\*](#), [\*\*\\*print-circle\\*\*\*](#), [\*\*\\*print-pretty\\*\*\*](#), [\*\*\\*print-level\\*\*\*](#), [\*\*\\*print-length\\*\*\*](#), [\*\*\\*print-case\\*\*\*](#), [\*\*\\*print-gensym\\*\*\*](#), [\*\*\\*print-array\\*\*\*](#).

**Exceptional Situations:**

If *destination* is a *string* with a *fill pointer*, the consequences are undefined if destructive modifications are performed directly on the *string* during the *dynamic extent* of the call.

**See Also:**

[\*\*write\*\*](#), [Section 13.1.10 \(Documentation of Implementation-Defined Scripts\)](#)

**Notes:** None.

## Function FUNCALL

**Syntax:**

**funcall** *function* &*rest args* => *result*\*

**Arguments and Values:**

*function*—a *function designator*.

*args*—*arguments* to the *function*.

*results*—the *values* returned by the *function*.

**Description:**

**funcall** applies *function* to *args*. If *function* is a *symbol*, it is coerced to a *function* as if by finding its *functional value* in the *global environment*.

**Examples:**

## CLHS: Declaration DYNAMIC-EXTENT

```
(funcall #'+ 1 2 3) => 6
(funcall 'car '(1 2 3)) => 1
(funcall 'position 1 '(1 2 3 2 1) :start 1) => 4
(cons 1 2) => (1 . 2)
(flet ((cons (x y) `(kons ,x ,y)))
  (let ((cons (symbol-function '+)))
    (funcall #'cons
              (funcall 'cons 1 2)
              (funcall cons 1 2))))
=> (KONS (1 . 2) 3)
```

**Affected By:** None.

### Exceptional Situations:

An error of type undefined-function should be signaled if *function* is a symbol that does not have a global definition as a function or that has a global definition as a macro or a special operator.

### See Also:

apply, function, Section 3.1 (Evaluation)

### Notes:

```
(funcall function arg1 arg2 ...)
== (apply function arg1 arg2 ... nil)
== (apply function (list arg1 arg2 ...))
```

The difference between funcall and an ordinary function call is that in the former case the *function* is obtained by ordinary evaluation of a form, and in the latter case it is obtained by the special interpretation of the function position that normally occurs.

## Function GCD

### Syntax:

gcd &*rest integers* => *greatest-common-denominator*

### Arguments and Values:

*integer*---an integer.

*greatest-common-denominator*---a non-negative integer.

### Description:

Returns the greatest common divisor of *integers*. If only one *integer* is supplied, its absolute value is returned. If no *integers* are given, gcd returns 0, which is an identity for this operation.

### Examples:

```
(gcd) => 0
(gcd 60 42) => 6
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(gcd 3333 -33 101) => 1
(gcd 3333 -33 1002001) => 11
(gcd 91 -49) => 7
(gcd 63 -42 35) => 7
(gcd 5) => 5
(gcd -4) => 4
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:**

Should signal an error of type type-error if any integer is not an integer.

**See Also:**

[lcm](#)

**Notes:**

For three or more arguments,

```
(gcd b c ... z) == (gcd (gcd a b) c ... z)
```

## Function GENSYM

**Syntax:**

**gensym** &*optional x* => *new-symbol*

**Arguments and Values:**

*x*—*a string* or a non-negative integer. Complicated defaulting behavior; see below.

*new-symbol*—*a fresh, uninterned symbol*.

**Description:**

Creates and returns a fresh, uninterned symbol, as if by calling make-symbol. (The only difference between **gensym** and **make-symbol** is in how the *new-symbol*'s name is determined.)

The name of the *new-symbol* is the concatenation of a prefix, which defaults to "G", and a suffix, which is the decimal representation of a number that defaults to the value of **\*gensym-counter\***.

If *x* is supplied, and is a string, then that string is used as a prefix instead of "G" for this call to **gensym** only.

If *x* is supplied, and is an integer, then that integer, instead of the value of **\*gensym-counter\***, is used as the suffix for this call to **gensym** only.

If and only if no explicit suffix is supplied, **\*gensym-counter\*** is incremented after it is used.

**Examples:**

```
(setq sym1 (gensym)) => #:G3142
(symbol-package sym1) => NIL
(setq sym2 (gensym 100)) => #:G100
(setq sym3 (gensym 100)) => #:G100
(eq sym2 sym3) => false
(find-symbol "G100") => NIL, NIL
(gensym "T") => #:T3143
(gensym) => #:G3144
```

**Side Effects:**

Might increment \*gensym-counter\*.

**Affected By:**

\*gensym-counter\*

**Exceptional Situations:**

Should signal an error of type type-error if *x* is not a string or a non-negative integer.

**See Also:**

gentemp, \*gensym-counter\*

**Notes:**

The ability to pass a numeric argument to gensym has been deprecated; explicitly binding \*gensym-counter\* is now stylistically preferred. (The somewhat baroque conventions for the optional argument are historical in nature, and supported primarily for compatibility with older dialects of Lisp. In modern code, it is recommended that the only kind of argument used be a string prefix. In general, though, to obtain more flexible control of the *new-symbol*'s name, consider using make-symbol instead.)

## Function GENTEMP

**Syntax:**

**gentemp** &*optional prefix package* => *new-symbol*

**Arguments and Values:**

*prefix*---a string. The default is "T".

*package*---a package designator. The default is the current package.

*new-symbol*---a fresh, interned symbol.

**Description:**

**gentemp** creates and returns a *fresh symbol, interned* in the indicated *package*. The *symbol* is guaranteed to be one that was not previously *accessible* in *package*. It is neither *bound* nor *fbound*, and has a *null property list*.

The *name* of the *new-symbol* is the concatenation of the *prefix* and a suffix, which is taken from an internal counter used only by **gentemp**. (If a *symbol* by that name is already *accessible* in *package*, the counter is incremented as many times as is necessary to produce a *name* that is not already the *name* of a *symbol accessible* in *package*.)

### Examples:

```
(gentemp) => T1298
(gentemp "FOO") => FOO1299
(find-symbol "FOO1300") => NIL, NIL
(gentemp "FOO") => FOO1300
(find-symbol "FOO1300") => FOO1300, :INTERNAL
(intern "FOO1301") => FOO1301, :INTERNAL
(gentemp "FOO") => FOO1302
(gentemp) => T1303
```

### Side Effects:

Its internal counter is incremented one or more times.

*Interns* the *new-symbol* in *package*.

### Affected By:

The current state of its internal counter, and the current state of the *package*.

### Exceptional Situations:

Should signal an error of *type type-error* if *prefix* is not a *string*. Should signal an error of *type type-error* if *package* is not a *package designator*.

### See Also:

**gensym**

### Notes:

The function **gentemp** is deprecated.

If *package* is the KEYWORD package, the result is an *external symbol* of *package*. Otherwise, the result is an *internal symbol* of *package*.

The **gentemp** internal counter is independent of **\*gensym-counter\***, the counter used by **gensym**. There is no provision for accessing the **gentemp** internal counter.

Just because **gentemp** creates a *symbol* which did not previously exist does not mean that such a *symbol* might not be seen in the future (e.g., in a data file---perhaps even created by the same program in another session). As such, this symbol is not truly unique in the same sense as a **gensym** would be. In particular, programs which do automatic code generation should be careful not to attach global attributes to such generated *symbols*.

## CLHS: Declaration DYNAMIC-EXTENT

(e.g., special declarations) and then write them into a file because such global attributes might, in a different session, end up applying to other symbols that were automatically generated on another day for some other purpose.

### Accessor GET

#### Syntax:

```
get symbol indicator &optional default => value  
(setf (get symbol indicator &optional default) new-value)
```

#### Arguments and Values:

*symbol*—a symbol.

*indicator*—an object.

*default*—an object. The default is nil.

*value*—if the indicated property exists, the object that is its value; otherwise, the specified *default*.

*new-value*—an object.

#### Description:

**get** finds a property on the property list[2] of *symbol* whose property indicator is identical to *indicator*, and returns its corresponding property value. If there are multiple properties[1] with that property indicator, **get** uses the first such property. If there is no property with that property indicator, *default* is returned.

**setf** of **get** may be used to associate a new object with an existing indicator already on the *symbol's* property list, or to create a new association if none exists. If there are multiple properties[1] with that property indicator, **setf** of **get** associates the *new-value* with the first such property. When a get form is used as a **setf place**, any *default* which is supplied is evaluated according to normal left-to-right evaluation rules, but its value is ignored.

#### Examples:

```
(defun make-person (first-name last-name)  
  (let ((person (gensym "PERSON")))  
    (setf (get person 'first-name) first-name)  
    (setf (get person 'last-name) last-name)  
    person)) => MAKE-PERSON  
(defvar *john* (make-person "John" "Dow")) => *JOHN*  
*john* => #:PERSON4603  
(defvar *sally* (make-person "Sally" "Jones")) => *SALLY*  
(get *john* 'first-name) => "John"  
(get *sally* 'last-name) => "Jones"  
(defun marry (man woman married-name)  
  (setf (get man 'wife) woman)  
  (setf (get woman 'husband) man)  
  (setf (get man 'last-name) married-name)  
  (setf (get woman 'last-name) married-name))
```

## CLHS: Declaration DYNAMIC-EXTENT

```
married-name) => MARRY
(marry *john* *sally* "Dow-Jones") => "Dow-Jones"
(get *john* 'last-name) => "Dow-Jones"
(get (get *john* 'wife) 'first-name) => "Sally"
(symbol-plist *john*)
=> (WIFE #:PERSON4604 LAST-NAME "Dow-Jones" FIRST-NAME "John")
(defmacro age (person &optional (default "thirty-something"))
  `(get ,person 'age ,default)) => AGE
(age *john*) => THIRTY-SOMETHING
(age *john* 20) => 20
(setq (age *john*) 25) => 25
(age *john*) => 25
(age *john* 20) => 25
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:**

Should signal an error of type type-error if *symbol* is not a symbol.

**See Also:**

getf, symbol-plist, remprop

**Notes:**

```
(get x y) == (getf (symbol-plist x) y)
```

Numbers and characters are not recommended for use as *indicators* in portable code since get tests with eq rather than eql, and consequently the effect of using such *indicators* is implementation-dependent.

There is no way using get to distinguish an absent property from one whose value is *default*. However, see get-properties.

## Function GET-INTERNAL-RUN-TIME

**Syntax:**

**get-internal-run-time** <no arguments> => *internal-time*

**Arguments and Values:**

*internal-time*—a non-negative integer.

**Description:**

Returns as an integer the current run time in internal time units. The precise meaning of this quantity is implementation-defined; it may measure real time, run time, CPU cycles, or some other quantity. The intent is that the difference between the values of two calls to this function be the amount of time between the two calls during which computational effort was expended on behalf of the executing program.

**Examples:** None.**Side Effects:** None.**Affected By:**The *implementation*, the time of day (i.e., the passage of time).**Exceptional Situations:** None.**See Also:****internal-time-units-per-second****Notes:**

Depending on the *implementation*, paging time and garbage collection time might be included in this measurement. Also, in a multitasking environment, it might not be possible to show the time for just the running process, so in some *implementations*, time taken by other processes during the same time interval might be included in this measurement as well.

**Function GET-INTERNAL-REAL-TIME****Syntax:****get-internal-real-time** <no arguments> => *internal-time***Arguments and Values:***internal-time*—a non-negative *integer*.**Description:**

**get-internal-real-time** returns as an *integer* the current time in *internal time units*, relative to an arbitrary time base. The difference between the values of two calls to this function is the amount of elapsed real time (i.e., clock time) between the two calls.

**Examples:** None.**Side Effects:** None.**Affected By:**

Time of day (i.e., the passage of time). The time base affects the result magnitude.

**Exceptional Situations:** None.**See Also:****internal-time-units-per-second**

**Notes:** None.

## Function GET-OUTPUT-STREAM-STRING

**Syntax:**

**get-output-stream-string** *string-output-stream => string*

**Arguments and Values:**

*string-output-stream*—a stream.

*string*—a string.

**Description:**

Returns a string containing, in order, all the characters that have been output to *string-output-stream*. This operation clears any characters on *string-output-stream*, so the *string* contains only those characters which have been output since the last call to **get-output-stream-string** or since the creation of the *string-output-stream*, whichever occurred most recently.

**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) => ""
```

**Side Effects:**

The *string-output-stream* is cleared.

**Affected By:** None.

**Exceptional Situations:**

The consequences are undefined if *string-output-string* is closed.

The consequences are undefined if *string-output-stream* is a stream that was not produced by **make-string-output-stream**. The consequences are undefined if *string-output-stream* was created implicitly by **with-output-to-string** or **format**.

**See Also:**

**make-string-output-stream**

**Notes:** None.

## Function GET-PROPERTIES

### Syntax:

**get-properties** *plist indicator-list => indicator, value, tail*

### Arguments and Values:

*plist*—a property list.

*indicator-list*—a proper list (of indicators).

*indicator*—an object that is an element of *indicator-list*.

*value*—an object.

*tail*—a list.

### Description:

**get-properties** is used to look up any of several property list entries all at once.

It searches the *plist* for the first entry whose indicator is identical to one of the objects in *indicator-list*. If such an entry is found, the *indicator* and *value* returned are the property indicator and its associated property value, and the *tail* returned is the tail of the *plist* that begins with the found entry (i.e., whose car is the *indicator*). If no such entry is found, the *indicator*, *value*, and *tail* are all nil.

### Examples:

```
(setq x '()) => NIL
(setq *indicator-list* '(prop1 prop2)) => (PROP1 PROP2)
(getf x 'prop1) => NIL
(setq (getf x 'prop1) 'val1) => VAL1
(eq (getf x 'prop1) 'val1) => true
(get-properties x *indicator-list*) => PROP1, VAL1, (PROP1 VAL1)
x => (PROP1 VAL1)
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

### See Also:

get, getf

**Notes:** None.

## Function GET-SETF-EXPANSION

### Syntax:

**get-setf-expansion** *place* &*optional environment*  
*=> vars, vals, store-vars, writer-form, reader-form*

### Arguments and Values:

*place*—a *place*.

*environment*—an *environment object*.

*vars, vals, store-vars, writer-form, reader-form*—a *setf expansion*.

### Description:

Determines five values constituting the *setf expansion* for *place* in *environment*; see [Section 5.1.1.2 \(Setf Expansions\)](#).

If *environment* is not supplied or **nil**, the environment is the *null lexical environment*.

### Examples:

```
(get-setf-expansion 'x)
=> NIL, NIL, (:G0001), (SETQ X #:G0001), X

;; This macro is like POP

(defmacro xpop (place &environment env)
  (multiple-value-bind (dummies vals new setter getter)
      (get-setf-expansion place env)
    `(let* (,@(mapcar #'list dummies vals) (,(car new) ,getter))
       (if (cdr new) (error "Can't expand this."))
       (prog1 (car ,(car new))
              (setq ,(car new) (cdr ,(car new)))
              ,setter)))))

(defsetf frob (x) (value)
  `(setf (car ,x) ,value)) => FROB
;; The following is an error; an error might be signaled at macro expansion time
(flet ((frob (x) (cdr x))) ;Invalid
  (xpop (frob z)))
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[\*\*defsetf\*\*](#), [\*\*define-setf-expander\*\*](#), [\*\*setf\*\*](#)

**Notes:**

Any compound form is a valid place, since any compound form whose operator *f* has no setf expander are expanded into a call to (setf *f*).

## **Function GET-UNIVERSAL-TIME, GET-DECODED-TIME**

**Syntax:**

**get-universal-time** <no arguments> => *universal-time*

**get-decoded-time** <no arguments>

=> *second, minute, hour, date, month, year, day, daylight-p, zone*

**Arguments and Values:**

*universal-time*—a universal time.

*second, minute, hour, date, month, year, day, daylight-p, zone*—a decoded time.

**Description:**

**get-universal-time** returns the current time, represented as a universal time.

**get-decoded-time** returns the current time, represented as a decoded time.

**Examples:**

```
; ; At noon on July 4, 1976 in Eastern Daylight Time.  
 (get-decoded-time) => 0, 0, 12, 4, 7, 1976, 6, true, 5  
 ; ; At exactly the same instant.  
 (get-universal-time) => 2414332800  
 ; ; Exactly five minutes later.  
 (get-universal-time) => 2414333100  
 ; ; The difference is 300 seconds (five minutes)  
 (- * **) => 300
```

**Side Effects:** None.

**Affected By:**

The time of day (i.e., the passage of time), the system clock's ability to keep accurate time, and the accuracy of the system clock's initial setting.

**Exceptional Situations:**

An error of type error might be signaled if the current time cannot be determined.

**See Also:**

[decode-universal-time](#), [encode-universal-time](#), Section 25.1.4 (Time)**Notes:**

```
(get-decoded-time) == (decode-universal-time (get-universal-time))
```

No *implementation* is required to have a way to verify that the time returned is correct. However, if an *implementation* provides a validity check (e.g., the failure to have properly initialized the system clock can be reliably detected) and that validity check fails, the *implementation* is strongly encouraged (but not required) to signal an error of *type error* (rather than, for example, returning a known-to-be-wrong value) that is *correctable* by allowing the user to interactively set the correct time.

**Accessor GETF****Syntax:**

**getf** *plist indicator &optional default => value*

(**setf** (**getf** *place indicator &optional default*) *new-value*)

**Arguments and Values:**

*plist*—a *property list*.

*place*—a *place*, the *value* of which is a *property list*.

*indicator*—an *object*.

*default*—an *object*. The default is **nil**.

*value*—an *object*.

*new-value*—an *object*.

**Description:**

**getf** finds a *property* on the *plist* whose *property indicator* is *identical* to *indicator*, and returns its corresponding *property value*. If there are multiple *properties*[1] with that *property indicator*, **getf** uses the first such *property*. If there is no *property* with that *property indicator*, *default* is returned.

**setf** of **getf** may be used to associate a new *object* with an existing indicator in the *property list* held by *place*, or to create a new association if none exists. If there are multiple *properties*[1] with that *property indicator*, **setf** of **getf** associates the *new-value* with the first such *property*. When a **getf form** is used as a **setf place**, any *default* which is supplied is evaluated according to normal left-to-right evaluation rules, but its *value* is ignored.

**setf** of **getf** is permitted to either *write* the *value* of *place* itself, or modify of any part, *car* or *cdr*, of the *list structure* held by *place*.

**Examples:**

## CLHS: Declaration DYNAMIC-EXTENT

```
(setq x '()) => NIL
(getf x 'prop1) => NIL
(getf x 'prop1 7) => 7
(getf x 'prop1) => NIL
(setqf (getf x 'prop1) 'val1) => VAL1
(eq (getf x 'prop1) 'val1) => true
(getf x 'prop1) => VAL1
(getf x 'prop1 7) => VAL1
x => (PROP1 VAL1)

;; Examples of implementation variation permitted.
(setq foo (list 'a 'b 'c 'd 'e 'f)) => (A B C D E F)
(setq bar (cddr foo)) => (C D E F)
(remf foo 'c) => true
foo => (A B E F)
bar
=> (C D E F)
OR=> (C)
OR=> (NIL)
OR=> (C NIL)
OR=> (C D)
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[get](#), [get-properties](#), [setf](#), [Section 5.1.2.2 \(Function Call Forms as Places\)](#)

**Notes:**

There is no way (using [getf](#)) to distinguish an absent property from one whose value is *default*; but see [get-properties](#).

Note that while supplying a *default* argument to [getf](#) in a [setf](#) situation is sometimes not very interesting, it is still important because some macros, such as [push](#) and [incf](#), require a *place* argument which data is both *read* from and *written* to. In such a context, if a *default* argument is to be supplied for the *read* situation, it must be syntactically valid for the *write* situation as well. For example,

```
(let ((plist '()))
  (incf (getf plist 'count 0))
  plist) => (COUNT 1)
```

## Accessor GETHASH

**Syntax:**

**gethash** *key hash-table &optional default => value, present-p*

```
(setf (gethash key hash-table &optional default) new-value)
```

**Arguments and Values:**

Accessor GETHASH

206

*key*---an object.

*hash-table*---a hash table.

*default*---an object. The default is nil.

*value*---an object.

*present-p*---a generalized boolean.

#### Description:

*Value* is the object in *hash-table* whose key is the same as *key* under the *hash-table*'s equivalence test. If there is no such entry, *value* is the *default*.

*Present-p* is true if an entry is found; otherwise, it is false.

**setf** may be used with **gethash** to modify the value associated with a given key, or to add a new entry. When a **gethash form** is used as a **setf place**, any *default* which is supplied is evaluated according to normal left-to-right evaluation rules, but its value is ignored.

#### Examples:

```
(setq table (make-hash-table)) => #<HASH-TABLE EQL 0/120 32206334>
(gethash 1 table) => NIL, false
(gethash 1 table 2) => 2, false
(setf (gethash 1 table) "one") => "one"
(setf (gethash 2 table "two") "two") => "two"
(gethash 1 table) => "one", true
(gethash 2 table) => "two", true
(gethash nil table) => NIL, false
(setf (gethash nil table) nil) => NIL
(gethash nil table) => NIL, true
(defvar *counters* (make-hash-table)) => *COUNTERS*
(gethash 'foo *counters*) => NIL, false
(gethash 'foo *counters* 0) => 0, false
(defmacro how-many (obj) `(values (gethash ,obj *counters* 0))) => HOW-MANY
(defun count-it (obj) (incf (how-many obj))) => COUNT-IT
(dolist (x '(bar foo bar bar baz)) (count-it x))
(how-many 'foo) => 2
(how-many 'bar) => 3
(how-many 'quux) => 0
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

#### See Also:

**remhash**

#### Notes:

## CLHS: Declaration DYNAMIC-EXTENT

The secondary value, *present-p*, can be used to distinguish the absence of an entry from the presence of an entry that has a value of *default*.

### Function GRAPHIC-CHAR-P

#### Syntax:

**graphic-char-p** *char* => *generalized-boolean*

#### Arguments and Values:

*char*---a character.

*generalized-boolean*---a generalized boolean.

#### Description:

Returns true if *character* is a graphic character; otherwise, returns false.

#### Examples:

```
(graphic-char-p #\G) => true
(graphic-char-p #\#) => true
(graphic-char-p #\Space) => true
(graphic-char-p #\Newline) => false
```

**Affected By:** None.

#### Exceptional Situations:

Should signal an error of type type-error if *character* is not a character.

#### See Also:

[read](#), [Section 2.1 \(Character Syntax\)](#), [Section 13.1.10 \(Documentation of Implementation-Defined Scripts\)](#)

**Notes:** None.

### Function HASH-TABLE-COUNT

#### Syntax:

**hash-table-count** *hash-table* => *count*

#### Arguments and Values:

*hash-table*---a hash table.

*count*---a non-negative integer.

#### Description:

## CLHS: Declaration DYNAMIC-EXTENT

Returns the number of entries in the *hash-table*. If *hash-table* has just been created or newly cleared (see clrhash) the entry count is 0.

### Examples:

```
(setq table (make-hash-table)) => #<HASH-TABLE EQL 0/120 32115135>
(hash-table-count table) => 0
(setf (gethash 57 table) "fifty-seven") => "fifty-seven"
(hash-table-count table) => 1
(dotimes (i 100) (setf (gethash i table) i)) => NIL
(hash-table-count table) => 100
```

**Side Effects:** None.

### Affected By:

clrhash, remhash, setf of gethash

**Exceptional Situations:** None.

### See Also:

hash-table-size

### Notes:

The following relationships are functionally correct, although in practice using hash-table-count is probably much faster:

```
(hash-table-count table) ==
(loop for value being the hash-values of table count t) ==
(let ((total 0))
  (maphash #'(lambda (key value)
              (declare (ignore key value))
              (incf total)))
  table)
total)
```

## Function HASH-TABLE-REHASH-SIZE

### Syntax:

**hash-table-rehash-size** *hash-table* => *rehash-size*

### Arguments and Values:

*hash-table*—a hash table.

*rehash-size*—a real of type (or (integer 1 \*) (float (1.0) \*)).

### Description:

## CLHS: Declaration DYNAMIC-EXTENT

Returns the current rehash size of *hash-table*, suitable for use in a call to [make-hash-table](#) in order to produce a *hash table* with state corresponding to the current state of the *hash-table*.

### Examples:

```
(setq table (make-hash-table :size 100 :rehash-size 1.4))
=> #<HASH-TABLE EQL 0/100 2556371>
(hash-table-rehash-size table) => 1.4
```

**Side Effects:** None.

**Affected By:** None.

### Exceptional Situations:

Should signal an error of [type type-error](#) if *hash-table* is not a *hash table*.

### See Also:

[make-hash-table](#), [hash-table-rehash-threshold](#)

### Notes:

If the hash table was created with an [integer](#) rehash size, the result is an [integer](#), indicating that the rate of growth of the *hash-table* when rehashed is intended to be additive; otherwise, the result is a [float](#), indicating that the rate of growth of the *hash-table* when rehashed is intended to be multiplicative. However, this value is only advice to the [implementation](#); the actual amount by which the *hash-table* will grow upon rehash is [implementation-dependent](#).

## Function HASH-TABLE-REHASH-THRESHOLD

### Syntax:

**hash-table-rehash-threshold** *hash-table* => *rehash-threshold*

### Arguments and Values:

*hash-table*—a *hash table*.

*rehash-threshold*—a *real* of [type](#) (*real* 0 1).

### Description:

Returns the current rehash threshold of *hash-table*, which is suitable for use in a call to [make-hash-table](#) in order to produce a *hash table* with state corresponding to the current state of the *hash-table*.

### Examples:

```
(setq table (make-hash-table :size 100 :rehash-threshold 0.5))
=> #<HASH-TABLE EQL 0/100 2562446>
(hash-table-rehash-threshold table) => 0.5
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:**

Should signal an error of type type-error if *hash-table* is not a hash table.

**See Also:**

[make-hash-table](#), [hash-table-rehash-size](#)

**Notes:** None.

## **Function HASH-TABLE-SIZE**

**Syntax:**

**hash-table-size** *hash-table* => *size*

**Arguments and Values:**

*hash-table*—a hash table.

*size*—a non-negative integer.

**Description:**

Returns the current size of *hash-table*, which is suitable for use in a call to [make-hash-table](#) in order to produce a hash table with state corresponding to the current state of the *hash-table*.

**Examples:** None.

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:**

Should signal an error of type type-error if *hash-table* is not a hash table.

**See Also:**

[hash-table-count](#), [make-hash-table](#)

**Notes:** None.

## **Function HASH-TABLE-TEST**

**Syntax:**

**hash-table-test** *hash-table => test***Arguments and Values:***hash-table*—a hash table.

*test*—a function designator. For the four standardized hash table test functions (see **make-hash-table**), the *test* value returned is always a symbol. If an implementation permits additional tests, it is implementation-dependent whether such tests are returned as function objects or function names.

**Description:**

Returns the test used for comparing keys in *hash-table*.

**Examples:** None.**Side Effects:** None.**Affected By:** None.**Exceptional Situations:**

Should signal an error of type type-error if *hash-table* is not a hash table.

**See Also:****make-hash-table****Notes:** None.**Function HASH-TABLE-P****Syntax:****hash-table-p** *object => generalized-boolean***Arguments and Values:***object*—an object.*generalized-boolean*—a generalized boolean.**Description:**

Returns true if *object* is of type hash-table; otherwise, returns false.

**Examples:**

```
(setq table (make-hash-table)) => #<HASH-TABLE EQL 0/120 32511220>
(hash-table-p table) => true
(hash-table-p 37) => false
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(hash-table-p '((a . 1) (b . 2))) => false
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:** None.

**Notes:**

```
(hash-table-p object) == (typep object 'hash-table)
```

## Function IDENTITY

**Syntax:**

**identity** *object* => *object*

**Arguments and Values:**

*object*—an object.

**Description:**

Returns its argument *object*.

**Examples:**

```
(identity 101) => 101
(mapcan #'identity (list (list 1 2 3) '(4 5 6))) => (1 2 3 4 5 6)
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:** None.

**Notes:**

**identity** is intended for use with functions that require a *function* as an argument.

(*eq1 x (identity x)*) returns *true* for all possible values of *x*, but (*eq x (identity x)*) might return *false* when *x* is a *number* or *character*.

**identity** could be defined by

```
(defun identity (x) x)
```

## Function IMPORT

### Syntax:

```
import symbols &optional package => t
```

### Arguments and Values:

*symbols*—a designator for a list of symbols.

*package*—a package designator. The default is the current package.

### Description:

**import** adds *symbol* or *symbols* to the internals of *package*, checking for name conflicts with existing *symbols* either present in *package* or accessible to it. Once the *symbols* have been *imported*, they may be referenced in the *importing package* without the use of a package prefix when using the Lisp reader.

A name conflict in **import** between the *symbol* being imported and a symbol inherited from some other *package* can be resolved in favor of the *symbol* being *imported* by making it a shadowing symbol, or in favor of the *symbol* already accessible by not doing the **import**. A name conflict in **import** with a *symbol* already present in the *package* may be resolved by uninterning that *symbol*, or by not doing the **import**.

The imported *symbol* is not automatically exported from the current package, but if it is already present and external, then the fact that it is external is not changed. If any *symbol* to be *imported* has no home package (i.e., (*symbol-package symbol*) => nil), **import** sets the home package of the *symbol* to *package*.

If the *symbol* is already present in the importing *package*, **import** has no effect.

### Examples:

```
(import 'common-lisp::car (make-package 'temp :use nil)) => T
(find-symbol "CAR" 'temp) => CAR, :INTERNAL
(find-symbol "CDR" 'temp) => NIL, NIL
```

The form (**import** 'editor:buffer) takes the external symbol named *buffer* in the EDITOR package (this symbol was located when the form was read by the Lisp reader) and adds it to the current package as an internal symbol. The symbol *buffer* is then present in the current package.

### Side Effects:

The package system is modified.

### Affected By:

Current state of the package system.

### Exceptional Situations:

**import** signals a correctable error of type package-error if any of the *symbols* to be *imported* has the same name (under string=) as some distinct *symbol* (under eql) already accessible in the *package*, even if the

conflict is with a shadowing symbol of the *package*.

**See Also:**

**shadow, export**

**Notes:** None.

## **Function INPUT-STREAM-P, OUTPUT-STREAM-P**

**Syntax:**

**input-stream-p** *stream* => *generalized-boolean*

**output-stream-p** *stream* => *generalized-boolean*

**Arguments and Values:**

*stream*—a stream.

*generalized-boolean*—a generalized boolean.

**Description:**

**input-stream-p** returns true if *stream* is an input stream; otherwise, returns false.

**output-stream-p** returns true if *stream* is an output stream; otherwise, returns false.

**Examples:**

```
(input-stream-p *standard-input*) => true
(input-stream-p *terminal-io*) => true
(input-stream-p (make-string-output-stream)) => false

(output-stream-p *standard-output*) => true
(output-stream-p *terminal-io*) => true
(output-stream-p (make-string-input-stream "jr")) => false
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:**

Should signal an error of type type-error if *stream* is not a stream.

**See Also:** None.

**Notes:** None.

**Standard Generic Function INITIALIZE-INSTANCE****Syntax:**

**initialize-instance** *instance* &*rest initargs* &*key allow-other-keys* => *instance*

**Method Signatures:**

**initialize-instance** (*instance standard-object*) &*rest initargs*

**Arguments and Values:**

*instance*—an *object*.

*initargs*—a *defaulted initialization argument list*.

**Description:**

Called by **make-instance** to initialize a newly created *instance*. The generic function is called with the new *instance* and the *defaulted initialization argument list*.

The system-supplied primary *method* on **initialize-instance** initializes the *slots* of the *instance* with values according to the *initargs* and the :*initform* forms of the *slots*. It does this by calling the generic function **shared-initialize** with the following arguments: the *instance*, **t** (this indicates that all *slots* for which no initialization arguments are provided should be initialized according to their :*initform* forms), and the *initargs*.

Programmers can define *methods* for **initialize-instance** to specify actions to be taken when an instance is initialized. If only *after methods* are defined, they will be run after the system-supplied primary *method* for initialization and therefore will not interfere with the default behavior of **initialize-instance**.

**Examples:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**shared-initialize**, **make-instance**, **slot-boundp**, **slot-makunbound**, [Section 7.1 \(Object Creation and Initialization\)](#), [Section 7.1.4 \(Rules for Initialization Arguments\)](#), [Section 7.1.2 \(Declaring the Validity of Initialization Arguments\)](#)

**Notes:** None.

**Function INSPECT****Syntax:**

**inspect** *object* => *implementation-dependent*

**Arguments and Values:**

*object*—an object.

**Description:**

**inspect** is an interactive version of **describe**. The nature of the interaction is *implementation-dependent*, but the purpose of **inspect** is to make it easy to wander through a data structure, examining and modifying parts of it.

**Examples:** None.

**Side Effects:**

*implementation-dependent*.

**Affected By:**

*implementation-dependent*.

**Exceptional Situations:**

*implementation-dependent*.

**See Also:**

**describe**

**Notes:**

Implementations are encouraged to respond to the typing of ? or a "help key" by providing help, including a list of commands.

## **Function INTEGERP**

**Syntax:**

**integerp** *object* => *generalized-boolean*

**Arguments and Values:**

*object*—an object.

*generalized-boolean*—a generalized boolean.

**Description:**

Returns true if *object* is of type integer; otherwise, returns false.

**Examples:**

## CLHS: Declaration DYNAMIC-EXTENT

```
(integerp 1) => true
(integerp (expt 2 130)) => true
(integerp 6/5) => false
(integerp nil) => false
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:** None.

**Notes:**

```
(integerp object) == (typep object 'integer)
```

## Function INTEGER-LENGTH

**Syntax:**

**integer-length** *integer* => *number-of-bits*

**Arguments and Values:**

*integer*---an integer.

*number-of-bits*---a non-negative integer.

**Description:**

Returns the number of bits needed to represent *integer* in binary two's-complement format.

**Examples:**

```
(integer-length 0) => 0
(integer-length 1) => 1
(integer-length 3) => 2
(integer-length 4) => 3
(integer-length 7) => 3
(integer-length -1) => 0
(integer-length -4) => 2
(integer-length -7) => 3
(integer-length -8) => 3
(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
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:**

Should signal an error of **type type-error** if *integer* is not an *integer*.

**See Also:** None.

**Notes:**

This function could have been defined by:

```
(defun integer-length (integer)
  (ceiling (log (if (minusp integer)
                     (- integer)
                     (+ integer))
                2)))
```

If *integer* is non-negative, then its value can be represented in unsigned binary form in a field whose width in bits is no smaller than (*integer-length integer*). Regardless of the sign of *integer*, its value can be represented in signed binary two's-complement form in a field whose width in bits is no smaller than (+ (*integer-length integer*) 1).

**Function INTERACTIVE-STREAM-P****Syntax:**

**interactive-stream-p** *stream* => *generalized-boolean*

**Arguments and Values:**

*stream*—a *stream*.

*generalized-boolean*—a *generalized boolean*.

**Description:**

Returns *true* if *stream* is an *interactive stream*; otherwise, returns *false*.

**Examples:**

```
(when (> measured limit)
  (let ((error (round (* (- measured limit) 100)
                      limit)))
    (unless (if (interactive-stream-p *query-io*)
                (yes-or-no-p "The frammis is out of tolerance by ~D%.~@"
                            "Is it safe to proceed? " error)
                (< error 15)) ;15% is acceptable
      (error "The frammis is out of tolerance by ~D%." error))))
```

**Affected By:** None.

**Exceptional Situations:**

Should signal an error of **type type-error** if *stream* is not a *stream*.

**See Also:**

**Notes:** None.

## **Function INTERN**

**Syntax:**

**intern** *string* &optional *package* => *symbol*, *status*

**Arguments and Values:**

*string*—a string.

*package*—a package designator. The default is the current package.

*symbol*—a symbol.

*status*—one of :inherited, :external, :internal, or nil.

**Description:**

**intern** enters a symbol named *string* into *package*. If a symbol whose name is the same as *string* is already accessible in *package*, it is returned. If no such symbol is accessible in *package*, a new symbol with the given name is created and entered into *package* as an internal symbol, or as an external symbol if the *package* is the KEYWORD package; *package* becomes the home package of the created symbol.

The first value returned by **intern**, *symbol*, is the symbol that was found or created. The meaning of the secondary value, *status*, is as follows:

:internal

The symbol was found and is present in *package* as an internal symbol.

:external

The symbol was found and is present as an external symbol.

:inherited

The symbol was found and is inherited via use-package (which implies that the symbol is internal).

nil

No pre-existing symbol was found, so one was created.

It is implementation-dependent whether the string that becomes the new symbol's name is the given *string* or a copy of it. Once a string has been given as the string argument to **intern** in this situation where a new symbol is created, the consequences are undefined if a subsequent attempt is made to alter that string.

**Examples:**

```
(in-package "COMMON-LISP-USER") => #<PACKAGE "COMMON-LISP-USER">
(intern "Never-Before") => |Never-Before|, NIL
(intern "Never-Before") => |Never-Before|, :INTERNAL
(intern "NEVER-BEFORE" "KEYWORD") => :NEVER-BEFORE, NIL
(intern "NEVER-BEFORE" "KEYWORD") => :NEVER-BEFORE, :EXTERNAL
```

**Side Effects:** None.**Affected By:** None.**Exceptional Situations:** None.**See Also:**[find-symbol](#), [read](#), [symbol](#), [unintern](#), [Section 2.3.4 \(Symbols as Tokens\)](#)**Notes:**

[intern](#) does not need to do any name conflict checking because it never creates a new [symbol](#) if there is already an [accessible symbol](#) with the name given.

## Function INVALID-METHOD-ERROR

**Syntax:**

**invalid-method-error** *method format-control &rest args => implementation-dependent*

**Arguments and Values:**

*method*—[a method](#).

*format-control*—[a format control](#).

*args*—[format arguments](#) for the *format-control*.

**Description:**

The [function invalid-method-error](#) is used to signal an error of [type error](#) when there is an applicable [method](#) whose [qualifiers](#) are not valid for the method combination type. The error message is constructed by using the [format-control](#) suitable for [format](#) and any *args* to it. Because an implementation may need to add additional contextual information to the error message, [invalid-method-error](#) should be called only within the dynamic extent of a method combination function.

The [function invalid-method-error](#) is called automatically when a [method](#) fails to satisfy every [qualifier](#) pattern and predicate in a [define-method-combination form](#). A method combination function that imposes additional restrictions should call [invalid-method-error](#) explicitly if it encounters a [method](#) it cannot accept.

Whether [invalid-method-error](#) returns to its caller or exits via [throw](#) is [implementation-dependent](#).

**Examples:** None.**Side Effects:**

The debugger might be entered.

**Affected By:**

**\*break-on-signals\***

**Exceptional Situations:** None.

**See Also:**

**define-method-combination**

**Notes:** None.

***Function INVOKE-RESTART***

**Syntax:**

**invoke-restart** *restart &rest arguments => result\**

**Arguments and Values:**

*restart*—a *restart designator*.

*argument*—an *object*.

*results*—the *values* returned by the *function* associated with *restart*, if that *function* returns.

**Description:**

Calls the *function* associated with *restart*, passing *arguments* to it. *Restart* must be valid in the current *dynamic environment*.

**Examples:**

```
(defun add3 (x) (check-type x number) (+ x 3))

(foo 'seven)
>> Error: The value SEVEN was not of type NUMBER.
>> To continue, type :CONTINUE followed by an option number:
>> 1: Specify a different value to use.
>> 2: Return to Lisp Toplevel.
>> Debug> (invoke-restart 'store-value 7)
=> 10
```

**Side Effects:**

A non-local transfer of control might be done by the restart.

**Affected By:**

Existing restarts.

**Exceptional Situations:**

If *restart* is not valid, an error of *type control-error* is signaled.

**See Also:**

[find-restart](#), [restart-bind](#), [restart-case](#), [invoke-restart-interactively](#)

**Notes:**

The most common use for [invoke-restart](#) is in a [handler](#). It might be used explicitly, or implicitly through [invoke-restart-interactively](#) or a [restart function](#).

[Restart functions](#) call [invoke-restart](#), not vice versa. That is, [invoke-restart](#) provides primitive functionality, and [restart functions](#) are non-essential "syntactic sugar."

## Function INVOKE-RESTART-INTERACTIVELY

**Syntax:**

**invoke-restart-interactively** *restart => result\**

**Arguments and Values:**

*restart*—a [restart designator](#).

*results*—the [values](#) returned by the [function](#) associated with *restart*, if that [function](#) returns.

**Description:**

[invoke-restart-interactively](#) calls the [function](#) associated with *restart*, prompting for any necessary arguments. If *restart* is a name, it must be valid in the current [dynamic environment](#).

[invoke-restart-interactively](#) prompts for arguments by executing the code provided in the :interactive keyword to [restart-case](#) or :interactive-function keyword to [restart-bind](#).

If no such options have been supplied in the corresponding [restart-bind](#) or [restart-case](#), then the consequences are undefined if the *restart* takes required arguments. If the arguments are optional, an argument list of [nil](#) is used.

Once the arguments have been determined, [invoke-restart-interactively](#) executes the following:

```
(apply #'invoke-restart restart arguments)
```

**Examples:**

```
(defun add3 (x) (check-type x number) (+ x 3))

(add3 'seven)
>> Error: The value SEVEN was not of type NUMBER.
>> To continue, type :CONTINUE followed by an option number:
>> 1: Specify a different value to use.
>> 2: Return to Lisp Toplevel.
>> Debug> (invoke-restart-interactively 'store-value)
>> Type a form to evaluate and use: 7
=> 10
```

**Side Effects:**

If prompting for arguments is necessary, some typeout may occur (on *query I/O*).

A non-local transfer of control might be done by the restart.

**Affected By:**

**\*query-io\***, active *restarts*

**Exceptional Situations:**

If *restart* is not valid, an error of ***type control-error*** is signaled.

**See Also:**

**find-restart**, **invoke-restart**, **restart-case**, **restart-bind**

**Notes:**

**invoke-restart-interactively** is used internally by the debugger and may also be useful in implementing other portable, interactive debugging tools.

## **Function INVOKE-DEBUGGER**

**Syntax:**

**invoke-debugger** *condition =>*

**Arguments and Values:**

*condition*—a *condition object*.

**Description:**

**invoke-debugger** attempts to enter the debugger with *condition*.

If **\*debugger-hook\*** is not **nil**, it should be a *function* (or the name of a *function*) to be called prior to entry to the standard debugger. The *function* is called with **\*debugger-hook\*** bound to **nil**, and the *function* must accept two arguments: the *condition* and the *value* of **\*debugger-hook\*** prior to binding it to **nil**. If the *function* returns normally, the standard debugger is entered.

The standard debugger never directly returns. Return can occur only by a non-local transfer of control, such as the use of a restart function.

**Examples:**

```
(ignore-errors :Normally, this would suppress debugger entry
  (handler-bind ((error #'invoke-debugger)) ;But this forces debugger entry
    (error "Foo.")))
Debug: Foo.
```

## CLHS: Declaration DYNAMIC-EXTENT

To continue, type :CONTINUE followed by an option number:  
1: Return to Lisp Toplevel.  
Debug>

### Side Effects:

**\*debugger-hook\*** is bound to **nil**, program execution is discontinued, and the debugger is entered.

### Affected By:

**\*debug-io\*** and **\*debugger-hook\***.

**Exceptional Situations:** None.

### See Also:

**error**, **break**

**Notes:** None.

## ***Function INTERSECTION, NINTERSECTION***

### Syntax:

**intersection** *list-1* *list-2* &**key** *key test test-not => result-list*

**nintersection** *list-1* *list-2* &**key** *key test test-not => result-list*

### Arguments and Values:

*list-1*---a **proper list**.

*list-2*---a **proper list**.

*test*---a **designator** for a **function** of two **arguments** that returns a **generalized boolean**.

*test-not*---a **designator** for a **function** of two **arguments** that returns a **generalized boolean**.

*key*---a **designator** for a **function** of one argument, or **nil**.

*result-list*---a **list**.

### Description:

**intersection** and **nintersection** return a **list** that contains every element that occurs in both *list-1* and *list-2*.

**nintersection** is the destructive version of **intersection**. It performs the same operation, but may destroy *list-1* using its cells to construct the result. *list-2* is not destroyed.

## CLHS: Declaration DYNAMIC-EXTENT

The intersection operation is described as follows. For all possible ordered pairs consisting of one element from *list-1* and one element from *list-2*, :test or :test-not are used to determine whether they satisfy the test. The first argument to the :test or :test-not function is an element of *list-1*; the second argument is an element of *list-2*. If :test or :test-not is not supplied, eql is used. It is an error if :test and :test-not are supplied in the same function call.

If :key is supplied (and not nil), it is used to extract the part to be tested from the *list* element. The argument to the :key function is an element of either *list-1* or *list-2*; the :key function typically returns part of the supplied element. If :key is not supplied or nil, the *list-1* and *list-2* elements are used.

For every pair that *satisfies the test*, exactly one of the two elements of the pair will be put in the result. No element from either *list* appears in the result that does not satisfy the test for an element from the other *list*. If one of the lists contains duplicate elements, there may be duplication in the result.

There is no guarantee that the order of elements in the result will reflect the ordering of the arguments in any particular way. The result *list* may share cells with, or be eq to, either *list-1* or *list-2* if appropriate.

### Examples:

```
(setq list1 (list 1 1 2 3 4 a b c "A" "B" "C" "d")
      list2 (list 1 4 5 b c d "a" "B" "c" "D"))
=> (1 4 5 B C D "a" "B" "c" "D")
(intersection list1 list2) => (C B 4 1 1)
(intersection list1 list2 :test 'equal) => ("B" C B 4 1 1)
(intersection list1 list2 :test #'equalp) => ("d" "C" "B" "A" C B 4 1 1)
(nintersection list1 list2) => (1 1 4 B C)
list1 => implementation-dependent ;e.g., (1 1 4 B C)
list2 => implementation-dependent ;e.g., (1 4 5 B C D "a" "B" "c" "D")
(setq list1 (copy-list '((1 . 2) (2 . 3) (3 . 4) (4 . 5))))
=> ((1 . 2) (2 . 3) (3 . 4) (4 . 5))
(setq list2 (copy-list '((1 . 3) (2 . 4) (3 . 6) (4 . 8))))
=> ((1 . 3) (2 . 4) (3 . 6) (4 . 8))
(nintersection list1 list2 :key #'cdr) => ((2 . 3) (3 . 4))
list1 => implementation-dependent ;e.g., ((1 . 2) (2 . 3) (3 . 4))
list2 => implementation-dependent ;e.g., ((1 . 3) (2 . 4) (3 . 6) (4 . 8))
```

### Side Effects:

nintersection can modify *list-1*, but not *list-2*.

**Affected By:** None.

### Exceptional Situations:

Should be prepared to signal an error of type type-error if *list-1* and *list-2* are not proper lists.

### See Also:

union, Section 3.2.1 (Compiler Terminology), Section 3.6 (Traversal Rules and Side Effects)

### Notes:

The :test-not parameter is deprecated.

***Function KEYWORDP*****Syntax:**

**keywordp** *object* => *generalized-boolean*

**Arguments and Values:**

*object*—an *object*.

*generalized-boolean*—a *generalized boolean*.

**Description:**

Returns *true* if *object* is a *keyword*[1]; otherwise, returns *false*.

**Examples:**

```
(keywordp 'elephant) => false
(keywordp 12) => false
(keywordp :test) => true
(keywordp ':test) => true
(keywordp nil) => false
(keywordp :nil) => true
(keywordp '(:test)) => false
(keywordp "hello") => false
(keywordp ":hello") => false
(keywordp '&optional) => false
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**constantp**, **keyword**, **symbolp**, **symbol-package**

**Notes:** None.

***Function LAST*****Syntax:**

**last** *list* &*optional n* => *tail*

**Arguments and Values:**

*list*—a *list*, which might be a *dotted list* but must not be a *circular list*.

*n*—a non-negative *integer*. The default is 1.

*tail*—an object.

### Description:

**last** returns the last *n conses* (not the last *n* elements) of *list*). If *list* is ( ), **last** returns ( ).

If *n* is zero, the atom that terminates *list* is returned. If *n* is greater than or equal to the number of cons cells in *list*, the result is *list*.

### Examples:

```
(last nil) => NIL
(last '(1 2 3)) => (3)
(last '(1 2 . 3)) => (2 . 3)
(setq x (list 'a 'b 'c 'd)) => (A B C D)
(last x) => (D)
(rplacd (last x) (list 'e 'f)) x => (A B C D E F)
(last x) => (F)

(last '(a b c)) => (C)

(last '(a b c) 0) => ()
(last '(a b c) 1) => (C)
(last '(a b c) 2) => (B C)
(last '(a b c) 3) => (A B C)
(last '(a b c) 4) => (A B C)

(last '(a . b) 0) => B
(last '(a . b) 1) => (A . B)
(last '(a . b) 2) => (A . B)
```

**Side Effects:** None.

**Affected By:** None.

### Exceptional Situations:

The consequences are undefined if *list* is a circular list. Should signal an error of type type–error if *n* is not a non-negative integer.

### See Also:

**butlast, nth**

### Notes:

The following code could be used to define **last**.

```
(defun last (list &optional (n 1))
  (check-type n (integer 0))
  (do ((l list (cdr l))
        (r list)
        (i 0 (+ i 1)))
      ((atom l) r)
      (if (>= i n) (pop r))))
```

## **Function LCM**

### **Syntax:**

**lcm** &rest integers => least-common-multiple

### **Arguments and Values:**

integer---an integer.

least-common-multiple---a non-negative integer.

### **Description:**

**lcm** returns the least common multiple of the *integers*.

If no *integer* is supplied, the integer 1 is returned.

If only one *integer* is supplied, the absolute value of that *integer* is returned.

For two arguments that are not both zero,

```
(lcm a b) == (/ (abs (* a b)) (gcd a b))
```

If one or both arguments are zero,

```
(lcm a 0) == (lcm 0 a) == 0
```

For three or more arguments,

```
(lcm a b c ... z) == (lcm (lcm a b) c ... z)
```

### **Examples:**

```
(lcm 10) => 10
(lcm 25 30) => 150
(lcm -24 18 10) => 360
(lcm 14 35) => 70
(lcm 0 5) => 0
(lcm 1 2 3 4 5 6) => 60
```

**Side Effects:** None.

**Affected By:** None.

### **Exceptional Situations:**

Should signal type-error if any argument is not an integer.

### **See Also:**

gcd

**Notes:** None.

## Function LOAD–LOGICAL–PATHNAME–TRANSLATIONS

**Syntax:**

**load–logical–pathname–translations** *host => just-loaded*

**Arguments and Values:**

*host*—a string.

*just-loaded*—a generalized boolean.

**Description:**

Searches for and loads the definition of a logical host named *host*, if it is not already defined. The specific nature of the search is implementation-defined.

If the *host* is already defined, no attempt to find or load a definition is attempted, and false is returned. If the *host* is not already defined, but a definition is successfully found and loaded, true is returned. Otherwise, an error is signaled.

**Examples:**

```
(translate-logical-pathname "hacks:weather;barometer.lisp.newest")
>> Error: The logical host HACKS is not defined.
(load-logical-pathname-translations "HACKS")
>> ; Loading SYS:SITE;HACKS.TRANSLATIONS
>> ; Loading done.
=> true
(translate-logical-pathname "hacks:weather;barometer.lisp.newest")
=> #P"HELIUM:[SHARED.HACKS.WEATHER]BAROMETER.LSP;0"
(load-logical-pathname-translations "HACKS")
=> false
```

**Affected By:** None.

**Exceptional Situations:**

If no definition is found, an error of type error is signaled.

**See Also:**

**logical–pathname**

**Notes:**

Logical pathname definitions will be created not just by implementors but also by programmers. As such, it is important that the search strategy be documented. For example, an implementation might define that the definition of a *host* is to be found in a file called "*host.translations*" in some specifically named directory.

## Accessor LDB

### Syntax:

```
ldb bytespec integer => byte
(setf (ldb bytespec place) new-byte)
```

### Pronunciation:

[lidib] or [liduhb] or ['el'dee'bee]

### Arguments and Values:

*bytespec*—a byte specifier.

*integer*—an integer.

*byte*, *new-byte*—a non-negative integer.

### Description:

**ldb** extracts and returns the byte of *integer* specified by *bytespec*.

**ldb** returns an integer in which the bits with weights  $2^{(s-1)}$  through  $2^0$  are the same as those in *integer* with weights  $2^{(p+s-1)}$  through  $2^p$ , and all other bits zero; *s* is (byte-size *bytespec*) and *p* is (byte-position *bytespec*).

**setf** may be used with **ldb** to modify a byte within the *integer* that is stored in a given *place*. The order of evaluation, when an **ldb** form is supplied to **setf**, is exactly left-to-right. The effect is to perform a dpb operation and then store the result back into the *place*.

### Examples:

```
(ldb (byte 2 1) 10) => 1
(setq a (list 8)) => (8)
(setf (ldb (byte 2 1) (car a)) 1) => 1
a => (10)
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

### See Also:

byte, byte-position, byte-size, dpb

### Notes:

## CLHS: Declaration DYNAMIC-EXTENT

```
(logbitp j (ldb (byte s p) n))
  ==  (and (< j s) (logbitp (+ j p) n))
```

In general,

```
(ldb (byte 0 x) y) => 0
```

for all valid values of *x* and *y*.

Historically, the name "ldb" comes from a DEC PDP-10 assembly language instruction meaning "load byte."

## **Function LDB-TEST**

### **Syntax:**

**ldb-test** *bytespec integer* => *generalized-boolean*

### **Arguments and Values:**

*bytespec*—a byte specifier.

*integer*—an integer.

*generalized-boolean*—a generalized boolean.

### **Description:**

Returns true if any of the bits of the byte in *integer* specified by *bytespec* is non-zero; otherwise returns false.

### **Examples:**

```
(ldb-test (byte 4 1) 16) => true
(ldb-test (byte 3 1) 16) => false
(ldb-test (byte 3 2) 16) => true
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[byte](#), [ldb](#), [zerop](#)

### **Notes:**

```
(ldb-test bytespec n) ==
(not (zerop (ldb bytespec n))) ==
(logtest (ldb bytespec -1) n)
```

**Function LDIFF, TAILP****Syntax:****ldiff** *list object => result-list***tailp** *object list => generalized-boolean***Arguments and Values:***list*—a *list*, which might be a dotted list.*object*—an *object*.*result-list*—a *list*.*generalized-boolean*—a *generalized boolean*.**Description:**If *object* is the same as some *tail* of *list*, **tailp** returns *true*; otherwise, it returns *false*.If *object* is the same as some *tail* of *list*, **ldiff** returns a *fresh list* of the *elements* of *list* that precede *object* in the *list structure* of *list*; otherwise, it returns a *copy*[2] of *list*.**Examples:**

```
(let ((lists '#((a b c) (a b c . d))))
  (dotimes (i (length lists)) ())
    (let ((list (aref lists i)))
      (format t "~2&list=~S ~21T(tailp object list)~"
              ~44T(ldiff list object)~%" list)
      (let ((objects (vector list (cddr list) (copy-list (cddr list))
                             '(f g h) '() 'd 'x)))
        (dotimes (j (length objects)) ()
          (let ((object (aref objects j)))
            (format t "~& object=~S ~21T~S ~44T~S"
                    object (tailp object list) (ldiff list object)))))))
>>
>> list=(A B C)           (tailp object list)   (ldiff list object)
>> object=(A B C)         T                      NIL
>> object=(C)             T                      (A B)
>> object=(C)             NIL                     (A B C)
>> object=(F G H)         NIL                     (A B C)
>> object=NIL             T                      (A B C)
>> object=D               NIL                     (A B C)
>> object=X               NIL                     (A B C)
>>
>> list=(A B C . D)       (tailp object list)   (ldiff list object)
>> object=(A B C . D)     T                      NIL
>> object=(C . D)         T                      (A B)
>> object=(C . D)         NIL                     (A B C . D)
>> object=(F G H)         NIL                     (A B C . D)
>> object=NIL             NIL                     (A B C . D)
>> object=D               T                      (A B C)
```

## CLHS: Declaration DYNAMIC-EXTENT

```
>> object=x           NIL          (A B C . D)
=> NIL
```

### Side Effects:

Neither **ldiff** nor **tailp** modifies either of its *arguments*.

**Affected By:** None.

### Exceptional Situations:

Should be prepared to signal an error of *type type-error* if *list* is not a *proper list* or a *dotted list*.

### See Also:

**set-difference**

### Notes:

If the *list* is a *circular list*, **tailp** will reliably *yield* a *value* only if the given *object* is in fact a *tail* of *list*. Otherwise, the consequences are unspecified: a given *implementation* which detects the circularity must return *false*, but since an *implementation* is not obliged to detect such a *situation*, **tailp** might just loop indefinitely without returning in that case.

**tailp** could be defined as follows:

```
(defun tailp (object list)
  (do ((list list (cdr list)))
      ((atom list) (eql list object))
      (if (eql object list)
          (return t))))
```

and **ldiff** could be defined by:

```
(defun ldiff (list object)
  (do ((list list (cdr list))
        (r '() (cons (car list) r)))
      ((atom list)
       (if (eql list object) (nreverse r) (nreconc r list)))
      (when (eql object list)
        (return (nreverse r)))))
```

## Function LENGTH

### Syntax:

**length** *sequence* => *n*

### Arguments and Values:

*sequence*—a *proper sequence*.

*n*—a non-negative *integer*.

**Description:**

Returns the number of *elements* in *sequence*.

If *sequence* is a *vector* with a *fill pointer*, the active length as specified by the *fill pointer* is returned.

**Examples:**

```
(length "abc") => 3
(setq str (make-array '(3) :element-type 'character
                      :initial-contents "abc"
                      :fill-pointer t)) => "abc"
(length str) => 3
(setq (fill-pointer str) 2) => 2
(length str) => 2
```

**Affected By:** None.

**Exceptional Situations:**

Should be prepared to signal an error of *type type-error* if *sequence* is not a *proper sequence*.

**See Also:**

[list-length](#), [sequence](#)

**Notes:** None.

## **Function LISP-IMPLEMENTATION-TYPE, LISP-IMPLEMENTATION-VERSION**

**Syntax:**

**lisp-implementation-type** <no arguments> => *description*

**lisp-implementation-version** <no arguments> => *description*

**Arguments and Values:**

*description*—a *string* or *nil*.

**Description:**

**lisp-implementation-type** and **lisp-implementation-version** identify the current implementation of Common Lisp.

**lisp-implementation-type** returns a *string* that identifies the generic name of the particular Common Lisp implementation.

**lisp-implementation-version** returns a *string* that identifies the version of the particular Common Lisp

implementation.

If no appropriate and relevant result can be produced, **nil** is returned instead of a *string*.

**Examples:**

```
(lisp-implementation-type)
=> "ACME Lisp"
OR=> "Joe's Common Lisp"
(lisp-implementation-version)
=> "1.3a"
=> "V2"
OR=> "Release 17.3, ECO #6"
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:** None.

**Notes:** None.

## **Function LIST, LIST\***

**Syntax:**

*list* &rest *objects* => *list*

*list\** &rest *objects+* => *result*

**Arguments and Values:**

*object*---an object.

*list*---a list.

*result*---an object.

**Description:**

**list** returns a list containing the supplied *objects*.

**list\*** is like **list** except that the last argument to **list** becomes the car of the last cons constructed, while the last argument to **list\*** becomes the cdr of the last cons constructed. Hence, any given call to **list\*** always produces one fewer conses than a call to **list** with the same number of arguments.

If the last argument to **list\*** is a list, the effect is to construct a new list which is similar, but which has additional elements added to the front corresponding to the preceding arguments of **list\***.

## CLHS: Declaration DYNAMIC-EXTENT

If list\* receives only one *object*, that *object* is returned, regardless of whether or not it is a list.

### 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 ;i.e., ()
(setq a '(1 2)) => (1 2)
(eq a (list* a)) => true
(list 3 4 'a (car '(b . c)) (+ 6 -2)) => (3 4 A B 4)
(list* 'a 'b 'c 'd) == (cons 'a (cons 'b (cons 'c 'd))) => (A B C . D)
(list* 'a 'b 'c '(d e f)) => (A B C D E F)
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

### See Also:

#### cons

### Notes:

```
(list* x) == x
```

## Function LIST-ALL-PACKAGES

### Syntax:

**list-all-packages** <no arguments> => *packages*

### Arguments and Values:

*packages*—a list of package objects.

### Description:

**list-all-packages** returns a fresh list of all registered packages.

### Examples:

```
(let ((before (list-all-packages)))
  (make-package 'temp)
  (set-difference (list-all-packages) before)) => (#<PACKAGE "TEMP">>)
```

**Side Effects:** None.

**Affected By:****defpackage, delete-package, make-package****Exceptional Situations:** None.**See Also:** None.**Notes:** None.

## **Function LIST-LENGTH**

**Syntax:****list-length** *list* => *length***Arguments and Values:***list*—a proper list or a circular list.*length*—a non-negative integer, or nil.**Description:**Returns the length of *list* if *list* is a proper list. Returns nil if *list* is a circular list.**Examples:**

```
(list-length '(a b c d)) => 4
(list-length '(a (b c) d)) => 3
(list-length '()) => 0
(list-length nil) => 0
(defun circular-list (&rest elements)
  (let ((cycle (copy-list elements)))
    (nconc cycle cycle)))
(list-length (circular-list 'a 'b)) => NIL
(list-length (circular-list 'a)) => NIL
(list-length (circular-list)) => 0
```

**Side Effects:** None.**Affected By:** None.**Exceptional Situations:**Should signal an error of type type-error if *list* is not a proper list or a circular list.**See Also:****length****Notes:**

list-length could be implemented as follows:

```
(defun list-length (x)
  (do ((n 0 (+ n 2))) ;Counter.
       (fast x (cddr fast)) ;Fast pointer: leaps by 2.
       (slow x (cdr slow))) ;Slow pointer: leaps by 1.
    (nil)
    ; If fast pointer hits the end, return the count.
    (when (endp fast) (return n))
    (when (endp (cdr fast)) (return (+ n 1)))
    ; If fast pointer eventually equals slow pointer,
    ; then we must be stuck in a circular list.
    ; (A deeper property is the converse: if we are
    ; stuck in a circular list, then eventually the
    ; fast pointer will equal the slow pointer.
    ; That fact justifies this implementation.)
    (when (and (eq fast slow) (> n 0)) (return nil))))
```

## Function LISTEN

**Syntax:**

**listen** &*optional input-stream => generalized-boolean*

**Arguments and Values:**

*input-stream*—an input stream designator. The default is standard input.

*generalized-boolean*—a generalized boolean.

**Description:**

Returns true if there is a character immediately available from *input-stream*; otherwise, returns false. On a non-interactive *input-stream*, **listen** returns true except when at end of file[1]. If an end of file is encountered, **listen** returns false. **listen** is intended to be used when *input-stream* obtains characters from an interactive device such as a keyboard.

**Examples:**

```
(progn (unread-char (read-char)) (list (listen) (read-char)))
>> 1
=> (T #\1)
  (progn (clear-input) (listen))
=> NIL ;Unless you're a very fast typist!
```

**Side Effects:** None.

**Affected By:**

**\*standard-input\***

**Exceptional Situations:** None.

**See Also:****interactive-stream-p, read-char-no-hang****Notes:** None.

## **Function LISTP**

**Syntax:****listp** *object* => *generalized-boolean***Arguments and Values:***object* --- an *object*.*generalized-boolean* --- a *generalized boolean*.**Description:**Returns *true* if *object* is of *type list*; otherwise, returns *false*.**Examples:**

```
(listp nil) => true
(listp (cons 1 2)) => true
(listp (make-array 6)) => false
(listp t) => false
```

**Side Effects:** None.**Affected By:** None.**Exceptional Situations:** None.**See Also:****consp****Notes:**If *object* is a *cons*, **listp** does not check whether *object* is a *proper list*; it returns *true* for any kind of *list*.

```
(listp object) == (typep object 'list) == (typep object '(or cons null))
```

## **Function LOAD**

**Syntax:****load** *filespec* &*key* *verbose print if-does-not-exist external-format*

=> *generalized-boolean*

### Arguments and Values:

*filespec*—*a stream*, or a *pathname designator*. The default is taken from **\*default-pathname-defaults\***.

*verbose*—*a generalized boolean*. The default is the *value* of **\*load-verbose\***.

*print*—*a generalized boolean*. The default is the *value* of **\*load-print\***.

*if-does-not-exist*—*a generalized boolean*. The default is *true*.

*external-format*—*an external file format designator*. The default is :default.

*generalized-boolean*—*a generalized boolean*.

### Description:

**load** loads the *file* named by *filespec* into the Lisp environment.

The manner in which a *source file* is distinguished from a *compiled file* is *implementation-dependent*. If the file specification is not complete and both a *source file* and a *compiled file* exist which might match, then which of those files **load** selects is *implementation-dependent*.

If *filespec* is a *stream*, **load** determines what kind of *stream* it is and loads directly from the *stream*. If *filespec* is a *logical pathname*, it is translated into a *physical pathname* as if by calling **translate-logical-pathname**.

**load** sequentially executes each *form* it encounters in the *file* named by *filespec*. If the *file* is a *source file* and the *implementation* chooses to perform *implicit compilation*, **load** must recognize *top level forms* as described in Section 3.2.3.1 (Processing of Top Level Forms) and arrange for each *top level form* to be executed before beginning *implicit compilation* of the next. (Note, however, that processing of *eval-when forms* by **load** is controlled by the :execute situation.)

If *verbose* is *true*, **load** prints a message in the form of a comment (i.e., with a leading *semicolon*) to *standard output* indicating what *file* is being *loaded* and other useful information. If *verbose* is *false*, **load** does not print this information.

If *print* is *true*, **load** incrementally prints information to *standard output* showing the progress of the *loading* process. For a *source file*, this information might mean printing the *values* yielded by each *form* in the *file* as soon as those *values* are returned. For a *compiled file*, what is printed might not reflect precisely the contents of the *source file*, but some information is generally printed. If *print* is *false*, **load** does not print this information.

If the file named by *filespec* is successfully loaded, **load** returns *true*.

If the file does not exist, the specific action taken depends on *if-does-not-exist*: if it is **nil**, **load** returns **nil**; otherwise, **load** signals an error.

The *external-format* specifies the *external file format* to be used when opening the *file* (see the *function open*), except that when the *file* named by *filespec* is a *compiled file*, the *external-format* is ignored. **compile-file** and **load** cooperate in an *implementation-dependent* way to assure the preservation of the

## CLHS: Declaration DYNAMIC-EXTENT

similarity of characters referred to in the source file at the time the source file was processed by the file compiler under a given external file format, regardless of the value of external-format at the time the compiled file is loaded.

**load** binds \*readtable\* and \*package\* to the values they held before *loading* the file.

\*load-truename\* is bound by **load** to hold the truename of the pathname of the file being *loaded*.

\*load-pathname\* is bound by **load** to hold a pathname that represents filespec merged against the defaults. That is, (pathname (merge-pathnames filespec)).

### Examples:

```
;Establish a data file...
(with-open-file (str "data.in" :direction :output :if-exists :error)
  (print 1 str) (print '(setq a 888) str) t)
=> T
  (load "data.in") => true
  a => 888
  (load (setq p (merge-pathnames "data.in")) :verbose t)
; Loading contents of file /fred/data.in
; Finished loading /fred/data.in
=> true
  (load p :print t)
; Loading contents of file /fred/data.in
; 1
; 888
; Finished loading /fred/data.in
=> true

;----[Begin file SETUP]----
(in-package "MY-STUFF")
(defmacro compile-truename () `',*compile-file-truename*)
(defvar *my-compile-truename* (compile-truename) "Just for debugging.")
(defvar *my-load-pathname* *load-pathname*)
(defun load-my-system ()
  (dolist (module-name '("FOO" "BAR" "BAZ"))
    (load (merge-pathnames module-name *my-load-pathname*))))
;----[End of file SETUP]----


(load "SETUP")
(load-my-system)
```

### Affected By:

The implementation, and the host computer's file system.

### Exceptional Situations:

If :if-does-not-exist is supplied and is true, or is not supplied, **load** signals an error of type file-error if the file named by filespec does not exist, or if the file system cannot perform the requested operation.

An error of type file-error might be signaled if (wild-pathname-p filespec) returns true.

**See Also:**

[\*\*error\*\*](#), [\*\*merge-pathnames\*\*](#), [\*\*\\*load-verbose\\*\*\*](#), [\*\*\\*default-pathname-defaults\\*\*\*](#), [\*\*pathname\*\*](#),  
[\*\*logical-pathname\*\*](#), [Section 20.1 \(File System Concepts\)](#), [Section 19.1.2 \(Pathnames as Filenames\)](#)

**Notes:** None.

## Function LOG

**Syntax:**

**log** *number* &*optional base* => *logarithm*

**Arguments and Values:**

*number*—a non-zero [number](#).

*base*—a [number](#).

*logarithm*—a [number](#).

**Description:**

**log** returns the logarithm of *number* in base *base*. If *base* is not supplied its value is *e*, the base of the natural logarithms.

**log** may return a [complex](#) when given a [real](#) negative *number*.

```
(log -1.0) == (complex 0.0 (float pi 0.0))
```

If *base* is zero, **log** returns zero.

The result of (log 8 2) may be either 3 or 3.0, depending on the implementation. An implementation can use floating-point calculations even if an exact integer result is possible.

The branch cut for the logarithm function of one argument (natural logarithm) lies along the negative real axis, continuous with quadrant II. The domain excludes the origin.

The mathematical definition of a complex logarithm is as follows, whether or not minus zero is supported by the implementation:

```
(log x) == (complex (log (abs x)) (phase x))
```

Therefore the range of the one-argument logarithm function is that strip of the complex plane containing numbers with imaginary parts between -<PI> (exclusive) and <PI> (inclusive) if minus zero is not supported, or -<PI> (inclusive) and <PI> (inclusive) if minus zero is supported.

The two-argument logarithm function is defined as

```
(log base number)
== (/ (log number) (log base))
```

## CLHS: Declaration DYNAMIC-EXTENT

This defines the principal values precisely. The range of the two-argument logarithm function is the entire complex plane.

### Examples:

```
(log 100 10)
=> 2.0
=> 2
(log 100.0 10) => 2.0
(log #c(0 1) #c(0 -1))
=> #C(-1.0 0.0)
OR=> #C(-1 0)
(log 8.0 2) => 3.0

(log #c(-16 16) #c(2 2)) => 3 or approximately #c(3.0 0.0)
                                or approximately 3.0 (unlikely)
```

### Affected By:

The implementation.

**Exceptional Situations:** None.

### See Also:

[exp, expt, Section 12.1.3.3 \(Rule of Float Substitutability\)](#)

**Notes:** None.

**Function LOGAND, LOGANDC1, LOGANDC2, LOGEQV, LOGIOR, LOGNAND, LOGNOR, LOGNOT, LOGORC1, LOGORC2, LOGXOR**

### Syntax:

**logand** &*rest integers* => *result-integer*

**logandc1** *integer-1 integer-2* => *result-integer*

**logandc2** *integer-1 integer-2* => *result-integer*

**logeqv** &*rest integers* => *result-integer*

## CLHS: Declaration DYNAMIC-EXTENT

**logior** &rest integers => result-integer

**logand** integer-1 integer-2 => result-integer

**lognor** integer-1 integer-2 => result-integer

**lognot** integer => result-integer

**logorc1** integer-1 integer-2 => result-integer

**logorc2** integer-1 integer-2 => result-integer

**logxor** &rest integers => result-integer

### Arguments and Values:

integers---integers.

integer---an integer.

integer-1---an integer.

integer-2---an integer.

result-integer---an integer.

### Description:

The functions **logandc1**, **logandc2**, **logand**, **logeqv**, **logior**, **logand**, **lognor**, **lognot**, **logorc1**, **logorc2**, and **logxor** perform bit-wise logical operations on their arguments, that are treated as if they were binary.

The next figure lists the meaning of each of the functions. Where an 'identity' is shown, it indicates the value yielded by the function when no arguments are supplied.

Function	Identity	Operation performed
<u>logandc1</u>	---	and complement of integer-1 with integer-2
<u>logandc2</u>	---	and integer-1 with complement of integer-2
<u>logand</u>	-1	and
<u>logeqv</u>	-1	equivalence (exclusive nor)
<u>logior</u>	0	inclusive or
<u>lognand</u>	---	complement of integer-1 and integer-2
<u>lognor</u>	---	complement of integer-1 or integer-2
<u>lognot</u>	---	complement
<u>logorc1</u>	---	or complement of integer-1 with integer-2
<u>logorc2</u>	---	or integer-1 with complement of integer-2
<u>logxor</u>	0	exclusive or

### Figure 12–18. Bit-wise Logical Operations on Integers

Negative *integers* are treated as if they were in two's-complement notation.

### Examples:

Function LOGAND, LOGANDC1, LOGANDC2, LOGEQV, LOGIOR, LOGNAND, LOGNOR, LOGNOT, LOGORC1, LOGORC2, LOGXOR

## CLHS: Declaration DYNAMIC-EXTENT

```
(logior 1 2 4 8) => 15
(logxor 1 3 7 15) => 10
(logeqv) => -1
(logand 16 31) => 16
(lognot 0) => -1
(lognot 1) => -2
(lognot -1) => 0
(lognot (1+ (lognot 1000))) => 999

;; In the following example, m is a mask. For each bit in
;; the mask that is a 1, the corresponding bits in x and y are
;; exchanged. For each bit in the mask that is a 0, the
;; corresponding bits of x and y are left unchanged.
(flet ((show (m x y)
           (format t "~%m = #o~6,'0O~%x = #o~6,'0O~%y = #o~6,'0O~%"
                   m x y)))
  (let ((m #o007750)
        (x #o452576)
        (y #o317407))
    (show m x y)
    (let ((z (logand (logxor x y) m)))
      (setq x (logxor z x))
      (setq y (logxor z y))
      (show m x y))))
>> m = #o007750
>> x = #o452576
>> y = #o317407
>>
>> m = #o007750
>> x = #o457426
>> y = #o312557
=> NIL
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:**

Should signal **type-error** if any argument is not an *integer*.

**See Also:**

**boole**

**Notes:**

(logbitp k -1) returns *true* for all values of *k*.

Because the following functions are not associative, they take exactly two arguments rather than any number of arguments.

```
(lognand n1 n2) == (lognot (logand n1 n2))
(lognor n1 n2) == (lognot (logior n1 n2))
(logandc1 n1 n2) == (logand (lognot n1) n2)
(logandc2 n1 n2) == (logand n1 (lognot n2))
(logiorc1 n1 n2) == (logior (lognot n1) n2)
```

Function LOGAND, LOGANDC1, LOGANDC2, LOGEQV, LOGIOR, LOGNAND, LOGNOR, LOGNOT, LOG2NOT, LOG3NOT

## CLHS: Declaration DYNAMIC-EXTENT

```
(logiorc2 n1 n2) == (logior n1 (lognot n2))  
(logbitp j (lognot x)) == (not (logbitp j x))
```

### Function LOGBITP

#### Syntax:

**logbitp** *index integer* => *generalized-boolean*

#### Arguments and Values:

*index*—a non-negative integer.

*integer*—an integer.

*generalized-boolean*—a generalized boolean.

#### Description:

**logbitp** is used to test the value of a particular bit in *integer*, that is treated as if it were binary. The value of **logbitp** is true if the bit in *integer* whose index is *index* (that is, its weight is  $2^{\text{index}}$ ) is a one-bit; otherwise it is false.

Negative *integers* are treated as if they were in two's-complement notation.

#### Examples:

```
(logbitp 1 1) => false  
(logbitp 0 1) => true  
(logbitp 3 10) => true  
(logbitp 1000000 -1) => true  
(logbitp 2 6) => true  
(logbitp 0 6) => false
```

**Side Effects:** None.

**Affected By:** None.

#### Exceptional Situations:

Should signal an error of type type-error if *index* is not a non-negative integer. Should signal an error of type type-error if *integer* is not an integer.

**See Also:** None.

#### Notes:

```
(logbitp k n) == (ldb-test (byte 1 k) n)
```

## Function LOGCOUNT

### Syntax:

**logcount** *integer* => *number-of-on-bits*

### Arguments and Values:

*integer*—an integer.

*number-of-on-bits*—a non-negative integer.

### Description:

Computes and returns the number of bits in the two's-complement binary representation of *integer* that are 'on' or 'set'. If *integer* is negative, the 0 bits are counted; otherwise, the 1 bits are counted.

### Examples:

```
(logcount 0) => 0
(logcount -1) => 0
(logcount 7) => 3
(logcount 13) => 3 ;Two's-complement binary: ...0001101
(logcount -13) => 2 ;Two's-complement binary: ...1110011
(logcount 30) => 4 ;Two's-complement binary: ...0011110
(logcount -30) => 4 ;Two's-complement binary: ...1100010
(logcount (expt 2 100)) => 1
(logcount (- (expt 2 100))) => 100
(logcount (- (1+ (expt 2 100)))) => 1
```

**Side Effects:** None.

**Affected By:** None.

### Exceptional Situations:

Should signal type-error if its argument is not an integer.

**See Also:** None.

### Notes:

Even if the implementation does not represent integers internally in two's complement binary, **logcount** behaves as if it did.

The following identity always holds:

```
(logcount x)
== (logcount (- (+ x 1)))
== (logcount (lognot x))
```

**Function LOGICAL-PATHNAME****Syntax:****logical-pathname** *pathspec => logical-pathname***Arguments and Values:***pathspec*—[a \*logical pathname\*](#), a [logical pathname namestring](#), or a [stream](#).*logical-pathname*—[a \*logical pathname\*](#).**Description:**

**logical-pathname** converts *pathspec* to a [logical pathname](#) and returns the new [logical pathname](#). If *pathspec* is a [logical pathname namestring](#), it should contain a host component and its following [colon](#). If *pathspec* is a [stream](#), it should be one for which [pathname](#) returns a [logical pathname](#).

If *pathspec* is a [stream](#), the [stream](#) can be either open or closed. **logical-pathname** returns the same [logical pathname](#) after a file is closed as it did when the file was open. It is an error if *pathspec* is a [stream](#) that is created with [make-two-way-stream](#), [make-echo-stream](#), [make-broadcast-stream](#), [make-concatenated-stream](#), [make-string-input-stream](#), or [make-string-output-stream](#).

**Examples:** None.**Affected By:** None.**Exceptional Situations:**Signals an error of [type type-error](#) if *pathspec* isn't supplied correctly.**See Also:**[logical-pathname](#), [translate-logical-pathname](#), [Section 19.3 \(Logical Pathnames\)](#)**Notes:** None.**Accessor LOGICAL-PATHNAME-TRANSLATIONS****Syntax:****logical-pathname-translations** *host => translations*( **setf** (**logical-pathname-translations** *host*) *new-translations* )**Arguments and Values:***host*—[a \*logical host designator\*](#).*translations*, *new-translations*—[a \*list\*](#).

**Description:**

Returns the host's *list* of translations. Each translation is a *list* of at least two elements: *from-wildcard* and *to-wildcard*. Any additional elements are *implementation-defined*. *From-wildcard* is a *logical pathname* whose host is *host*. *To-wildcard* is a *pathname*.

(*setf (logical-pathname-translations host) translations*) sets a *logical pathname* host's *list* of *translations*. If *host* is a *string* that has not been previously used as a *logical pathname* host, a new *logical pathname* host is defined; otherwise an existing host's translations are replaced. *logical pathname* host names are compared with *string-equal*.

When setting the translations list, each *from-wildcard* can be a *logical pathname* whose host is *host* or a *logical pathname* namestring parseable by (*parse-namestring string host*), where *host* represents the appropriate *object* as defined by *parse-namestring*. Each *to-wildcard* can be anything coercible to a *pathname* by (*pathname to-wildcard*). If *to-wildcard* coerces to a *logical pathname*, *translate-logical-pathname* will perform repeated translation steps when it uses it.

*host* is either the host component of a *logical pathname* or a *string* that has been defined as a *logical pathname* host name by *setf* of *logical-pathname-translations*.

**Examples:**

```
;;;A very simple example of setting up a logical pathname host. No
;;;translations are necessary to get around file system restrictions, so
;;;all that is necessary is to specify the root of the physical directory
;;;tree that contains the logical file system.
;;;The namestring syntax on the right-hand side is implementation-dependent.
(setf (logical-pathname-translations "foo")
      '((("**;*.*.*"           "MY-LISPM:>library>foo>**>") )))

;;;Sample use of that logical pathname. The return value
;;;is implementation-dependent.
(translate-logical-pathname "foo:bar;baz;mum.quux.3")
=> #P"MY-LISPM:>library>foo>bar>baz>mum.quux.3"

;;;A more complex example, dividing the files among two file servers
;;;and several different directories. This Unix doesn't support
;;;:WILD-INFERIORS in the directory, so each directory level must
;;;be translated individually. No file name or type translations
;;;are required except for .MAIL to .MBX.
;;;The namestring syntax on the right-hand side is implementation-dependent.
(setf (logical-pathname-translations "prog")
      '((("RELEASED;*.*.*"       "MY-UNIX:/sys/bin/my-prog/")
         ("RELEASED;*;*.*.*"     "MY-UNIX:/sys/bin/my-prog/*/")
         ("EXPERIMENTAL;*.*.*"   "MY-UNIX:/usr/Joe/development/prog/")
         ("EXPERIMENTAL;DOCUMENTATION;*.*.*"
          "MY-VAX:SYS$DISK:[JOE.DOC]")
         ("EXPERIMENTAL;*;*.*.*" "MY-UNIX:/usr/Joe/development/prog/*/")
         ("MAIL;**;*.MAIL"        "MY-VAX:SYS$DISK:[JOE.MAIL.PROG...]*.MBX")))

;;;Sample use of that logical pathname. The return value
;;;is implementation-dependent.
(translate-logical-pathname "prog:mail;save;ideas.mail.3")
=> #P"MY-VAX:SYS$DISK:[JOE.MAIL.PROG.SAVE]IDEAS.MBX.3"

;;;Example translations for a program that uses three files main.lisp,
;;;auxiliary.lisp, and documentation.lisp. These translations might be
```

## CLHS: Declaration DYNAMIC-EXTENT

```

;;;supplied by a software supplier as examples.

;;;For Unix with long file names
(setf (logical-pathname-translations "prog")
      '(("CODE;*.*.*"          "/lib/prog/")))

;;;Sample use of that logical pathname.  The return value
;;;is implementation-dependent.
(translate-logical-pathname "prog:code;documentation.lisp")
=> #P"/lib/prog/documentation.lisp"

;;;For Unix with 14-character file names, using .lisp as the type
(setf (logical-pathname-translations "prog")
      '(("CODE;DOCUMENTATION.*.*.*"  "/lib/prog/docum.*")
        ("CODE;*.*.*"           "/lib/prog/")))

;;;Sample use of that logical pathname.  The return value
;;;is implementation-dependent.
(translate-logical-pathname "prog:code;documentation.lisp")
=> #P"/lib/prog/docum.lisp"

;;;For Unix with 14-character file names, using .l as the type
;;;The second translation shortens the compiled file type to .b
(setf (logical-pathname-translations "prog")
      `(("**;*.LISP.*"           ,(logical-pathname "PROG:**;*.L.*"))
        (,(compile-file-pathname (logical-pathname "PROG:**;*.LISP.*"))
         ,(logical-pathname "PROG:**;*.B.*"))
        ("CODE;DOCUMENTATION.*.*.*" "/lib/prog/documentatio.*")
        ("CODE;*.*.*"             "/lib/prog/")))

;;;Sample use of that logical pathname.  The return value
;;;is implementation-dependent.
(translate-logical-pathname "prog:code;documentation.lisp")
=> #P"/lib/prog/documentatio.l"

;;;For a Cray with 6 character names and no directories, types, or versions.
(setf (logical-pathname-translations "prog")
      (let ((l '("MAIN" "PGMN")
                ("AUXILIARY" "PGAUX")
                ("DOCUMENTATION" "PGDOC"))))
        (logpath (logical-pathname "prog:code;"))
        (phypath (pathname "XXX")))
      (append
       ; Translations for source files
       (mapcar #'(lambda (x)
                  (let ((log (first x))
                        (phy (second x)))
                    (list (make-pathname :name log
                                         :type "LISP"
                                         :version :wild
                                         :defaults logpath)
                          (make-pathname :name phy
                                         :defaults phypath))))
       l)
       ; Translations for compiled files
       (mapcar #'(lambda (x)
                  (let* ((log (first x))
                         (phy (second x))
                         (com (compile-file-pathname
                               (make-pathname :name log
                                             :type "LISP"
                                             :version :wild
                                             :defaults logpath)
                               :com t)))))))

```

## CLHS: Declaration DYNAMIC-EXTENT

```
:version :wild
:defaults logpath))))
(setq phy (concatenate 'string phy "B"))
(list com
      (make-pathname :name phy
                     :defaults phypath))))
1)))))

;;; Sample use of that logical pathname.  The return value
;;; is implementation-dependent.
(translate-logical-pathname "prog:code;documentation.lisp")
=> #P"PGDOC"
```

**Affected By:** None.

**Exceptional Situations:**

If *host* is incorrectly supplied, an error of type type-error is signaled.

**See Also:**

[logical-pathname](#), [Section 19.1.2 \(Pathnames as Filenames\)](#)

**Notes:**

Implementations can define additional functions that operate on logical pathname hosts, for example to specify additional translation rules or options.

## Function LOGTEST

**Syntax:**

**logtest** *integer-1* *integer-2* => *generalized-boolean*

**Arguments and Values:**

*integer-1*—an integer.

*integer-2*—an integer.

*generalized-boolean*—a generalized boolean.

**Description:**

Returns true if any of the bits designated by the 1's in *integer-1* is 1 in *integer-2*; otherwise it is false. *integer-1* and *integer-2* are treated as if they were binary.

Negative *integer-1* and *integer-2* are treated as if they were represented in two's-complement binary.

**Examples:**

```
(logtest 1 7) => true
(logtest 1 2) => false
```

```
(logtest -2 -1) => true
(logtest 0 -1) => false
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:**

Should signal an error of type type-error if *integer-1* is not an integer. Should signal an error of type type-error if *integer-2* is not an integer.

**See Also:** None.

**Notes:**

```
(logtest x y) == (not (zerop (logand x y)))
```

## Function MACHINE-INSTANCE

**Syntax:**

**machine-instance** <no arguments> => *description*

**Arguments and Values:**

*description*—a string or nil.

**Description:**

Returns a string that identifies the particular instance of the computer hardware on which Common Lisp is running, or nil if no such string can be computed.

**Examples:**

```
(machine-instance)
=> "ACME.COM"
OR=> "S/N 123231"
OR=> "18.26.0.179"
OR=> "AA-00-04-00-A7-A4"
```

**Side Effects:** None.

**Affected By:**

The machine instance, and the implementation.

**Exceptional Situations:** None.

**See Also:**

**machine-type, machine-version**

**Notes:** None.

## ***Function MACHINE-TYPE***

**Syntax:**

**machine-type** <no arguments> => *description*

**Arguments and Values:**

*description*—a string or nil.

**Description:**

Returns a string that identifies the generic name of the computer hardware on which Common Lisp is running.

**Examples:**

```
(machine-type)
=> "DEC PDP-10"
OR=> "Symbolics LM-2"
```

**Side Effects:** None.

**Affected By:**

The machine type. The implementation.

**Exceptional Situations:** None.

**See Also:**

**machine-version**

**Notes:** None.

## ***Function MACHINE-VERSION***

**Syntax:**

**machine-version** <no arguments> => *description*

**Arguments and Values:**

*description*—a string or nil.

**Description:**

Returns a string that identifies the version of the computer hardware on which Common Lisp is running, or nil if no such value can be computed.

**Examples:**

```
(machine-version) => "KL-10, microcode 9"
```

**Side Effects:** None.**Affected By:**The machine version, and the implementation.**Exceptional Situations:** None.**See Also:****machine-type, machine-instance****Notes:** None.**Accessor MACRO-FUNCTION****Syntax:****macro-function** *symbol* &*optional environment* => *function*(setf (**macro-function** *symbol* &*optional environment*) *new-function*)**Arguments and Values:***symbol*---a symbol.*environment*---an environment object.*function*---a macro function or nil.*new-function*---a macro function.**Description:**Determines whether *symbol* has a function definition as a macro in the specified *environment*.If so, the macro expansion function, a function of two arguments, is returned. If *symbol* has no function definition in the lexical environment *environment*, or its definition is not a macro, **macro-function** returns nil.It is possible for both **macro-function** and **special-operator-p** to return true of *symbol*. The macro definition must be available for use by programs that understand only the standard Common Lisp special forms.**Examples:**

```
(defmacro macfun (x) '(macro-function 'macfun)) => MACFUN
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(not (macro-function 'macfun)) => false

(macrolet ((foo (&environment env)
                 (if (macro-function 'bar env)
                     "yes
                     "no)))
  (list (foo)
        (macrolet ((bar () :beep))
          (foo)))))

=> (NO YES)
```

### Affected By:

(setf macro-function), defmacro, and macrolet.

### Exceptional Situations:

The consequences are undefined if *environment* is non-nil in a use of setf of macro-function.

### See Also:

defmacro, Section 3.1 (Evaluation)

### Notes:

setf can be used with macro-function to install a macro as a symbol's global function definition:

```
(setf (macro-function symbol) fn)
```

The value installed must be a function that accepts two arguments, the entire macro call and an environment, and computes the expansion for that call. Performing this operation causes *symbol* to have only that macro definition as its global function definition; any previous definition, whether as a macro or as a function, is lost.

## Function MAKUNBOUND

### Syntax:

**makunbound** *symbol* => *symbol*

### Arguments and Values:

*symbol*—a symbol

### Description:

Makes the *symbol* be unbound, regardless of whether it was previously bound.

### Examples:

```
(setf (symbol-value 'a) 1)
(boundp 'a) => true
a => 1
```

```
(makunbound 'a) => A
(boundp 'a) => false
```

**Side Effects:**

The value cell of symbol is modified.

**Affected By:** None.

**Exceptional Situations:**

Should signal an error of type type-error if symbol is not a symbol.

**See Also:**

**boundp, fmakunbound**

**Notes:** None.

**Function MAP****Syntax:**

**map** *result-type* *function* &*rest sequences+* => *result*

**Arguments and Values:**

*result-type* --- a sequence type specifier, or nil.

*function* --- a function designator. *function* must take as many arguments as there are *sequences*.

*sequence* --- a proper sequence.

*result* --- if *result-type* is a type specifier other than nil, then a sequence of the type it denotes; otherwise (if the *result-type* is nil), nil.

**Description:**

Applies *function* to successive sets of arguments in which one argument is obtained from each sequence. The *function* is called first on all the elements with index 0, then on all those with index 1, and so on. The *result-type* specifies the type of the resulting sequence.

**map** returns nil if *result-type* is nil. Otherwise, **map** returns a sequence such that element *j* is the result of applying *function* to element *j* of each of the *sequences*. The result sequence is as long as the shortest of the *sequences*. The consequences are undefined if the result of applying *function* to the successive elements of the *sequences* cannot be contained in a sequence of the type given by *result-type*.

If the *result-type* is a subtype of list, the result will be a list.

If the *result-type* is a subtype of vector, then if the implementation can determine the element type specified for the *result-type*, the element type of the resulting array is the result of *upgrading* that element type; or, if

## CLHS: Declaration DYNAMIC-EXTENT

the implementation can determine that the element type is unspecified (or \*), the element type of the resulting array is `t`; otherwise, an error is signaled.

### Examples:

```
(map 'string #'(lambda (x y)
  (char "01234567890ABCDEF" (mod (+ x y) 16)))
  '(1 2 3 4)
  '(10 9 8 7)) => "AAAAA"
(setq seq '("lower" "UPPER" "" "123")) => ("lower" "UPPER" "" "123")
(map nil #'nstring-upcase seq) => NIL
seq => ("LOWER" "UPPER" "" "123")
(map 'list #'-' '(1 2 3 4)) => (-1 -2 -3 -4)
(map 'string
  #'(lambda (x) (if (oddp x) #\1 #\0)))
  '(1 2 3 4)) => "1010"

(map '(vector * 4) #'cons "abc" "de") should signal an error
```

**Affected By:** None.

### Exceptional Situations:

An error of type type-error must be signaled if the *result-type* is not a recognizable subtype of list, not a recognizable subtype of vector, and not nil.

Should be prepared to signal an error of type type-error if any *sequence* is not a proper sequence.

An error of type type-error should be signaled if *result-type* specifies the number of elements and the minimum length of the *sequences* is different from that number.

### See Also:

[Section 3.6 \(Traversal Rules and Side Effects\)](#)

**Notes:** None.

## Function MAP-INTO

### Syntax:

**map-into** *result-sequence* *function* &*rest sequences* => *result-sequence*

### Arguments and Values:

*result-sequence*—a proper sequence.

*function*—a designator for a function of as many arguments as there are *sequences*.

*sequence*—a proper sequence.

### Description:

## CLHS: Declaration DYNAMIC-EXTENT

Destructively modifies *result-sequence* to contain the results of applying *function* to each element in the argument *sequences* in turn.

*result-sequence* and each element of *sequences* can each be either a *list* or a *vector*. If *result-sequence* and each element of *sequences* are not all the same length, the iteration terminates when the shortest *sequence* (of any of the *sequences* or the *result-sequence*) is exhausted. If *result-sequence* is a *vector* with a *fill pointer*, the *fill pointer* is ignored when deciding how many iterations to perform, and afterwards the *fill pointer* is set to the number of times *function* was applied. If *result-sequence* is longer than the shortest element of *sequences*, extra elements at the end of *result-sequence* are left unchanged. If *result-sequence* is *nil*, **map-into** immediately returns *nil*, since *nil* is a *sequence* of length zero.

If *function* has side effects, it can count on being called first on all of the elements with index 0, then on all of those numbered 1, and so on.

### Examples:

```
(setq a (list 1 2 3 4) b (list 10 10 10 10)) => (10 10 10 10)
(map-into a #'+ a b) => (11 12 13 14)
a => (11 12 13 14)
b => (10 10 10 10)
(setq k '(one two three)) => (ONE TWO THREE)
(map-into a #'cons k a) => ((ONE . 11) (TWO . 12) (THREE . 13) 14)
(map-into a #'gensym) => (#:G9090 #:G9091 #:G9092 #:G9093)
a => (#:G9090 #:G9091 #:G9092 #:G9093)
```

**Affected By:** None.

### Exceptional Situations:

Should be prepared to signal an error of *type type-error* if *result-sequence* is not a *proper sequence*. Should be prepared to signal an error of *type type-error* if *sequence* is not a *proper sequence*.

**See Also:** None.

### Notes:

**map-into** differs from **map** in that it modifies an existing *sequence* rather than creating a new one. In addition, **map-into** can be called with only two arguments, while **map** requires at least three arguments.

**map-into** could be defined by:

```
(defun map-into (result-sequence function &rest sequences)
  (loop for index below (apply #'min
                                (length result-sequence)
                                (mapcar #'length sequences)))
        do (setf (elt result-sequence index)
                  (apply function
                         (mapcar #'(lambda (seq) (elt seq index))
                                 sequences))))
  result-sequence)
```

**Function MAPC, MAPCAR, MAPCAN, MAPL, MAPLIST, MAPCON****Syntax:****mapc** *function &rest lists+ => list-1***mapcar** *function &rest lists+ => result-list***mapcan** *function &rest lists+ => concatenated-results***mapl** *function &rest lists+ => list-1***maplist** *function &rest lists+ => result-list***mapcon** *function &rest lists+ => concatenated-results***Arguments and Values:***function*—a designator for a function that must take as many arguments as there are *lists*.*list*—a proper list.*list-1*—the first *list* (which must be a proper list).*result-list*—a list.*concatenated-results*—a list.**Description:**

The mapping operation involves applying *function* to successive sets of arguments in which one argument is obtained from each sequence. Except for **mapc** and **mapl**, the result contains the results returned by *function*. In the cases of **mapc** and **mapl**, the resulting sequence is *list*.

*function* is called first on all the elements with index 0, then on all those with index 1, and so on. *result-type* specifies the type of the resulting sequence. If *function* is a symbol, it is coerced to a function as if by symbol-function.

**mapcar** operates on successive elements of the *lists*. *function* is applied to the first element of each *list*, then to the second element of each *list*, and so on. The iteration terminates when the shortest *list* runs out, and excess elements in other lists are ignored. The value returned by **mapcar** is a list of the results of successive calls to *function*.

**mapc** is like **mapcar** except that the results of applying *function* are not accumulated. The *list* argument is returned.

**maplist** is like **mapcar** except that *function* is applied to successive sublists of the *lists*. *function* is first applied to the *lists* themselves, and then to the cdr of each *list*, and then to the cdr of the cdr of each *list*, and so on.

**mapl** is like **maplist** except that the results of applying *function* are not accumulated; *list-1* is returned.

## CLHS: Declaration DYNAMIC-EXTENT

mapcan and mapcon are like mapcar and maplist respectively, except that the results of applying *function* are combined into a *list* by the use of nconc rather than list. That is,

```
(mapcon f x1 ... xn)
== (apply #'nconc (maplist f x1 ... xn))
```

and similarly for the relationship between mapcan and mapcar.

### Examples:

```
(mapcar #'car '((1 a) (2 b) (3 c))) => (1 2 3)
(mapcar #'abs '(3 -4 2 -5 -6)) => (3 4 2 5 6)
(mapcar #'cons '(a b c) '(1 2 3)) => ((A . 1) (B . 2) (C . 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))
(maplist #'(lambda (x) (cons 'foo x)) '(a b c d))
=> ((FOO A B C D) (FOO B C D) (FOO C D) (FOO D))
(maplist #'(lambda (x) (if (member (car x) (cdr x)) 0 1)) '(a b a c d b c))
=> (0 0 1 0 1 1 1)
;An entry is 1 if the corresponding element of the input
; list was the last instance of that element in the input list.

(setq dummy nil) => NIL
(mapc #'(lambda (&rest x) (setq dummy (append dummy x)))
      '(1 2 3 4)
      '(a b c d e)
      '(x y z)) => (1 2 3 4)
dummy => (1 A X 2 B Y 3 C Z)

(setq dummy nil) => NIL
(mapl #'(lambda (x) (push x dummy)) '(1 2 3 4)) => (1 2 3 4)
dummy => ((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)
(mapcan #'(lambda (x) (and (numberp x) (list x)))
        '(a 1 b c 3 4 d 5))
=> (1 3 4 5)
```

In this case the function serves as a filter; this is a standard Lisp idiom using mapcan.

```
(mapcon #'list '(1 2 3 4)) => ((1 2 3 4) (2 3 4) (3 4) (4))
```

**Affected By:** None.

### Exceptional Situations:

Should be prepared to signal an error of type type-error if any *list* is not a proper list.

### See Also:

dolist, map, Section 3.6 (Traversal Rules and Side Effects)

**Notes:** None.

## Function MAPHASH

### Syntax:

**maphash** *function hash-table => nil*

### Arguments and Values:

*function*—a designator for a function of two arguments, the key and the value.

*hash-table*—a hash table.

### Description:

Iterates over all entries in the *hash-table*. For each entry, the *function* is called with two arguments—the key and the value of that entry.

The consequences are unspecified if any attempt is made to add or remove an entry from the *hash-table* while a **maphash** is in progress, with two exceptions: the *function* can use setf of gethash to change the value part of the entry currently being processed, or it can use remhash to remove that entry.

### Examples:

```
(setq table (make-hash-table)) => #<HASH-TABLE EQL 0/120 32304110>
(dotimes (i 10) (setf (gethash i table) i)) => NIL
(let ((sum-of-squares 0))
  (maphash #'(lambda (key val)
              (let ((square (* val val)))
                (incf sum-of-squares square)
                (setf (gethash key table) square)))
            table)
  sum-of-squares) => 285
(hash-table-count table) => 10
(maphash #'(lambda (key val)
              (when (oddp val) (remhash key table)))
            table) => NIL
(hash-table-count table) => 5
(maphash #'(lambda (k v) (print (list k v))) table)
(0 0)
(8 64)
(2 4)
(6 36)
(4 16)
=> NIL
```

### Side Effects:

None, other than any which might be done by the *function*.

**Affected By:** None.

**Exceptional Situations:** None.

### See Also:

[loop](#), [with-hash-table-iterator](#), Section 3.6 (Traversal Rules and Side Effects)**Notes:** None.**Accessor MASK-FIELD****Syntax:**

```
mask-field bytespec integer => masked-integer
(setf (mask-field bytespec place) new-masked-integer)
```

**Arguments and Values:***bytespec*—a byte specifier.*integer*—an integer.*masked-integer*, *new-masked-integer*—a non-negative integer.**Description:**

**mask-field** performs a "mask" operation on *integer*. It returns an integer that has the same bits as *integer* in the byte specified by *bytespec*, but that has zero-bits everywhere else.

**setf** may be used with **mask-field** to modify a byte within the integer that is stored in a given *place*. The effect is to perform a deposit-field operation and then store the result back into the *place*.

**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
```

**Side Effects:** None.**Affected By:** None.**Exceptional Situations:** None.**See Also:**[byte](#), [ldb](#)**Notes:**

```
(ldb bs (mask-field bs n)) == (ldb bs n)
(logbitp j (mask-field (byte s p) n))
== (and (>= j p) (< j s) (logbitp j n))
```

```
(mask-field bs n) == (logand n (dpb -1 bs 0))
```

## Function MAX, MIN

### Syntax:

**max** &rest *reals+* => *max-real*

**min** &rest *reals+* => *min-real*

### Arguments and Values:

*real*—a *real*.

*max-real*, *min-real*—a *real*.

### Description:

**max** returns the *real* that is greatest (closest to positive infinity). **min** returns the *real* that is least (closest to negative infinity).

For **max**, the implementation has the choice of returning the largest argument as is or applying the rules of floating-point *contagion*, taking all the arguments into consideration for *contagion* purposes. Also, if one or more of the arguments are  $\equiv$ , then any one of them may be chosen as the value to return. For example, if the *reals* are a mixture of *rationals* and *floats*, and the largest argument is a *rational*, then the implementation is free to produce either that *rational* or its *float* approximation; if the largest argument is a *float* of a smaller format than the largest format of any *float* argument, then the implementation is free to return the argument in its given format or expanded to the larger format. Similar remarks apply to **min** (replacing "largest argument" by "smallest argument").

### Examples:

```
(max 3) => 3
(min 3) => 3
(max 6 12) => 12
(min 6 12) => 6
(max -6 -12) => -6
(min -6 -12) => -12
(max 1 3 2 -7) => 3
(min 1 3 2 -7) => -7
(max -2 3 0 7) => 7
(min -2 3 0 7) => -2
(max 5.0 2) => 5.0
(min 5.0 2)
=> 2
OR=> 2.0
(max 3.0 7 1)
=> 7
OR=> 7.0
(min 3.0 7 1)
=> 1
OR=> 1.0
(max 1.0s0 7.0d0) => 7.0d0
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(min 1.0s0 7.0d0)
=> 1.0s0
OR=> 1.0d0
(max 3 1 1.0s0 1.0d0)
=> 3
OR=> 3.0d0
(min 3 1 1.0s0 1.0d0)
=> 1
OR=> 1.0s0
OR=> 1.0d0
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:**

Should signal an error of type type-error if any *number* is not a *real*.

**See Also:** None.

**Notes:** None.

## **Function MEMBER, MEMBER-IF, MEMBER-IF-NOT**

**Syntax:**

**member** *item list &key key test test-not => tail*

**member-if** *predicate list &key key => tail*

**member-if-not** *predicate list &key key => tail*

**Arguments and Values:**

*item*---an object.

*list*---a proper list.

*predicate*---a designator for a function of one argument that returns a generalized boolean.

*test*---a designator for a function of two arguments that returns a generalized boolean.

*test-not*---a designator for a function of two arguments that returns a generalized boolean.

*key*---a designator for a function of one argument, or **nil**.

*tail*---a list.

**Description:**

## CLHS: Declaration DYNAMIC-EXTENT

**member**, **member-if**, and **member-if-not** each search *list* for *item* or for a top-level element that *satisfies the test*. The argument to the *predicate* function is an element of *list*.

If some element *satisfies the test*, the tail of *list* beginning with this element is returned; otherwise **nil** is returned.

*list* is searched on the top level only.

### Examples:

```
(member 2 '(1 2 3)) => (2 3)
(member 2 '((1 . 2) (3 . 4)) :test-not #'= :key #'cdr) => ((3 . 4))
(member 'e '(a b c d)) => NIL

(member-if #'listp '(a b nil c d)) => (NIL C D)
(member-if #'numberp '(a #\Space 5/3 foo)) => (5/3 FOO)
(member-if-not #'zerop
  '(3 6 9 11 . 12)
  :key #'(lambda (x) (mod x 3))) => (11 . 12)
```

**Side Effects:** None.

**Affected By:** None.

### Exceptional Situations:

Should be prepared to signal an error of *type type-error* if *list* is not a *proper list*.

### See Also:

[find, position, Section 3.6 \(Traversal Rules and Side Effects\)](#)

### Notes:

The *:test-not* parameter is deprecated.

The *function member-if-not* is deprecated.

In the following

```
(member 'a '(g (a y) c a d e a f)) => (A D E A F)
```

the value returned by **member** is *identical* to the portion of the *list* beginning with **a**. Thus **rplaca** on the result of **member** can be used to alter the part of the *list* where **a** was found (assuming a check has been made that **member** did not return **nil**).

## Function MERGE

### Syntax:

**merge** *result-type sequence-1 sequence-2 predicate &key key => result-sequence*

**Arguments and Values:**

*result-type*—a sequence type specifier.

*sequence-1*—a sequence.

*sequence-2*—a sequence.

*predicate*—a designator for a function of two arguments that returns a generalized boolean.

*key*—a designator for a function of one argument, or nil.

*result-sequence*—a proper sequence of type result-type.

**Description:**

Destructively merges *sequence-1* with *sequence-2* according to an order determined by the *predicate*. merge determines the relationship between two elements by giving keys extracted from the sequence elements to the *predicate*.

The first argument to the *predicate* function is an element of *sequence-1* as returned by the *key* (if supplied); the second argument is an element of *sequence-2* as returned by the *key* (if supplied). *Predicate* should return true if and only if its first argument is strictly less than the second (in some appropriate sense). If the first argument is greater than or equal to the second (in the appropriate sense), then *predicate* should return false. merge considers two elements *x* and *y* to be equal if (`funcall predicate x y`) and (`funcall predicate y x`) both yield false.

The argument to the *key* is the *sequence* element. Typically, the return value of the *key* becomes the argument to *predicate*. If *key* is not supplied or nil, the sequence element itself is used. The *key* may be executed more than once for each sequence element, and its side effects may occur in any order.

If *key* and *predicate* return, then the merging operation will terminate. The result of merging two sequences *x* and *y* is a new sequence of type *result-type z*, such that the length of *z* is the sum of the lengths of *x* and *y*, and *z* contains all the elements of *x* and *y*. If *x<sub>1</sub>* and *x<sub>2</sub>* are two elements of *x*, and *x<sub>1</sub>* precedes *x<sub>2</sub>* in *x*, then *x<sub>1</sub>* precedes *x<sub>2</sub>* in *z*, and similarly for elements of *y*. In short, *z* is an interleaving of *x* and *y*.

If *x* and *y* were correctly sorted according to the *predicate*, then *z* will also be correctly sorted. If *x* or *y* is not so sorted, then *z* will not be sorted, but will nevertheless be an interleaving of *x* and *y*.

The merging operation is guaranteed stable; if two or more elements are considered equal by the *predicate*, then the elements from *sequence-1* will precede those from *sequence-2* in the result.

*sequence-1* and/or *sequence-2* may be destroyed.

If the *result-type* is a subtype of list, the result will be a list.

If the *result-type* is a subtype of vector, then if the implementation can determine the element type specified for the *result-type*, the element type of the resulting array is the result of *upgrading* that element type; or, if the implementation can determine that the element type is unspecified (or `*`), the element type of the resulting array is `t`; otherwise, an error is signaled.

**Examples:**

```
(setq test1 (list 1 3 4 6 7))
(setq test2 (list 2 5 8))
(merge 'list test1 test2 #'<) => (1 2 3 4 5 6 7 8)
(setq test1 (copy-seq "BOY"))
(setq test2 (copy-seq :nosy))
(merge 'string test1 test2 #'char-lessp) => "BnOosYy"
(setq test1 (vector ((red . 1) (blue . 4))))
(setq test2 (vector ((yellow . 2) (green . 7))))
(merge 'vector test1 test2 #'< :key #'cdr)
=> #((RED . 1) (YELLOW . 2) (BLUE . 4) (GREEN . 7))

(merge '(vector * 4) '(1 5) '(2 4 6) #'<) should signal an error
```

**Affected By:** None.**Exceptional Situations:**

An error must be signaled if the *result-type* is neither a *recognizable subtype* of list, nor a *recognizable subtype* of vector.

An error of type type-error should be signaled if *result-type* specifies the number of elements and the sum of the lengths of *sequence-1* and *sequence-2* is different from that number.

**See Also:**

sort, stable-sort, Section 3.2.1 (Compiler Terminology), Section 3.6 (Traversal Rules and Side Effects)

**Notes:** None.**Function MERGE-PATHNAMES****Syntax:**

**merge-pathnames** *pathname* &*optional default-pathname default-version*  
 $\Rightarrow$  *merged-pathname*

**Arguments and Values:**

*pathname*—a pathname designator.

*default-pathname*—a pathname designator. The default is the *value* of \*default-pathname-defaults\*.

*default-version*—a valid pathname version. The default is :newest.

*merged-pathname*—a pathname.

**Description:**

Constructs a pathname from *pathname* by filling in any unsupplied components with the corresponding values

from *default-pathname* and *default-version*.

Defaulting of pathname components is done by filling in components taken from another *pathname*. This is especially useful for cases such as a program that has an input file and an output file. Unspecified components of the output pathname will come from the input pathname, except that the type should not default to the type of the input pathname but rather to the appropriate default type for output from the program; for example, see the **function compile-file-pathname**.

If no version is supplied, *default-version* is used. If *default-version* is **nil**, the version component will remain unchanged.

If *pathname* explicitly specifies a host and not a device, and if the host component of *default-pathname* matches the host component of *pathname*, then the device is taken from the *default-pathname*; otherwise the device will be the default file device for that host. If *pathname* does not specify a host, device, directory, name, or type, each such component is copied from *default-pathname*. If *pathname* does not specify a name, then the version, if not provided, will come from *default-pathname*, just like the other components. If *pathname* does specify a name, then the version is not affected by *default-pathname*. If this process leaves the version missing, the *default-version* is used. If the host's file name syntax provides a way to input a version without a name or type, the user can let the name and type default but supply a version different from the one in *default-pathname*.

If *pathname* is a **stream**, *pathname* effectively becomes ( pathname *pathname* ). **merge-pathnames** can be used on either an open or a closed **stream**.

If *pathname* is a **pathname** it represents the name used to open the file. This may be, but is not required to be, the actual name of the file.

**merge-pathnames** recognizes a **logical pathname namestring** when *default-pathname* is a **logical pathname**, or when the **namestring** begins with the name of a defined **logical host** followed by a **colon**. In the first of these two cases, the host portion of the **logical pathname namestring** and its following **colon** are optional.

**merge-pathnames** returns a **logical pathname** if and only if its first argument is a **logical pathname**, or its first argument is a **logical pathname namestring** with an explicit host, or its first argument does not specify a host and the *default-pathname* is a **logical pathname**.

**Pathname** merging treats a relative directory specially. If ( pathname-directory *pathname* ) is a **list** whose **car** is **:relative**, and ( pathname-directory *default-pathname* ) is a **list**, then the merged directory is the value of

```
(append (pathname-directory default-pathname)
       (cdr ;remove :relative from the front
         (pathname-directory pathname)))
```

except that if the resulting **list** contains a **string** or **:wild** immediately followed by **:back**, both of them are removed. This removal of redundant **:back keywords** is repeated as many times as possible. If ( pathname-directory *default-pathname* ) is not a **list** or ( pathname-directory *pathname* ) is not a **list** whose **car** is **:relative**, the merged directory is ( or ( pathname-directory *pathname* ) ( pathname-directory *default-pathname* ) )

**merge-pathnames** maps customary case in *pathname* into customary case in the output *pathname*.

**Examples:**

```
(merge-pathnames "CMUC::FORMAT"
                 "CMUC::PS:<LISPIO>.FASL")
=> #P"CMUC::PS:<LISPIO>FORMAT.FASL.0"
```

**Affected By:** None.**Exceptional Situations:** None.**See Also:**

**\*default-pathname-defaults\***, **pathname**, **logical-pathname**, [Section 20.1 \(File System Concepts\)](#),  
[Section 19.1.2 \(Pathnames as Filenames\)](#)

**Notes:**

The net effect is that if just a name is supplied, the host, device, directory, and type will come from *default-pathname*, but the version will come from *default-version*. If nothing or just a directory is supplied, the name, type, and version will come from *default-pathname* together.

**Function METHOD-COMBINATION-ERROR****Syntax:**

**method-combination-error** *format-control* &*rest args* => [\*implementation-dependent\*](#)

**Arguments and Values:**

*format-control*—a [\*format control\*](#).

*args*—[\*format arguments\*](#) for *format-control*.

**Description:**

The [\*\*function method-combination-error\*\*](#) is used to signal an error in method combination.

The error message is constructed by using a *format-control* suitable for [\*\*format\*\*](#) and any *args* to it. Because an implementation may need to add additional contextual information to the error message, **method-combination-error** should be called only within the dynamic extent of a method combination function.

Whether [\*\*method-combination-error\*\*](#) returns to its caller or exits via [\*\*throw\*\*](#) is [\*implementation-dependent\*](#).

**Examples:** None.**Side Effects:**

The debugger might be entered.

**Affected By:**

**\*break-on-signals\***

**Exceptional Situations:** None.

**See Also:**

**define-method-combination**

**Notes:** None.

***Standard Generic Function METHOD-QUALIFIERS***

**Syntax:**

**method-qualifiers** *method* => *qualifiers*

**Method Signatures:**

**method-qualifiers** (*method* standard-method)

**Arguments and Values:**

*method*—a method.

*qualifiers*—a proper list.

**Description:**

Returns a list of the qualifiers of the *method*.

**Examples:**

```
(defmethod some-gf :before ((a integer)) a)
=> #<STANDARD-METHOD SOME-GF (:BEFORE) (INTEGER) 42736540>
(method-qualifiers *) => (:BEFORE)
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**define-method-combination**

**Notes:** None.

***Function MACROEXPAND, MACROEXPAND-1***

**Syntax:**

**macroexpand** *form &optional env => expansion, expanded-p*

**macroexpand-1** *form &optional env => expansion, expanded-p*

### Arguments and Values:

*form*—a form.

*env*—an environment object. The default is nil.

*expansion*—a form.

*expanded-p*—a generalized boolean.

### Description:

**macroexpand** and **macroexpand-1** expand macros.

If *form* is a macro form, then **macroexpand-1** expands the macro form call once.

**macroexpand** repeatedly expands *form* until it is no longer a macro form. In effect, **macroexpand** calls **macroexpand-1** repeatedly until the secondary value it returns is nil.

If *form* is a macro form, then the *expansion* is a macro expansion and *expanded-p* is true. Otherwise, the *expansion* is the given *form* and *expanded-p* is false.

Macro expansion is carried out as follows. Once **macroexpand-1** has determined that the *form* is a macro form, it obtains an appropriate expansion function for the macro or symbol macro. The value of **\*macroexpand-hook\*** is coerced to a function and then called as a function of three arguments: the expansion function, the *form*, and the *env*. The value returned from this call is taken to be the expansion of the *form*.

In addition to macro definitions in the global environment, any local macro definitions established within *env* by **macrolet** or **symbol-macrolet** are considered. If only *form* is supplied as an argument, then the environment is effectively null, and only global macro definitions as established by **defmacro** are considered. Macro definitions are shadowed by local function definitions.

### Examples:

```
(defmacro alpha (x y) `(beta ,x ,y)) => ALPHA
(defmacro beta (x y) `(gamma ,x ,y)) => BETA
(defmacro delta (x y) `(gamma ,x ,y)) => EPSILON
(defmacro expand (form &environment env)
  (multiple-value-bind (expansion expanded-p)
    (macroexpand form env)
    `(~(values ',expansion ',expanded-p))) => EXPAND
(defmacro expand-1 (form &environment env)
  (multiple-value-bind (expansion expanded-p)
    (macroexpand-1 form env)
    `(~(values ',expansion ',expanded-p))) => EXPAND-1
;; Simple examples involving just the global environment
(macroexpand-1 '(alpha a b)) => (BETA A B), true
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(expand-1 (alpha a b)) => (BETA A B), true
(macroexpand '(alpha a b)) => (GAMMA A B), true
(expand (alpha a b)) => (GAMMA A B), true
(macroexpand-1 'not-a-macro) => NOT-A-MACRO, false
(expand-1 not-a-macro) => NOT-A-MACRO, false
(macroexpand '(not-a-macro a b)) => (NOT-A-MACRO A B), false
(expand (not-a-macro a b)) => (NOT-A-MACRO A B), false

;; Examples involving lexical environments
(macrolet ((alpha (x y) `(delta ,x ,y)))
  (macroexpand-1 '(alpha a b)) => (BETA A B), true
(macrolet ((alpha (x y) `(delta ,x ,y)))
  (expand-1 (alpha a b)) => (DELTA A B), true
(macrolet ((alpha (x y) `(delta ,x ,y)))
  (macroexpand '(alpha a b)) => (GAMMA A B), true
(macrolet ((alpha (x y) `(delta ,x ,y)))
  (expand (alpha a b)) => (GAMMA A B), true
(macrolet ((beta (x y) `'(epsilon ,x ,y)))
  (expand (alpha a b)) => (EPSILON A B), true
(let ((x (list 1 2 3)))
  (symbol-macrolet ((a (first x)))
    (expand a)) => (FIRST X), true
(let ((x (list 1 2 3)))
  (symbol-macrolet ((a (first x)))
    (macroexpand 'a)) => A, false
(symbol-macrolet ((b (alpha x y)))
  (expand-1 b)) => (ALPHA X Y), true
(symbol-macrolet ((b (alpha x y)))
  (expand b)) => (GAMMA X Y), true
(symbol-macrolet ((b (alpha x y)))
  (a b))
  (expand-1 a)) => B, true
(symbol-macrolet ((b (alpha x y)))
  (a b))
  (expand a)) => (GAMMA X Y), true

;; Examples of shadowing behavior
(flet ((beta (x y) (+ x y)))
  (expand (alpha a b)) => (BETA A B), true
(macrolet ((alpha (x y) `(delta ,x ,y)))
  (flet ((alpha (x y) (+ x y)))
    (expand (alpha a b))) => (ALPHA A B), false
(let ((x (list 1 2 3)))
  (symbol-macrolet ((a (first x)))
    (let ((a x))
      (expand a)))) => A, false
```

### Affected By:

defmacro, setf of macro-function, macrolet, symbol-macrolet

**Exceptional Situations:** None.

### See Also:

\*macroexpand-hook\*, defmacro, setf of macro-function, macrolet, symbol-macrolet, Section 3.1 (Evaluation)

**Notes:**

Neither **macroexpand** nor **macroexpand-1** makes any explicit attempt to expand *macro forms* that are either *subforms* of the *form* or *subforms* of the *expansion*. Such expansion might occur implicitly, however, due to the semantics or implementation of the *macro function*.

## **Function MINUSP, PLUSP**

**Syntax:**

**minusp** *real* => *generalized-boolean*

**plusp** *real* => *generalized-boolean*

**Arguments and Values:**

*real*—*a real*.

*generalized-boolean*—*a generalized boolean*.

**Description:**

**minusp** returns *true* if *real* is less than zero; otherwise, returns *false*.

**plusp** returns *true* if *real* is greater than zero; otherwise, returns *false*.

Regardless of whether an *implementation* provides distinct representations for positive and negative *float* zeros, (**minusp -0.0**) always returns *false*.

**Examples:**

```
(minusp -1) => true
(plusp 0) => false
(plusp least-positive-single-float) => true
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:**

Should signal an error of *type type-error* if *real* is not a *real*.

**See Also:** None.

**Notes:** None.

## Function MISMATCH

### Syntax:

**mismatch** *sequence-1 sequence-2 &key from-end test test-not key start1 start2 end1 end2 => position*

### Arguments and Values:

*Sequence-1*—a sequence.

*Sequence-2*—a sequence.

*from-end*—a generalized boolean. The default is false.

*test*—a designator for a function of two arguments that returns a generalized boolean.

*test-not*—a designator for a function of two arguments that returns a generalized boolean.

*start1, end1*—bounding index designators of *sequence-1*. The defaults for *start1* and *end1* are 0 and nil, respectively.

*start2, end2*—bounding index designators of *sequence-2*. The defaults for *start2* and *end2* are 0 and nil, respectively.

*key*—a designator for a function of one argument, or nil.

*position*—a bounding index of *sequence-1*, or nil.

### Description:

The specified subsequences of *sequence-1* and *sequence-2* are compared element-wise.

The *key* argument is used for both the *sequence-1* and the *sequence-2*.

If *sequence-1* and *sequence-2* are of equal length and match in every element, the result is false. Otherwise, the result is a non-negative integer, the index within *sequence-1* of the leftmost or rightmost position, depending on *from-end*, at which the two subsequences fail to match. If one subsequence is shorter than and a matching prefix of the other, the result is the index relative to *sequence-1* beyond the last position tested.

If *from-end* is true, then one plus the index of the rightmost position in which the *sequences* differ is returned. In effect, the subsequences are aligned at their right-hand ends; then, the last elements are compared, the penultimate elements, and so on. The index returned is an index relative to *sequence-1*.

### 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
(mismatch '(1 2 3 4 5 6) '(3 4 5 6 7) :start1 2 :end2 4) => NIL
```

**Side Effects:** None.**Affected By:** None.**Exceptional Situations:** None.**See Also:**[Section 3.6 \(Traversal Rules and Side Effects\)](#)**Notes:**The `:test-not argument` is deprecated.

## **Function MAKE-ARRAY**

**Syntax:**

**make-array** *dimensions &key element-type initial-element initial-contents adjustable fill-pointer displaced-to displaced-index-offset*

=> *new-array*

**Arguments and Values:**

*dimensions*—*a designator* for a *list* of *valid array dimensions*.

*element-type*—*a type specifier*. The default is `t`.

*initial-element*—*an object*.

*initial-contents*—*an object*.

*adjustable*—*a generalized boolean*. The default is `nil`.

*fill-pointer*—*a valid fill pointer* for the *array* to be created, or `t` or `nil`. The default is `nil`.

*displaced-to*—*an array or nil*. The default is `nil`. This option must not be supplied if either *initial-element* or *initial-contents* is supplied.

*displaced-index-offset*—*a valid array row-major index* for *displaced-to*. The default is 0. This option must not be supplied unless a *non-nil displaced-to* is supplied.

*new-array*—*an array*.

**Description:**

Creates and returns an *array* constructed of the most *specialized type* that can accommodate elements of *type* given by *element-type*. If *dimensions* is `nil` then a zero-dimensional *array* is created.

*Dimensions* represents the dimensionality of the new *array*.

## CLHS: Declaration DYNAMIC-EXTENT

*element-type* indicates the *type* of the elements intended to be stored in the *new-array*. The *new-array* can actually store any *objects* of the *type* which results from *upgrading element-type*; see [Section 15.1.2.1 \(Array Upgrading\)](#).

If *initial-element* is supplied, it is used to initialize each *element* of *new-array*. If *initial-element* is supplied, it must be of the *type* given by *element-type*. *initial-element* cannot be supplied if either the *:initial-contents* option is supplied or *displaced-to* is *non-nil*. If *initial-element* is not supplied, the consequences of later reading an uninitialized *element* of *new-array* are undefined unless either *initial-contents* is supplied or *displaced-to* is *non-nil*.

*initial-contents* is used to initialize the contents of *array*. For example:

```
(make-array '(4 2 3) :initial-contents
            '(((a b c) (1 2 3))
              ((d e f) (3 1 2))
              ((g h i) (2 3 1))
              ((j k l) (0 0 0))))
```

*initial-contents* is composed of a nested structure of *sequences*. The numbers of levels in the structure must equal the rank of *array*. Each leaf of the nested structure must be of the *type* given by *element-type*. If *array* is zero-dimensional, then *initial-contents* specifies the single *element*. Otherwise, *initial-contents* must be a *sequence* whose length is equal to the first dimension; each element must be a nested structure for an *array* whose dimensions are the remaining dimensions, and so on. *Initial-contents* cannot be supplied if either *initial-element* is supplied or *displaced-to* is *non-nil*. If *initial-contents* is not supplied, the consequences of later reading an uninitialized *element* of *new-array* are undefined unless either *initial-element* is supplied or *displaced-to* is *non-nil*.

If *adjustable* is *non-nil*, the array is *expressly adjustable* (and so *actually adjustable*); otherwise, the array is not *expressly adjustable* (and it is *implementation-dependent* whether the array is *actually adjustable*).

If *fill-pointer* is *non-nil*, the *array* must be one-dimensional; that is, the *array* must be a *vector*. If *fill-pointer* is *t*, the length of the *vector* is used to initialize the *fill pointer*. If *fill-pointer* is an *integer*, it becomes the initial *fill pointer* for the *vector*.

If *displaced-to* is *non-nil*, **make-array** will create a *displaced array* and *displaced-to* is the *target* of that *displaced array*. In that case, the consequences are undefined if the *actual array element type* of *displaced-to* is not *type equivalent* to the *actual array element type* of the *array* being created. If *displaced-to* is *nil*, the *array* is not a *displaced array*.

The *displaced-index-offset* is made to be the index offset of the *array*. When an array A is given as the *:displaced-to argument* to **make-array** when creating array B, then array B is said to be displaced to array A. The total number of elements in an *array*, called the total size of the *array*, is calculated as the product of all the dimensions. It is required that the total size of A be no smaller than the sum of the total size of B plus the offset n supplied by the *displaced-index-offset*. The effect of displacing is that array B does not have any elements of its own, but instead maps *accesses* to itself into *accesses* to array A. The mapping treats both *arrays* as if they were one-dimensional by taking the elements in row-major order, and then maps an *access* to element k of array B to an *access* to element k+n of array A.

If **make-array** is called with *adjustable*, *fill-pointer*, and *displaced-to* each *nil*, then the result is a *simple array*. If **make-array** is called with one or more of *adjustable*, *fill-pointer*, or *displaced-to* being *true*, whether the resulting *array* is a *simple array* is *implementation-dependent*.

## CLHS: Declaration DYNAMIC-EXTENT

When an array A is given as the :displaced-to argument to **make-array** when creating array B, then array B is said to be displaced to array A. The total number of elements in an array, called the total size of the array, is calculated as the product of all the dimensions. The consequences are unspecified if the total size of A is smaller than the sum of the total size of B plus the offset n supplied by the *displaced-index-offset*. The effect of displacing is that array B does not have any elements of its own, but instead maps accesses to itself into accesses to array A. The mapping treats both arrays as if they were one-dimensional by taking the elements in row-major order, and then maps an access to element k of array B to an access to element k+n of array A.

### Examples:

```
(make-array 5) ;; Creates a one-dimensional array of five elements.
(make-array '(3 4) :element-type '(mod 16)) ;; Creates a
; ;two-dimensional array, 3 by 4, with four-bit elements.
(make-array 5 :element-type 'single-float) ;; Creates an array of single-floats.

(make-array nil :initial-element nil) => #0ANIL
(make-array 4 :initial-element nil) => #(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 'character
            :initial-element #\a
            :fill-pointer 3) => "aaa"
```

The following is an example of making a displaced array.

```
(setq a (make-array '(4 3)))
=> #<ARRAY 4x3 simple 32546632>
(dotimes (i 4)
  (dotimes (j 3)
    (setf (aref a i j) (list i 'x j '= (* i j)))))
=> NIL
(setq b (make-array 8 :displaced-to a
                    :displaced-index-offset 2))
=> #<ARRAY 8 indirect 32550757>
(dotimes (i 8)
  (print (list i (aref b i))))
>> (0 (0 X 2 = 0))
>> (1 (1 X 0 = 0))
>> (2 (1 X 1 = 1))
>> (3 (1 X 2 = 2))
>> (4 (2 X 0 = 0))
>> (5 (2 X 1 = 2))
>> (6 (2 X 2 = 4))
>> (7 (3 X 0 = 0))
=> NIL
```

The last example depends on the fact that arrays are, in effect, stored in row-major order.

```
(setq a1 (make-array 50))
=> #<ARRAY 50 simple 32562043>
(setq b1 (make-array 20 :displaced-to a1 :displaced-index-offset 10))
=> #<ARRAY 20 indirect 32563346>
(length b1) => 20
```

```
(setq a2 (make-array 50 :fill-pointer 10))
=> #<ARRAY 50 fill-pointer 10 46100216>
(setq b2 (make-array 20 :displaced-to a2 :displaced-index-offset 10))
=> #<ARRAY 20 indirect 46104010>
  (length a2) => 10
  (length b2) => 20

  (setq a3 (make-array 50 :fill-pointer 10))
  => #<ARRAY 50 fill-pointer 10 46105663>
  (setq b3 (make-array 20 :displaced-to a3 :displaced-index-offset 10
                       :fill-pointer 5))
=> #<ARRAY 20 indirect, fill-pointer 5 46107432>
  (length a3) => 10
  (length b3) => 5
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[adjustable-array-p](#), [oref](#), [arrayp](#), [array-element-type](#), [array-rank-limit](#), [array-dimension-limit](#), [fill-pointer](#), [upgraded-array-element-type](#)

**Notes:**

There is no specified way to create an array for which adjustable-array-p definitely returns false. There is no specified way to create an array that is not a simple array.

## Function MAKE-BROADCAST-STREAM

**Syntax:**

**make-broadcast-stream** &*rest streams* => *broadcast-stream*

**Arguments and Values:**

*stream*—an output stream.

*broadcast-stream*—a broadcast stream.

**Description:**

Returns a broadcast stream.

**Examples:**

```
(setq a-stream (make-string-output-stream)
      b-stream (make-string-output-stream)) => #<String Output Stream>
(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"
```

**Side Effects:** None.**Affected By:** None.**Exceptional Situations:**Should signal an error of *type type-error* if any *stream* is not an *output stream*.**See Also:****broadcast-stream-streams****Notes:** None.**Function MAKE-CONDITION****Syntax:****make-condition** *type* &rest *slot-initializations* => *condition***Arguments and Values:***type*—a *type specifier* (for a *subtype* of *condition*).*slot-initializations*—an *initialization argument list*.*condition*—a *condition*.**Description:**Constructs and returns a *condition* of type *type* using *slot-initializations* for the initial values of the slots. The newly created *condition* is returned.**Examples:**

```
(defvar *oops-count* 0)

(setq a (make-condition 'simple-error
                        :format-control "This is your ~:R error."
                        :format-arguments (list (incf *oops-count*)))
=) #<SIMPLE-ERROR 32245104>

(format t "~&~A~%" a)
>> This is your first error.
=> NIL

(error a)
>> Error: This is your first error.
>> To continue, type :CONTINUE followed by an option number:
>> 1: Return to Lisp Toplevel.
>> Debug>
```

**Side Effects:** None.

**Affected By:**

The set of defined condition types.

**Exceptional Situations:** None.

**See Also:**

define-condition, Section 9.1 (Condition System Concepts)

**Notes:** None.

**Function MAKE-CONCATENATED-STREAM****Syntax:**

**make-concatenated-stream** &rest *input-streams* => *concatenated-stream*

**Arguments and Values:**

*input-stream*—an input stream.

*concatenated-stream*—a concatenated stream.

**Description:**

Returns a concatenated stream that has the indicated *input-streams* initially associated with it.

**Examples:**

```
(read (make-concatenated-stream
      (make-string-input-stream "1")
      (make-string-input-stream "2")))) => 12
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:**

Should signal type-error if any argument is not an input stream.

**See Also:**

concatenated-stream-streams

**Notes:** None.

**Function MAKE-DISPATCH-MACRO-CHARACTER****Syntax:**

**make-dispatch-macro-character** *char &optional non-terminating-p readable => t*

**Arguments and Values:**

*char*—a character.

*non-terminating-p*—a generalized boolean. The default is false.

*readtable*—a readtable. The default is the current readtable.

**Description:**

**make-dispatch-macro-character** makes *char* be a dispatching macro character in *readtable*.

Initially, every character in the dispatch table associated with the *char* has an associated function that signals an error of type reader-error.

If *non-terminating-p* is true, the dispatching macro character is made a non-terminating macro character; if *non-terminating-p* is false, the dispatching macro character is made a terminating macro character.

**Examples:**

```
(get-macro-character #\{} => NIL, false
(make-dispatch-macro-character #\{} => T
(not (get-macro-character #\{})) => false
```

**Side Effects:** None.

The *readtable* is altered.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

\*readtable\*, set-dispatch-macro-character

**Notes:** None.

**Function MAKE-ECHO-STREAM****Syntax:**

**make-echo-stream** *input-stream output-stream => echo-stream*

**Arguments and Values:**

*input-stream*—an input stream.

*output-stream*—an output stream.

*echo-stream*—an echo stream.

**Description:**

Creates and returns an echo stream that takes input from *input-stream* and sends output to *output-stream*.

**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"
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

echo-stream-input-stream, echo-stream-output-stream, make-two-way-stream

**Notes:** None.

## Function MAKE-HASH-TABLE

**Syntax:**

**make-hash-table** &*key test size rehash-size rehash-threshold => hash-table*

**Arguments and Values:**

*test*—a designator for one of the functions **eq**, **eql**, **equal**, or **equalp**. The default is **eql**.

*size*—a non-negative integer. The default is implementation-dependent.

*rehash-size*—a real of type (or (integer 1 \*) (float (1.0) \*)). The default is implementation-dependent.

*rehash-threshold*—a real of type (real 0 1). The default is implementation-dependent.

*hash-table*—a hash table.

**Description:**

Creates and returns a new *hash table*.

*test* determines how *keys* are compared. An *object* is said to be present in the *hash-table* if that *object* is the *same* under the *test* as the *key* for some entry in the *hash-table*.

*size* is a hint to the *implementation* about how much initial space to allocate in the *hash-table*. This information, taken together with the *rehash-threshold*, controls the approximate number of entries which it should be possible to insert before the table has to grow. The actual size might be rounded up from *size* to the next `good' size; for example, some *implementations* might round to the next prime number.

*rehash-size* specifies a minimum amount to increase the size of the *hash-table* when it becomes full enough to require rehashing; see *rehash-threshold* below. If *rehash-size* is an *integer*, the expected growth rate for the table is additive and the *integer* is the number of entries to add; if it is a *float*, the expected growth rate for the table is multiplicative and the *float* is the ratio of the new size to the old size. As with *size*, the actual size of the increase might be rounded up.

*rehash-threshold* specifies how full the *hash-table* can get before it must grow. It specifies the maximum desired hash-table occupancy level.

The *values* of *rehash-size* and *rehash-threshold* do not constrain the *implementation* to use any particular method for computing when and by how much the size of *hash-table* should be enlarged. Such decisions are *implementation-dependent*, and these *values* only hints from the *programmer* to the *implementation*, and the *implementation* is permitted to ignore them.

**Examples:**

```
(setq table (make-hash-table)) => #<HASH-TABLE EQL 0/120 46142754>
(setf (gethash "one" table) 1) => 1
(gethash "one" table) => NIL, false
(setq table (make-hash-table :test 'equal)) => #<HASH-TABLE EQUAL 0/139 46145547>
(setf (gethash "one" table) 1) => 1
(gethash "one" table) => 1, T
(make-hash-table :rehash-size 1.5 :rehash-threshold 0.7)
=> #<HASH-TABLE EQL 0/120 46156620>
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[gethash, hash-table](#)

**Notes:** None.

**Standard Generic Function MAKE-INSTANCES-OBSOLETE****Syntax:**

**make-instances-obsolete** *class* => *class*

**Method Signatures:**

**make-instances-obsolete** (*class standard-class*)

**make-instances-obsolete** (*class symbol*)

**Arguments and Values:**

*class*—a *class designator*.

**Description:**

The *function make-instances-obsolete* has the effect of initiating the process of updating the instances of the *class*. During updating, the generic function **update-instance-for-redefined-class** will be invoked.

The generic function **make-instances-obsolete** is invoked automatically by the system when **defclass** has been used to redefine an existing standard class and the set of local *slots accessible* in an instance is changed or the order of *slots* in storage is changed. It can also be explicitly invoked by the user.

If the second of the above *methods* is selected, that *method* invokes **make-instances-obsolete** on (*find-class class*).

**Examples:**

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**update-instance-for-redefined-class**, [Section 4.3.6 \(Redefining Classes\)](#)

**Notes:** None.

**Standard Generic Function MAKE-INSTANCE****Syntax:**

**make-instance** *class &rest initargs &key &allow-other-keys => instance*

**Method Signatures:**

**make-instance** (*class standard-class*) *&rest initargs*

**make-instance** (*class symbol*) *&rest initargs*

**Arguments and Values:**

*class*—a *class*, or a *symbol* that names a *class*.

*initargs*—an *initialization argument list*.

*instance*---a *fresh instance* of *class class*.

#### Description:

The *generic function make-instance* creates and returns a new *instance* of the given *class*.

If the second of the above *methods* is selected, that *method* invokes *make-instance* on the arguments (*find-class class*) and *initargs*.

The initialization arguments are checked within *make-instance*.

The *generic function make-instance* may be used as described in [Section 7.1 \(Object Creation and Initialization\)](#).

**Affected By:** None.

#### Exceptional Situations:

If any of the initialization arguments has not been declared as valid, an error of *type error* is signaled.

#### See Also:

[defclass](#), [class-of](#), [allocate-instance](#), [initialize-instance](#), [Section 7.1 \(Object Creation and Initialization\)](#)

**Notes:** None.

## **Function MAKE-LOAD-FORM-SAVING-SLOTS**

#### Syntax:

**make-load-form-saving-slots** *object &key slot-names environment*

=> *creation-form, initialization-form*

#### Arguments and Values:

*object*---an *object*.

*slot-names*---a *list*.

*environment*---an *environment object*.

*creation-form*---a *form*.

*initialization-form*---a *form*.

#### Description:

Returns *forms* that, when *evaluated*, will construct an *object* equivalent to *object*, without *executing initialization forms*. The *slots* in the new *object* that correspond to initialized *slots* in *object* are initialized using the values from *object*. Uninitialized *slots* in *object* are not initialized in the new *object*.

**make-load-form-saving-slots** works for any *instance* of **standard-object** or **structure-object**.

*Slot-names* is a *list* of the names of the *slots* to preserve. If *slot-names* is not supplied, its value is all of the *local slots*.

**make-load-form-saving-slots** returns two values, thus it can deal with circular structures. Whether the result is useful in an application depends on whether the *object's type* and slot contents fully capture the application's idea of the *object's state*.

*Environment* is the environment in which the forms will be processed.

**Examples:** None.

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**make-load-form, make-instance, setf, slot-value, slot-makunbound**

**Notes:**

**make-load-form-saving-slots** can be useful in user-written **make-load-form** methods.

When the *object* is an *instance* of **standard-object**, **make-load-form-saving-slots** could return a creation form that *calls allocate-instance* and an initialization form that contains *calls* to **setf** of **slot-value** and **slot-makunbound**, though other *functions* of similar effect might actually be used.

## **Standard Generic Function MAKE-LOAD-FORM**

**Syntax:**

**make-load-form *object* &optional *environment* => *creation-form*[, *initialization-form*]**

**Method Signatures:**

**make-load-form (*object standard-object*) &optional *environment***

**make-load-form (*object structure-object*) &optional *environment***

**make-load-form (*object condition*) &optional *environment***

**make-load-form (*object class*) &optional *environment***

**Arguments and Values:**

*object*—an *object*.

*environment*—an environment object.

*creation-form*—a form.

*initialization-form*—a form.

### Description:

The generic function make-load-form creates and returns one or two forms, a *creation-form* and an *initialization-form*, that enable load to construct an object equivalent to object. Environment is an environment object corresponding to the lexical environment in which the forms will be processed.

The file compiler calls make-load-form to process certain classes of literal objects; see Section 3.2.4.4 (Additional Constraints on Externalizable Objects).

Conforming programs may call make-load-form directly, providing object is a generalized instance of standard-object, structure-object, or condition.

The creation form is a form that, when evaluated at load time, should return an object that is equivalent to object. The exact meaning of equivalent depends on the type of object and is up to the programmer who defines a method for make-load-form; see Section 3.2.4 (Literal Objects in Compiled Files).

The initialization form is a form that, when evaluated at load time, should perform further initialization of the object. The value returned by the initialization form is ignored. If make-load-form returns only one value, the initialization form is nil, which has no effect. If object appears as a constant in the initialization form, at load time it will be replaced by the equivalent object constructed by the creation form; this is how the further initialization gains access to the object.

Both the *creation-form* and the *initialization-form* may contain references to any externalizable object. However, there must not be any circular dependencies in creation forms. An example of a circular dependency is when the creation form for the object X contains a reference to the object Y, and the creation form for the object Y contains a reference to the object X. Initialization forms are not subject to any restriction against circular dependencies, which is the reason that initialization forms exist; see the example of circular data structures below.

The creation form for an object is always *evaluated* before the initialization form for that object. When either the creation form or the initialization form references other objects that have not been referenced earlier in the file being compiled, the compiler ensures that all of the referenced objects have been created before *evaluating* the referencing form. When the referenced object is of a type which the file compiler processes using make-load-form, this involves *evaluating* the creation form returned for it. (This is the reason for the prohibition against circular references among creation forms).

Each initialization form is *evaluated* as soon as possible after its associated creation form, as determined by data flow. If the initialization form for an object does not reference any other objects not referenced earlier in the file and processed by the file compiler using make-load-form, the initialization form is evaluated immediately after the creation form. If a creation or initialization form F does contain references to such objects, the creation forms for those other objects are evaluated before F, and the initialization forms for those other objects are also evaluated before F whenever they do not depend on the object created or initialized by F. Where these rules do not uniquely determine an order of evaluation between two creation/initialization forms, the order of evaluation is unspecified.

## CLHS: Declaration DYNAMIC-EXTENT

While these creation and initialization forms are being evaluated, the *objects* are possibly in an uninitialized state, analogous to the state of an *object* between the time it has been created by **allocate-instance** and it has been processed fully by **initialize-instance**. Programmers writing *methods* for **make-load-form** must take care in manipulating *objects* not to depend on *slots* that have not yet been initialized.

It is *implementation-dependent* whether **load** calls **eval** on the *forms* or does some other operation that has an equivalent effect. For example, the *forms* might be translated into different but equivalent *forms* and then evaluated, they might be compiled and the resulting functions called by **load**, or they might be interpreted by a special-purpose function different from **eval**. All that is required is that the effect be equivalent to evaluating the *forms*.

The *method specialized* on **class** returns a creation *form* using the *name* of the *class* if the *class* has a *proper name* in *environment*, signaling an error of *type error* if it does not have a *proper name*. *Evaluation* of the creation *form* uses the *name* to find the *class* with that *name*, as if by calling **find-class**. If a *class* with that *name* has not been defined, then a *class* may be computed in an *implementation-defined* manner. If a *class* cannot be returned as the result of *evaluating* the creation *form*, then an error of *type error* is signaled.

Both *conforming implementations* and *conforming programs* may further *specialize make-load-form*.

### Examples:

```
(defclass obj ()
  ((x :initarg :x :reader obj-x)
   (y :initarg :y :reader obj-y)
   (dist :accessor obj-dist)))
=> #<STANDARD-CLASS OBJ 250020030>
(defmethod shared-initialize :after ((self obj) slot-names &rest keys)
  (declare (ignore slot-names keys))
  (unless (slot-boundp self 'dist)
    (setf (obj-dist self)
          (sqrt (+ (expt (obj-x self) 2) (expt (obj-y self) 2))))))
=> #<STANDARD-METHOD SHARED-INITIALIZE (:AFTER) (OBJ T) 26266714>
(defmethod make-load-form ((self obj) &optional environment)
  (declare (ignore environment))
  ; Note that this definition only works because X and Y do not
  ; contain information which refers back to the object itself.
  ; For a more general solution to this problem, see revised example below.
  `(make-instance ',(class-of self)
                 :x ,(obj-x self) :y ,(obj-y self)))
=> #<STANDARD-METHOD MAKE-LOAD-FORM (OBJ) 26267532>
(setq obj1 (make-instance 'obj :x 3.0 :y 4.0)) => #<OBJ 26274136>
(obj-dist obj1) => 5.0
(make-load-form obj1) => (MAKE-INSTANCE 'OBJ :X '3.0 :Y '4.0)
```

In the above example, an equivalent *instance* of *obj* is reconstructed by using the values of two of its *slots*. The value of the third *slot* is derived from those two values.

Another way to write the **make-load-form method** in that example is to use **make-load-form-saving-slots**. The code it generates might yield a slightly different result from the **make-load-form method** shown above, but the operational effect will be the same. For example:

```
; Redefine method defined above.
(defmethod make-load-form ((self obj) &optional environment)
  (make-load-form-saving-slots self
                               :slot-names '(x y))
```

## CLHS: Declaration DYNAMIC-EXTENT

```
:environment environment))  
=> #<STANDARD-METHOD MAKE-LOAD-FORM (OBJ) 42755655>  
;; Try MAKE-LOAD-FORM on object created above.  
(make-load-form obj1)  
=> (ALLOCATE-INSTANCE '#<STANDARD-CLASS OBJ 250020030>,  
(PROGN  
  (SETF (SLOT-VALUE '#<OBJ 26274136> 'X) '3.0)  
  (SETF (SLOT-VALUE '#<OBJ 26274136> 'Y) '4.0)  
  (INITIALIZE-INSTANCE '#<OBJ 26274136>))
```

In the following example, *instances* of my-frob are "interned" in some way. An equivalent *instance* is reconstructed by using the value of the name slot as a key for searching existing *objects*. In this case the programmer has chosen to create a new *object* if no existing *object* is found; alternatively an error could have been signaled in that case.

```
(defclass my-frob ()  
  ((name :initarg :name :reader my-name)))  
(defmethod make-load-form ((self my-frob) &optional environment)  
  (declare (ignore environment))  
  `(find-my-frob ',(my-name self) :if-does-not-exist :create))
```

In the following example, the data structure to be dumped is circular, because each parent has a list of its children and each child has a reference back to its parent. If **make-load-form** is called on one *object* in such a structure, the creation form creates an equivalent *object* and fills in the children slot, which forces creation of equivalent *objects* for all of its children, grandchildren, etc. At this point none of the parent *slots* have been filled in. The initialization form fills in the parent *slot*, which forces creation of an equivalent *object* for the parent if it was not already created. Thus the entire tree is recreated at **load** time. At compile time, **make-load-form** is called once for each *object* in the tree. All of the creation forms are evaluated, in *implementation-dependent* order, and then all of the initialization forms are evaluated, also in *implementation-dependent* order.

```
(defclass tree-with-parent () ((parent :accessor tree-parent)  
                                (children :initarg :children)))  
(defmethod make-load-form ((x tree-with-parent) &optional environment)  
  (declare (ignore environment))  
  (values  
    ;; creation form  
    `(make-instance ',(class-of x) :children ',(slot-value x 'children))  
    ;; initialization form  
    `'(setf (tree-parent ',x) ',(slot-value x 'parent))))
```

In the following example, the data structure to be dumped has no special properties and an equivalent structure can be reconstructed simply by reconstructing the *slots*' contents.

```
(defstruct my-struct a b c)  
(defmethod make-load-form ((s my-struct) &optional environment)  
  (make-load-form-saving-slots s :environment environment))
```

**Affected By:** None.

### Exceptional Situations:

The *methods specialized* on **standard-object**, **structure-object**, and **condition** all signal an error of *type error*.

It is *implementation-dependent* whether calling **make-load-form** on a *generalized instance* of a *system class* signals an error or returns creation and initialization *forms*.

#### See Also:

[compile-file](#), [make-load-form-saving-slots](#), [Section 3.2.4.4 \(Additional Constraints on Externalizable Objects\)](#) [Section 3.1 \(Evaluation\)](#), [Section 3.2 \(Compilation\)](#)

#### Notes:

The *file compiler* calls **make-load-form** in specific circumstances detailed in [Section 3.2.4.4 \(Additional Constraints on Externalizable Objects\)](#).

Some *implementations* may provide facilities for defining new *subclasses* of *classes* which are specified as *system classes*. (Some likely candidates include **generic-function**, **method**, and **stream**). Such *implementations* should document how the *file compiler* processes *instances* of such *classes* when encountered as *literal objects*, and should document any relevant *methods* for **make-load-form**.

## Function MAKE-LIST

#### Syntax:

**make-list** *size* &*key* *initial-element* => *list*

#### Arguments and Values:

*size*—a non-negative *integer*.

*initial-element*—an *object*. The default is **nil**.

*list*—a *list*.

#### Description:

Returns a *list* of *length* given by *size*, each of the *elements* of which is *initial-element*.

#### Examples:

```
(make-list 5) => (NIL NIL NIL NIL NIL)
(make-list 3 :initial-element 'rah) => (RAH RAH RAH)
(make-list 2 :initial-element '(1 2 3)) => ((1 2 3) (1 2 3))
(make-list 0) => NIL ;i.e., ()
(make-list 0 :initial-element 'new-element) => NIL
```

**Side Effects:** None.

**Affected By:** None.

#### Exceptional Situations:

Should signal an error of **type-error** if *size* is not a non-negative *integer*.

**See Also:**[cons](#), [list](#)**Notes:** None.**Function MAKE-PACKAGE****Syntax:****make-package** *package-name* &*key nicknames use => package***Arguments and Values:***package-name*—a string designator.*nicknames*—a list of string designators. The default is the empty list.*use*—a list of package designators. The default is implementation-defined.*package*—a package.**Description:**Creates a new package with the name *package-name*.*Nicknames* are additional names which may be used to refer to the new package.*use* specifies zero or more packages the external symbols of which are to be inherited by the new package. See the function use-package.**Examples:**

```
(make-package 'temporary :nicknames '("TEMP" "temp")) => #<PACKAGE "TEMPORARY">
(make-package "OWNER" :use '("temp")) => #<PACKAGE "OWNER">
(package-used-by-list 'temp) => (#<PACKAGE "OWNER">)
(package-use-list 'owner) => (#<PACKAGE "TEMPORARY">)
```

**Side Effects:** None.**Affected By:**The existence of other packages in the system.**Exceptional Situations:**The consequences are unspecified if packages denoted by *use* do not exist.A correctable error is signaled if the *package-name* or any of the *nicknames* is already the name or nickname of an existing package.

**See Also:****defpackage, use-package****Notes:**

In situations where the *packages* to be used contain symbols which would conflict, it is necessary to first create the package with `:use '()`, then to use **shadow** or **shadowing-import** to address the conflicts, and then after that to use **use-package** once the conflicts have been addressed.

When packages are being created as part of the static definition of a program rather than dynamically by the program, it is generally considered more stylistically appropriate to use **defpackage** rather than **make-package**.

## **Function MAKE-PATHNAME**

**Syntax:**

**make-pathname** &key host device directory name type version defaults case

=> pathname

**Arguments and Values:**

*host*—a *valid physical pathname host*. Complicated defaulting behavior; see below.

*device*—a *valid pathname device*. Complicated defaulting behavior; see below.

*directory*—a *valid pathname directory*. Complicated defaulting behavior; see below.

*name*—a *valid pathname name*. Complicated defaulting behavior; see below.

*type*—a *valid pathname type*. Complicated defaulting behavior; see below.

*version*—a *valid pathname version*. Complicated defaulting behavior; see below.

*defaults*—a *pathname designator*. The default is a *pathname* whose host component is the same as the host component of the *value* of **\*default-pathname-defaults\***, and whose other components are all **nil**.

*case*—one of `:common` or `:local`. The default is `:local`.

*pathname*—a *pathname*.

**Description:**

Constructs and returns a *pathname* from the supplied keyword arguments.

After the components supplied explicitly by *host*, *device*, *directory*, *name*, *type*, and *version* are filled in, the merging rules used by **merge-pathnames** are used to fill in any unsupplied components from the defaults supplied by *defaults*.

## CLHS: Declaration DYNAMIC-EXTENT

Whenever a *pathname* is constructed the components may be canonicalized if appropriate. For the explanation of the arguments that can be supplied for each component, see [Section 19.2.1 \(Pathname Components\)](#).

If *case* is supplied, it is treated as described in [Section 19.2.2.1.2 \(Case in Pathname Components\)](#).

The resulting *pathname* is a *logical pathname* if and only its host component is a *logical host* or a *string* that names a defined *logical host*.

If the *directory* is a *string*, it should be the name of a top level directory, and should not contain any punctuation characters; that is, specifying a *string*, *str*, is equivalent to specifying the list (:absolute *str*). Specifying the symbol :wild is equivalent to specifying the list (:absolute :wild-inferiors), or (:absolute :wild) in a file system that does not support :wild-inferiors.

### Examples:

```
; Implementation A -- an implementation with access to a single
; Unix file system. This implementation happens to never display
; the 'host' information in a namestring, since there is only one host.
(make-pathname :directory '(:absolute "public" "games")
               :name "chess" :type "db")
=> #P"/public/games/chess.db"

; Implementation B -- an implementation with access to one or more
; VMS file systems. This implementation displays 'host' information
; in the namestring only when the host is not the local host.
; It uses a double colon to separate a host name from the host's local
; file name.
(make-pathname :directory '(:absolute "PUBLIC" "GAMES")
               :name "CHESS" :type "DB")
=> #P"SYS$DISK:[PUBLIC.GAMES]CHESS.DB"
(make-pathname :host "BOBBY"
               :directory '(:absolute "PUBLIC" "GAMES")
               :name "CHESS" :type "DB")
=> #P"BOBBY::SYS$DISK:[PUBLIC.GAMES]CHESS.DB"

; Implementation C -- an implementation with simultaneous access to
; multiple file systems from the same Lisp image. In this
; implementation, there is a convention that any text preceding the
; first colon in a pathname namestring is a host name.
(dolist (case '(:common :local))
  (dolist (host '("MY-LISPM" "MY-VAX" "MY-UNIX"))
    (print (make-pathname :host host :case case
                           :directory '(:absolute "PUBLIC" "GAMES")
                           :name "CHESS" :type "DB"))))

>> #P"MY-LISPM:>public>games>chess.db"
>> #P"MY-VAX:SYS$DISK:[PUBLIC.GAMES]CHESS.DB"
>> #P"MY-UNIX:/public/games/chess.db"
>> #P"MY-LISPM:>public>games>chess.db"
>> #P"MY-VAX:SYS$DISK:[PUBLIC.GAMES]CHESS.DB"
>> #P"MY-UNIX:/PUBLIC/GAMES/CHESS.DB"
=> NIL
```

### Affected By:

The *file system*.

**Exceptional Situations:** None.

**See Also:**

[merge-pathnames](#), [pathname](#), [logical-pathname](#), [Section 20.1 \(File System Concepts\)](#), [Section 19.1.2 \(Pathnames as Filenames\)](#)

**Notes:**

Portable programs should not supply :unspecified for any component. See [Section 19.2.2.2.3 \(:UNSPECIFIC as a Component Value\)](#).

## **Function MAKE-RANDOM-STATE**

**Syntax:**

**make-random-state** &optional state => new-state

**Arguments and Values:**

state---a *random state*, or **nil**, or **t**. The default is **nil**.

new-state---a *random state object*.

**Description:**

Creates a *fresh object* of *type random-state* suitable for use as the *value* of **\*random-state\***.

If state is a *random state object*, the new-state is a *copy*[5] of that *object*. If state is **nil**, the new-state is a *copy*[5] of the *current random state*. If state is **t**, the new-state is a *fresh random state object* that has been randomly initialized by some means.

**Examples:**

```
(let* ((rs1 (make-random-state nil))
       (rs2 (make-random-state t))
       (rs3 (make-random-state rs2)))
  (rs4 nil))
  (list (loop for i from 1 to 10
             collect (random 100)
             when (= i 5)
             do (setq rs4 (make-random-state)))
        (loop for i from 1 to 10 collect (random 100 rs1))
        (loop for i from 1 to 10 collect (random 100 rs2))
        (loop for i from 1 to 10 collect (random 100 rs3))
        (loop for i from 1 to 10 collect (random 100 rs4))))
=> ((29 25 72 57 55 68 24 35 54 65)
     (29 25 72 57 55 68 24 35 54 65)
     (93 85 53 99 58 62 2 23 23 59)
     (93 85 53 99 58 62 2 23 23 59)
     (68 24 35 54 65 54 55 50 59 49))
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:**

Should signal an error of type type-error if *state* is not a random state, or nil, or t.

**See Also:**

random, \*random-state\*

**Notes:**

One important use of make-random-state is to allow the same series of pseudo-random numbers to be generated many times within a single program.

**Function MAKE-STRING-INPUT-STREAM****Syntax:**

**make-string-input-stream** *string* &*optional start end* => *string-stream*

**Arguments and Values:**

*string*—a string.

*start*, *end*—bounding index designators of *string*. The defaults for *start* and *end* are 0 and nil, respectively.

*string-stream*—an input string stream.

**Description:**

Returns an input string stream. This stream will supply, in order, the characters in the substring of *string* bounded by *start* and *end*. After the last character has been supplied, the string stream will then be at end of file.

**Examples:**

```
(let ((string-stream (make-string-input-stream "1 one ")))
  (list (read string-stream nil nil)
        (read string-stream nil nil)
        (read string-stream nil nil)))
=> (1 ONE NIL)

(read (make-string-input-stream "prefixtargetsuffix" 6 12)) => TARGET
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

with-input-from-string

**Notes:** None.

## Function MAKE-STRING-OUTPUT-STREAM

**Syntax:**

**make-string-output-stream** &*key element-type => string-stream*

**Arguments and Values:**

*element-type*—a type specifier. The default is character.

*string-stream*—an output string stream.

**Description:**

Returns an output string stream that accepts characters and makes available (via get-output-stream-string) a string that contains the characters that were actually output.

The *element-type* names the type of the elements of the string; a string is constructed of the most specialized type that can accommodate elements of that *element-type*.

**Examples:**

```
(let ((s (make-string-output-stream)))
  (write-string "testing... " s)
  (prin1 1234 s)
  (get-output-stream-string s))
=> "testing... 1234"
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

get-output-stream-string, with-output-to-string

**Notes:** None.

## Function MAKE-SEQUENCE

**Syntax:**

**make-sequence** *result-type size* &*key initial-element => sequence*

**Arguments and Values:**

*result-type*—a sequence type specifier.

*size*—a non-negative integer.

*initial-element*—an object. The default is implementation-dependent.

*sequence*—a proper sequence.

### Description:

Returns a sequence of the type *result-type* and of length *size*, each of the elements of which has been initialized to *initial-element*.

If the *result-type* is a subtype of list, the result will be a list.

If the *result-type* is a subtype of vector, then if the implementation can determine the element type specified for the *result-type*, the element type of the resulting array is the result of *upgrading* that element type; or, if the implementation can determine that the element type is unspecified (or \*), the element type of the resulting array is t; otherwise, an error is signaled.

### Examples:

```
(make-sequence 'list 0) => ()
(make-sequence 'string 26 :initial-element #\.)
=> "....."
(make-sequence '(vector double-float) 2
               :initial-element 1d0)
=> #(1.0d0 1.0d0)

(make-sequence '(vector * 2) 3) should signal an error
(make-sequence '(vector * 4) 3) should signal an error
```

### Affected By:

The implementation.

### Exceptional Situations:

The consequences are unspecified if *initial-element* is not an object which can be stored in the resulting sequence.

An error of type type-error must be signaled if the *result-type* is neither a recognizable subtype of list, nor a recognizable subtype of vector.

An error of type type-error should be signaled if *result-type* specifies the number of elements and *size* is different from that number.

### See Also:

make-array, make-list

### Notes:

```
(make-sequence 'string 5) == (make-string 5)
```

***Function MAKE-STRING*****Syntax:**

**make-string** *size &key initial-element element-type => string*

**Arguments and Values:**

*size*—a valid array dimension.

*initial-element*—a character. The default is implementation-dependent.

*element-type*—a type specifier. The default is character.

*string*—a simple string.

**Description:**

**make-string** returns a simple string of length *size* whose elements have been initialized to *initial-element*.

The *element-type* names the type of the elements of the string; a string is constructed of the most specialized type that can accommodate elements of the given type.

**Examples:**

```
(make-string 10 :initial-element #\5) => "5555555555"
(length (make-string 10)) => 10
```

**Affected By:**

The implementation.

**Exceptional Situations:** None.

**See Also:** None.

**Notes:** None.

***Function MAKE-SYMBOL*****Syntax:**

**make-symbol** *name => new-symbol*

**Arguments and Values:**

*name*—a string.

*new-symbol*—a fresh, uninterned symbol.

**Description:**

## CLHS: Declaration DYNAMIC-EXTENT

**make-symbol** creates and returns a *fresh, uninterned symbol* whose *name* is the given *name*. The *new-symbol* is neither *bound* nor *fbound* and has a *null property list*.

It is *implementation-dependent* whether the *string* that becomes the *new-symbol's name* is the given *name* or a copy of it. Once a *string* has been given as the *name argument* to **make-symbol**, the consequences are undefined if a subsequent attempt is made to alter that *string*.

### Examples:

```
(setq temp-string "temp") => "temp"
(setq temp-symbol (make-symbol temp-string)) => #:|temp|
(symbol-name temp-symbol) => "temp"
(eq (symbol-name temp-symbol) temp-string) => implementation-dependent
(find-symbol "temp") => NIL, NIL
(eq (make-symbol temp-string) (make-symbol temp-string)) => false
```

**Side Effects:** None.

**Affected By:** None.

### Exceptional Situations:

Should signal an error of *type type-error* if *name* is not a *string*.

### See Also:

**copy-symbol**

### Notes:

No attempt is made by **make-symbol** to convert the case of the *name* to uppercase. The only case conversion which ever occurs for *symbols* is done by the *Lisp reader*. The program interface to *symbol* creation retains case, and the program interface to interning symbols is case-sensitive.

## Function MAKE-SYNONYM-STREAM

### Syntax:

**make-synonym-stream** *symbol* => *synonym-stream*

### Arguments and Values:

*symbol*—a *symbol* that names a *dynamic variable*.

*synonym-stream*—a *synonym stream*.

### Description:

Returns a *synonym stream* whose *synonym stream symbol* is *symbol*.

### Examples:

## CLHS: Declaration DYNAMIC-EXTENT

```
(setq a-stream (make-string-input-stream "a-stream")
      b-stream (make-string-input-stream "b-stream"))
=> #<String Input Stream>
(setq s-stream (make-synonym-stream 'c-stream))
=> #<SYNONYM-STREAM for C-STREAM>
(setq c-stream a-stream)
=> #<String Input Stream>
(read s-stream) => A-STREAM
(setq c-stream b-stream)
=> #<String Input Stream>
(read s-stream) => B-STREAM
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:**

Should signal **type-error** if its argument is not a symbol.

**See Also:**

**Notes:** None.

## Function MAKE-TWO-WAY-STREAM

**Syntax:**

**make-two-way-stream** *input-stream output-stream => two-way-stream*

**Arguments and Values:**

*input-stream*—a *stream*.

*output-stream*—a *stream*.

*two-way-stream*—a *two-way stream*.

**Description:**

Returns a *two-way stream* that gets its input from *input-stream* and sends its output to *output-stream*.

**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...)
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:**

Should signal an error of type type-error if *input-stream* is not an input stream. Should signal an error of type type-error if *output-stream* is not an output stream.

**See Also:** None.

**Notes:** None.

## **Function MOD, REM**

**Syntax:**

**mod** *number divisor => modulus*

**rem** *number divisor => remainder*

**Arguments and Values:**

*number*—a real.

*divisor*—a real.

*modulus, remainder*—a real.

**Description:**

**mod** and **rem** are generalizations of the modulus and remainder functions respectively.

**mod** performs the operation floor on *number* and *divisor* and returns the remainder of the floor operation.

**rem** performs the operation truncate on *number* and *divisor* and returns the remainder of the truncate operation.

**mod** and **rem** are the modulus and remainder functions when *number* and *divisor* are integers.

**Examples:**

```
(rem -1 5) => -1
(mod -1 5) => 4
(mod 13 4) => 1
(rem 13 4) => 1
(mod -13 4) => 3
(rem -13 4) => -1
(mod 13 -4) => -3
(rem 13 -4) => 1
(mod -13 -4) => -1
(rem -13 -4) => -1
(mod 13.4 1) => 0.4
(rem 13.4 1) => 0.4
```

```
(mod -13.4 1) => 0.6
(rem -13.4 1) => -0.4
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[floor](#), [truncate](#)

**Notes:**

## **Function NAME-CHAR**

**Syntax:**

**name-char** *name* => *char-p*

**Arguments and Values:**

*name*—[a string designator](#).

*char-p*—[a character](#) or [nil](#).

**Description:**

Returns the [character object](#) whose [name](#) is *name* (as determined by [string-equal](#)—i.e., lookup is not case sensitive). If such a [character](#) does not exist, [nil](#) is returned.

**Examples:**

```
(name-char 'space) => #\Space
(name-char "space") => #\Space
(name-char "Space") => #\Space
(let ((x (char-name #\a)))
  (or (not x) (eql (name-char x) #\a))) => true
```

**Affected By:** None.

**Exceptional Situations:**

Should signal an error of [type type-error](#) if *name* is not a [string designator](#).

**See Also:**

[char-name](#)

**Notes:** None.

## **Function NAMESTRING, FILE-NAMESTRING, DIRECTORY-NAMESTRING, HOST-NAMESTRING, ENOUGH-NAMESTRING**

### Syntax:

**namestring** *pathname* => *namestring*

**file-namestring** *pathname* => *namestring*

**directory-namestring** *pathname* => *namestring*

**host-namestring** *pathname* => *namestring*

**enough-namestring** *pathname* &optional *defaults* => *namestring*

### Arguments and Values:

*pathname*—a *pathname designator*.

*defaults*—a *pathname designator*. The default is the value of **\*default-pathname-defaults\***.

*namestring*—a *string* or *nil*.

### Description:

These functions convert *pathname* into a namestring. The name represented by *pathname* is returned as a namestring in an implementation-dependent canonical form.

**namestring** returns the full form of *pathname*.

**file-namestring** returns just the name, type, and version components of *pathname*.

**directory-namestring** returns the directory name portion.

**host-namestring** returns the host name.

**enough-namestring** returns an abbreviated namestring that is just sufficient to identify the file named by *pathname* when considered relative to the *defaults*. It is required that

```
(merge-pathnames (enough-namestring pathname defaults) defaults)
== (merge-pathnames (parse-namestring pathname nil defaults) defaults)
```

in all cases, and the result of **enough-namestring** is the shortest reasonable string that will satisfy this criterion.

It is not necessarily possible to construct a valid namestring by concatenating some of the three shorter namestrings in some order.

### Examples:

```
(namestring "getty")
```

## CLHS: Declaration DYNAMIC-EXTENT

```
=> "getty"
 (setq q (make-pathname :host "kathy"
                        :directory
                        (pathname-directory *default-pathname-defaults*)
                        :name "getty"))
=> #S(PATHNAME :HOST "kathy" :DEVICE NIL :DIRECTORY directory-name
      :NAME "getty" :TYPE NIL :VERSION NIL)
 (file-namestring q) => "getty"
 (directory-namestring q) => directory-name
 (host-namestring q) => "kathy"

 ;;;Using Unix syntax and the wildcard conventions used by the
 ;;;particular version of Unix on which this example was created:
 (namestring
  (translate-pathname "/usr/dmr/hacks/frob.l"
                      "/usr/d*/hacks/*.l"
                      "/usr/d*/backup/hacks/backup-*.*"))
=> "/usr/dmr/backup/hacks/backup-frob.l"
 (namestring
  (translate-pathname "/usr/dmr/hacks/frob.l"
                      "/usr/d*/hacks/fr*.l"
                      "/usr/d*/backup/hacks/backup-*.*"))
=> "/usr/dmr/backup/hacks/backup-ob.l"

 ;;;This is similar to the above example but uses two different hosts,
 ;;;U: which is a Unix and V: which is a VMS. Note the translation
 ;;;of file type and alphabetic case conventions.
 (namestring
  (translate-pathname "U:/usr/dmr/hacks/frob.l"
                      "U:/usr/d*/hacks/*.l"
                      "V:SYS$DISK:[D*.BACKUP.HACKS]BACKUP-*.*"))
=> "V:SYS$DISK:[DMR.BACKUP.HACKS]BACKUP-FROB.LSP"
 (namestring
  (translate-pathname "U:/usr/dmr/hacks/frob.l"
                      "U:/usr/d*/hacks/fr*.l"
                      "V:SYS$DISK:[D*.BACKUP.HACKS]BACKUP-*.*"))
=> "V:SYS$DISK:[DMR.BACKUP.HACKS]BACKUP-OB.LSP"
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[truename](#), [merge-pathnames](#), [pathname](#), [logical-pathname](#), [Section 20.1 \(File System Concepts\)](#), [Section 19.1.2 \(Pathnames as Filenames\)](#)

**Notes:** None.

## Function NCONC

**Syntax:**

**nconc** &*rest lists* => *concatenated-list*

**Arguments and Values:**

## CLHS: Declaration DYNAMIC-EXTENT

*list*—each but the last must be a list (which might be a *dotted list* but must not be a *circular list*); the last *list* may be any object.

*concatenated-list*—a list.

### Description:

Returns a list that is the concatenation of *lists*. If no *lists* are supplied, (`nconc`) returns nil. `nconc` is defined using the following recursive relationship:

```
(nconc) => ()
(nconc nil . lists) == (nconc . lists)
(nconc list) => list
(nconc list-1 list-2) == (progn (rplacd (last list-1) list-2) list-1)
(nconc list-1 list-2 . lists) == (nconc (nconc list-1 list-2) . lists)
```

### Examples:

```
(nconc) => NIL
(setq x '(a b c)) => (A B C)
(setq y '(d e f)) => (D E F)
(nconc x y) => (A B C D E F)
x => (A B C D E F)
```

Note, in the example, that the value of `x` is now different, since its last cons has been rplacd'd to the value of `y`. If (`nconc x y`) were evaluated again, it would yield a piece of a *circular list*, whose printed representation would be (A B C D E F D E F D E F ...), repeating forever; if the \*print-circle\* switch were non-nil, it would be printed as (A B C . #1=(D E F . #1#)).

```
(setq foo (list 'a 'b 'c 'd 'e)
  bar (list 'f 'g 'h 'i 'j)
  baz (list 'k 'l 'm)) => (K L M)
(setq foo (nconc foo bar baz)) => (A B C D E F G H I J K L M)
foo => (A B C D E F G H I J K L M)
bar => (F G H I J K L M)
baz => (K L M)

(setq foo (list 'a 'b 'c 'd 'e)
  bar (list 'f 'g 'h 'i 'j)
  baz (list 'k 'l 'm)) => (K L M)
(setq foo (nconc nil foo bar nil baz)) => (A B C D E F G H I J K L M)
foo => (A B C D E F G H I J K L M)
bar => (F G H I J K L M)
baz => (K L M)
```

### Side Effects:

The *lists* are modified rather than copied.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**append, concatenate**

**Notes:** None.

***Local Function NEXT-METHOD-P***

**Syntax:**

**next-method-p** <no arguments> => *generalized-boolean*

**Arguments and Values:**

*generalized-boolean*—a generalized boolean.

**Description:**

The locally defined function **next-method-p** can be used within the body forms (but not the lambda list) defined by a method-defining form to determine whether a next method exists.

The function next-method-p has lexical scope and indefinite extent.

Whether or not **next-method-p** is fbound in the global environment is implementation-dependent; however, the restrictions on redefinition and shadowing of **next-method-p** are the same as for symbols in the COMMON-LISP package which are fbound in the global environment. The consequences of attempting to use **next-method-p** outside of a method-defining form are undefined.

**Examples:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**call-next-method, defmethod, call-method**

**Notes:** None.

***Standard Generic Function NO-APPLICABLE-METHOD***

**Syntax:**

**no-applicable-method** *generic-function* &*rest function-arguments* => *result\**

**Method Signatures:**

**no-applicable-method** (*generic-function*) &*rest function-arguments*

**Arguments and Values:**

## CLHS: Declaration DYNAMIC-EXTENT

*generic-function*---a generic function on which no applicable method was found.

*function-arguments*---arguments to the *generic-function*.

*result*---an object.

### Description:

The generic function no-applicable-method is called when a generic function is invoked and no method on that generic function is applicable. The default method signals an error.

The generic function no-applicable-method is not intended to be called by programmers. Programmers may write methods for it.

**Examples:** None.

**Affected By:** None.

### Exceptional Situations:

The default method signals an error of type error.

### See Also:

**Notes:** None.

## Standard Generic Function NO-NEXT-METHOD

### Syntax:

**no-next-method** *generic-function* *method* &*rest args* => *result*\*

### Method Signatures:

**no-next-method** (*generic-function* standard-generic-function) (*method* standard-method) &*rest args*

### Arguments and Values:

*generic-function* --- generic function to which *method* belongs.

*method* --- method that contained the call to call-next-method for which there is no next method.

*args* --- arguments to call-next-method.

*result*---an object.

### Description:

The generic function no-next-method is called by call-next-method when there is no next method.

## CLHS: Declaration DYNAMIC-EXTENT

The generic function no-next-method is not intended to be called by programmers. Programmers may write methods for it.

**Examples:** None.

**Affected By:** None.

**Exceptional Situations:**

The system-supplied method on no-next-method signals an error of type error.

**See Also:**

**call-next-method**

**Notes:** None.

## Function NOT

**Syntax:**

**not** *x* => *boolean*

**Arguments and Values:**

*x*—a generalized boolean (i.e., any object).

*boolean*—a boolean.

**Description:**

Returns t if *x* is false; otherwise, returns nil.

**Examples:**

```
(not nil) => T
(not '()) => T
(not (integerp 'sss)) => T
(not (integerp 1)) => NIL
(not 3.7) => NIL
(not 'apple) => NIL
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**null**

**Notes:**

**not** is intended to be used to invert the 'truth value' of a *boolean* (or *generalized boolean*) whereas **null** is intended to be used to test for the *empty list*. Operationally, **not** and **null** compute the same result; which to use is a matter of style.

**Accessor NTH****Syntax:**

**nth** *n list* => *object*

```
(setf (nth n list) new-object)
```

**Arguments and Values:**

*n*—a non-negative *integer*.

*list*—a *list*, which might be a *dotted list* or a *circular list*.

*object*—an *object*.

*new-object*—an *object*.

**Description:**

**nth** locates the *n*th element of *list*, where the *car* of the *list* is the "zeroth" element. Specifically,

```
(nth n list) == (car (nthcdr n list))
```

**nth** may be used to specify a *place* to **setf**. Specifically,

```
(setf (nth n list) new-object) == (setf (car (nthcdr n list)) new-object)
```

**Examples:**

```
(nth 0 '(foo bar baz)) => FOO
(nth 1 '(foo bar baz)) => BAR
(nth 3 '(foo bar baz)) => NIL
(setq 0-to-3 (list 0 1 2 3)) => (0 1 2 3)
(setf (nth 2 0-to-3) "two") => "two"
0-to-3 => (0 1 "two" 3)
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**elt, first, nthcdr**

**Notes:** None.

## Function NTHCDR

**Syntax:**

**nthcdr** *n list => tail*

**Arguments and Values:**

*n*—a non-negative integer.

*list*—a list, which might be a dotted list or a circular list.

*tail*—an object.

**Description:**

Returns the tail of *list* that would be obtained by calling cdr *n* times in succession.

**Examples:**

```
(nthcdr 0 '()) => NIL
(nthcdr 3 '()) => NIL
(nthcdr 0 '(a b c)) => (A B C)
(nthcdr 2 '(a b c)) => (C)
(nthcdr 4 '(a b c)) => ()
(nthcdr 1 '(0 . 1)) => 1

(locally (declare (optimize (safety 3)))
  (nthcdr 3 '(0 . 1)))
Error: Attempted to take CDR of 1.
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:**

Should signal an error of type type-error if *n* is not a non-negative integer.

For *n* being an integer greater than 1, the error checking done by (nthcdr *n list*) is the same as for (nthcdr (- *n* 1) (cdr *list*)); see the function cdr.

**See Also:**

cdr, nth, rest

**Notes:** None.

***Function NULL*****Syntax:**

**null** *object* => *boolean*

**Arguments and Values:**

*object*---an object.

*boolean*---a boolean.

**Description:**

Returns **t** if *object* is the empty list; otherwise, returns **nil**.

**Examples:**

```
(null '()) => T
(null nil) => T
(null t) => NIL
(null 1) => NIL
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**not**

**Notes:**

**null** is intended to be used to test for the empty list whereas **not** is intended to be used to invert a boolean (or generalized boolean). Operationally, **null** and **not** compute the same result; which to use is a matter of style.

```
(null object) == (typep object 'null) == (eq object '())
```

***Function NUMERATOR, DENOMINATOR*****Syntax:**

**numerator** *rational* => *numerator*

**denominator** *rational* => *denominator*

**Arguments and Values:**

*rational*—a *rational*.

*numerator*—an *integer*.

*denominator*—a positive *integer*.

#### Description:

**numerator** and **denominator** reduce *rational* to canonical form and compute the numerator or denominator of that number.

**numerator** and **denominator** return the numerator or denominator of the canonical form of *rational*.

If *rational* is an *integer*, **numerator** returns *rational* and **denominator** returns 1.

#### Examples:

```
(numerator 1/2) => 1
(denominator 12/36) => 3
(numerator -1) => -1
(denominator (/ -33)) => 33
(numerator (/ 8 -6)) => -4
(denominator (/ 8 -6)) => 3
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

#### See Also:

*l*

#### Notes:

```
(gcd (numerator x) (denominator x)) => 1
```

## Function NUMBERP

#### Syntax:

**numberp** *object* => *generalized-boolean*

#### Arguments and Values:

*object*—an *object*.

*generalized-boolean*—a *generalized boolean*.

#### Description:

## CLHS: Declaration DYNAMIC-EXTENT

Returns true if *object* is of type number; otherwise, returns false.

### Examples:

```
(numberp 12) => true
(numberp (expt 2 130)) => true
(numberp #c(5/3 7.2)) => true
(numberp nil) => false
(numberp (cons 1 2)) => false
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:** None.

### Notes:

```
(numberp object) == (typep object 'number)
```

## Function OPEN

### Syntax:

**open** *filespec* &*key direction element-type if-exists if-does-not-exist external-format*

=> *stream*

### Arguments and Values:

*filespec*—a pathname designator.

*direction*—one of :input, :output, :io, or :probe. The default is :input.

*element-type*—a type specifier for recognizable subtype of character; or a type specifier for a finite recognizable subtype of integer; or one of the symbols signed-byte, unsigned-byte, or :default. The default is character.

*if-exists*—one of :error, :new-version, :rename, :rename-and-delete, :overwrite, :append, :supersede, or nil. The default is :new-version if the version component of *filespec* is :newest, or :error otherwise.

*if-does-not-exist*—one of :error, :create, or nil. The default is :error if *direction* is :input or *if-exists* is :overwrite or :append; :create if *direction* is :output or :io, and *if-exists* is neither :overwrite nor :append; or nil when *direction* is :probe.

*external-format*—an external file format designator. The default is :default.

*stream*—a file stream or nil.

**Description:**

**open** creates, opens, and returns a *file stream* that is connected to the file specified by *filespec*. *Filespec* is the name of the file to be opened. If the *filespec designator* is a *stream*, that *stream* is not closed first or otherwise affected.

The keyword arguments to **open** specify the characteristics of the *file stream* that is returned, and how to handle errors.

If *direction* is :input or :probe, or if *if-exists* is not :new-version and the version component of the *filespec* is :newest, then the file opened is that file already existing in the file system that has a version greater than that of any other file in the file system whose other pathname components are the same as those of *filespec*.

An implementation is required to recognize all of the **open** keyword options and to do something reasonable in the context of the host operating system. For example, if a file system does not support distinct file versions and does not distinguish the notions of deletion and expunging, :new-version might be treated the same as :rename or :supersede, and :rename-and-delete might be treated the same as :supersede.

**:direction**

These are the possible values for *direction*, and how they affect the nature of the *stream* that is created:

**:input**

Causes the creation of an *input file stream*.

**:output**

Causes the creation of an *output file stream*.

**:io**

Causes the creation of a *bidirectional file stream*.

**:probe**

Causes the creation of a "no-directional" *file stream*; in effect, the *file stream* is created and then closed prior to being returned by **open**.

**:element-type**

The *element-type* specifies the unit of transaction for the *file stream*. If it is :default, the unit is determined by *file system*, possibly based on the *file*.

**:if-exists**

*if-exists* specifies the action to be taken if *direction* is :output or :io and a file of the name *filespec* already exists. If *direction* is :input, not supplied, or :probe, *if-exists* is ignored. These are the results of **open** as modified by *if-exists*:

**:error**

An error of *type file-error* is signaled.

**:new-version**

A new file is created with a larger version number.

**:rename**

The existing file is renamed to some other name and then a new file is created.

**:rename-and-delete**

The existing file is renamed to some other name, then it is deleted but not expunged, and then a new file is created.

**:overwrite**

## CLHS: Declaration DYNAMIC-EXTENT

Output operations on the *stream* destructively modify the existing file. If *direction* is :io the file is opened in a bidirectional mode that allows both reading and writing. The file pointer is initially positioned at the beginning of the file; however, the file is not truncated back to length zero when it is opened.

### :append

Output operations on the *stream* destructively modify the existing file. The file pointer is initially positioned at the end of the file.

If *direction* is :io, the file is opened in a bidirectional mode that allows both reading and writing.

### :supersede

The existing file is superseded; that is, a new file with the same name as the old one is created. If possible, the implementation should not destroy the old file until the new *stream* is closed.

### nil

No file or *stream* is created; instead, **nil** is returned to indicate failure.

### :if-does-not-exist

*if-does-not-exist* specifies the action to be taken if a file of name *filespec* does not already exist. These are the results of **open** as modified by *if-does-not-exist*:

### :error

An error of *type file-error* is signaled.

### :create

An empty file is created. Processing continues as if the file had already existed but no processing as directed by *if-exists* is performed.

### nil

No file or *stream* is created; instead, **nil** is returned to indicate failure.

### :external-format

This option selects an *external file format* for the *file*: The only *standardized* value for this option is :default, although *implementations* are permitted to define additional *external file formats* and *implementation-dependent* values returned by **stream-external-format** can also be used by *conforming programs*.

The *external-format* is meaningful for any kind of *file stream* whose *element type* is a *subtype* of *character*. This option is ignored for *streams* for which it is not meaningful; however, *implementations* may define other *element types* for which it is meaningful. The consequences are unspecified if a *character* is written that cannot be represented by the given *external file format*.

When a file is opened, a *file stream* is constructed to serve as the file system's ambassador to the Lisp environment; operations on the *file stream* are reflected by operations on the file in the file system.

A file can be deleted, renamed, or destructively modified by **open**.

For information about opening relative pathnames, see [Section 19.2.3 \(Merging Pathnames\)](#).

### Examples:

```
(open filespec :direction :probe) => #<Closed Probe File Stream...>
(setq q (merge-pathnames (user-homedir-pathname) "test"))
=> #<PATHNAME :HOST NIL :DEVICE device-name :DIRECTORY directory-name
  :NAME "test" :TYPE NIL :VERSION :NEWEST>
(open filespec :if-does-not-exist :create) => #<Input File Stream...>
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(setq s (open filespec :direction :probe)) => #<Closed Probe File Stream...>
(truename s) => #<PATHNAME :HOST NIL :DEVICE device-name :DIRECTORY
                  directory-name :NAME filespec :TYPE extension :VERSION 1>
(open s :direction :output :if-exists nil) => NIL
```

### Affected By:

The nature and state of the host computer's *file system*.

### Exceptional Situations:

If *if-exists* is :error, (subject to the constraints on the meaning of *if-exists* listed above), an error of *type file-error* is signaled.

If *if-does-not-exist* is :error (subject to the constraints on the meaning of *if-does-not-exist* listed above), an error of *type file-error* is signaled.

If it is impossible for an implementation to handle some option in a manner close to what is specified here, an error of *type error* might be signaled.

An error of *type file-error* is signaled if (wild-pathname-p *filespec*) returns true.

An error of *type error* is signaled if the *external-format* is not understood by the *implementation*.

The various *file systems* in existence today have widely differing capabilities, and some aspects of the *file system* are beyond the scope of this specification to define. A given *implementation* might not be able to support all of these options in exactly the manner stated. An *implementation* is required to recognize all of these option keywords and to try to do something "reasonable" in the context of the host *file system*. Where necessary to accomodate the *file system*, an *implementation* deviate slightly from the semantics specified here without being disqualified for consideration as a *conforming implementation*. If it is utterly impossible for an *implementation* to handle some option in a manner similar to what is specified here, it may simply signal an error.

With regard to the :element-type option, if a *type* is requested that is not supported by the *file system*, a substitution of types such as that which goes on in *upgrading* is permissible. As a minimum requirement, it should be the case that opening an *output stream* to a *file* in a given *element type* and later opening an *input stream* to the same *file* in the same *element type* should work compatibly.

### See Also:

[with-open-file](#), [close](#), [pathname](#), [logical-pathname](#), [Section 19.2.3 \(Merging Pathnames\)](#), [Section 19.1.2 \(Pathnames as Filenames\)](#)

### Notes:

**open** does not automatically close the file when an abnormal exit occurs.

When *element-type* is a *subtype* of [character](#), [read-char](#) and/or [write-char](#) can be used on the resulting *file stream*.

When *element-type* is a *subtype* of [integer](#), [read-byte](#) and/or [write-byte](#) can be used on the resulting *file*

stream.

When *element-type* is :default, the type can be determined by using stream-element-type.

## Function OPEN-STREAM-P

**Syntax:**

**open-stream-p** *stream* => *generalized-boolean*

**Arguments and Values:**

*stream*—a stream.

*generalized-boolean*—a generalized boolean.

**Description:**

Returns true if *stream* is an open stream; otherwise, returns false.

Streams are open until they have been explicitly closed with close, or until they are implicitly closed due to exit from a with-output-to-string, with-open-file, with-input-from-string, or with-open-stream-form.

**Examples:**

```
(open-stream-p *standard-input*) => true
```

**Side Effects:** None.

**Affected By:**

close.

**Exceptional Situations:**

Should signal an error of type type-error if *stream* is not a stream.

**See Also:** None.

**Notes:** None.

*Standard Generic Function (SETF CLASS-NAME)*

**Syntax:**

**(setf class-name)** *new-value* *class* => *new-value*

**Method Signatures:**

**(setf class-name)** *new-value* (*class* class)

**Arguments and Values:**

*new-value*—a symbol.

*class*—a class.

**Description:**

The generic function (`(setf class-name)`) sets the name of a *class* object.

**Examples:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[find-class](#), [proper name](#), [Section 4.3 \(Classes\)](#)

**Notes:** None.

**Function PAIRLIS****Syntax:**

**pairlis** *keys data &optional alist => new-alist*

**Arguments and Values:**

*keys*—a proper list.

*data*—a proper list.

*alist*—an association list. The default is the empty list.

*new-alist*—an association list.

**Description:**

Returns an association list that associates elements of *keys* to corresponding elements of *data*. The consequences are undefined if *keys* and *data* are not of the same length.

If *alist* is supplied, **pairlis** returns a modified *alist* with the new pairs prepended to it. The new pairs may appear in the resulting association list in either forward or backward order. The result of

```
(pairlis '(one two) '(1 2) '((three . 3) (four . 19)))
```

might be

```
((one . 1) (two . 2) (three . 3) (four . 19))
```

## CLHS: Declaration DYNAMIC-EXTENT

or

```
((two . 2) (one . 1) (three . 3) (four . 19))
```

### Examples:

```
(setq keys '(1 2 3)
      data '("one" "two" "three")
      alist '((4 . "four")))
(pairlis keys data) => ((3 . "three") (2 . "two") (1 . "one"))
(pairlis keys data alist)
=> ((3 . "three") (2 . "two") (1 . "one") (4 . "four"))
alist => ((4 . "four"))
```

**Side Effects:** None.

**Affected By:** None.

### Exceptional Situations:

Should be prepared to signal an error of type type-error if *keys* and *data* are not proper lists.

### See Also:

#### acons

**Notes:** None.

## Function PARSE-NAMESTRING

### Syntax:

**parse-namestring** *thing* &*optional host default-pathname &key start end junk-allowed*

=> *pathname, position*

### Arguments and Values:

*thing*—a string, a pathname, or a stream associated with a file.

*host*—a valid pathname host, a logical host, or nil.

*default-pathname*—a pathname designator. The default is the value of \*default-pathname-defaults\*.

*start, end*—bounding index designators of *thing*. The defaults for *start* and *end* are 0 and nil, respectively.

*junk-allowed*—a generalized boolean. The default is false.

*pathname*—a pathname, or nil.

*position*—a bounding index designator for *thing*.

**Description:**

Converts *thing* into a *pathname*.

The *host* supplies a host name with respect to which the parsing occurs.

If *thing* is a *stream associated with a file*, processing proceeds as if the *pathname* used to open that *file* had been supplied instead.

If *thing* is a *pathname*, the *host* and the host component of *thing* are compared. If they match, two values are immediately returned: *thing* and *start*; otherwise (if they do not match), an error is signaled.

Otherwise (if *thing* is a *string*), **parse-namestring** parses the name of a *file* within the substring of *thing* bounded by *start* and *end*.

If *thing* is a *string* then the substring of *thing* *bounded* by *start* and *end* is parsed into a *pathname* as follows:

- \* If *host* is a *logical host* then *thing* is parsed as a *logical pathname namestring on the host*.
- \* If *host* is *nil* and *thing* is a syntactically valid *logical pathname namestring* containing an explicit host, then it is parsed as a *logical pathname namestring*.
- \* If *host* is *nil*, *default-pathname* is a *logical pathname*, and *thing* is a syntactically valid *logical pathname namestring* without an explicit host, then it is parsed as a *logical pathname namestring on the host* that is the host component of *default-pathname*.
- \* Otherwise, the parsing of *thing* is *implementation-defined*.

In the first of these cases, the host portion of the *logical pathname* namestring and its following *colon* are optional.

If the host portion of the namestring and *host* are both present and do not match, an error is signaled.

If *junk-allowed* is *true*, then the *primary value* is the *pathname* parsed or, if no syntactically correct *pathname* was seen, *nil*. If *junk-allowed* is *false*, then the entire substring is scanned, and the *primary value* is the *pathname* parsed.

In either case, the *secondary value* is the index into *thing* of the delimiter that terminated the parse, or the index beyond the substring if the parse terminated at the end of the substring (as will always be the case if *junk-allowed* is *false*).

Parsing a *null string* always succeeds, producing a *pathname* with all components (except the host) equal to *nil*.

If *thing* contains an explicit host name and no explicit device name, then it is *implementation-defined* whether **parse-namestring** will supply the standard default device for that host as the device component of the resulting *pathname*.

**Examples:**

```
(setq q (parse-namestring "test"))
=> #S(PATHNAME :HOST NIL :DEVICE NIL :DIRECTORY NIL :NAME "test"
      :TYPE NIL :VERSION NIL)
(pathnamep q) => true
(parse-namestring "test")
```

## CLHS: Declaration DYNAMIC-EXTENT

```
=> #S(PATHNAME :HOST NIL :DEVICE NIL :DIRECTORY NIL :NAME "test"
      :TYPE NIL :VERSION NIL), 4
(setq s (open xxx)) => #<Input File Stream...>
(parse-namestring s)
=> #S(PATHNAME :HOST NIL :DEVICE NIL :DIRECTORY NIL :NAME xxx
      :TYPE NIL :VERSION NIL), 0
(parse-namestring "test" nil nil :start 2 :end 4 )
=> #S(PATHNAME ...), 15
(parse-namestring "foo.lisp")
=> #P"foo.lisp"
```

**Affected By:** None.

### Exceptional Situations:

If *junk-allowed* is *false*, an error of **type parse-error** is signaled if *thing* does not consist entirely of the representation of a **pathname**, possibly surrounded on either side by **whitespace[1]** characters if that is appropriate to the cultural conventions of the implementation.

If *host* is supplied and not **nil**, and *thing* contains a manifest host name, an error of **type error** is signaled if the hosts do not match.

If *thing* is a **logical pathname** namestring and if the host portion of the namestring and *host* are both present and do not match, an error of **type error** is signaled.

### See Also:

**pathname**, **logical-pathname**, **Section 20.1 (File System Concepts)**, **Section 19.2.2.2.3 (:UNSPECIFIC as a Component Value)**, **Section 19.1.2 (Pathnames as Filenames)**

**Notes:** None.

## Function PARSE-INTEGER

### Syntax:

**parse-integer** *string &key start end radix junk-allowed => integer, pos*

### Arguments and Values:

*string*---a **string**.

*start*, *end*---**bounding index designators** of *string*. The defaults for *start* and *end* are 0 and **nil**, respectively.

*radix*---a **radix**. The default is 10.

*junk-allowed*---a **generalized boolean**. The default is *false*.

*integer*---an **integer** or *false*.

*pos*---a **bounding index** of *string*.

**Description:**

**parse-integer** parses an *integer* in the specified *radix* from the substring of *string* delimited by *start* and *end*.

**parse-integer** expects an optional sign (+ or -) followed by a non-empty sequence of digits to be interpreted in the specified *radix*. Optional leading and trailing *whitespace*[1] is ignored.

**parse-integer** does not recognize the syntactic radix-specifier prefixes #O, #B, #X, and #nR, nor does it recognize a trailing decimal point.

If *junk-allowed* is *false*, an error of *type parse-error* is signaled if substring does not consist entirely of the representation of a signed *integer*, possibly surrounded on either side by *whitespace*[1] *characters*.

The first *value* returned is either the *integer* that was parsed, or else **nil** if no syntactically correct *integer* was seen but *junk-allowed* was *true*.

The second *value* is either the index into the *string* of the delimiter that terminated the parse, or the upper *bounding index* of the substring if the parse terminated at the end of the substring (as is always the case if *junk-allowed* is *false*).

**Examples:**

```
(parse-integer "123") => 123, 3
(parse-integer "123" :start 1 :radix 5) => 13, 3
(parse-integer "no-integer" :junk-allowed t) => NIL, 0
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:**

If *junk-allowed* is *false*, an error is signaled if substring does not consist entirely of the representation of an *integer*, possibly surrounded on either side by *whitespace*[1] characters.

**See Also:** None.

**Notes:** None.

**Function PEEK-CHAR****Syntax:**

**peek-char** &optional *peek-type* *input-stream* *eof-error-p* *eof-value* *recursive-p* => *char*

**Arguments and Values:**

*peek-type*—a *character* or **t** or **nil**.

*input-stream*—*input stream designator*. The default is *standard input*.

*eof-error-p*—a generalized boolean. The default is true.

*eof-value*—an object. The default is nil.

*recursive-p*—a generalized boolean. The default is false.

*char*—a character or the *eof-value*.

### Description:

**peek-char** obtains the next character in *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.

If *peek-type* is not supplied or nil, **peek-char** returns the next character to be read from *input-stream*, without actually removing it from *input-stream*. The next time input is done from *input-stream*, the character will still be there. If *peek-type* is t, then **peek-char** skips over whitespace[2] characters, but not comments, and then performs the peeking operation on the next character. The last character examined, the one that starts an object, is not removed from *input-stream*. If *peek-type* is a character, then **peek-char** skips over input characters until a character that is char= to that character is found; that character is left in *input-stream*.

If an end of file[2] occurs and *eof-error-p* is false, *eof-value* is returned.

If *recursive-p* is true, this call is expected to be embedded in a higher-level call to **read** or a similar function used by the Lisp reader.

When *input-stream* is an echo stream, characters that are only peeked at are not echoed. In the case that *peek-type* is not nil, the characters that are passed by **peek-char** are treated as if by **read-char**, and so are echoed unless they have been marked otherwise by unread-char.

### Examples:

```
(with-input-from-string (input-stream "      1 2 3 4 5")
  (format t "~S ~S ~S"
          (peek-char t input-stream)
          (peek-char #\4 input-stream)
          (peek-char nil input-stream)))
>> #\1 #\4 #\4
=> NIL
```

### Affected By:

\*readtable\*, \*standard-input\*, \*terminal-io\*.

### Exceptional Situations:

If *eof-error-p* is true and an end of file[2] occurs an error of type end-of-file is signaled.

If *peek-type* is a character, an end of file[2] occurs, and *eof-error-p* is true, an error of type end-of-file is signaled.

If *recursive-p* is true and an end of file[2] occurs, an error of type end-of-file is signaled.

**See Also:** None.

**Notes:** None.

## Function PHASE

**Syntax:**

**phase** *number* => *phase*

**Arguments and Values:**

*number*—a number.

*phase*—a number.

**Description:**

**phase** returns the phase of *number* (the angle part of its polar representation) in radians, in the range  $-<\text{PI}>$  (exclusive) if minus zero is not supported, or  $-<\text{PI}>$  (inclusive) if minus zero is supported, to  $<\text{PI}>$  (inclusive). The phase of a positive real number is zero; that of a negative real number is  $<\text{PI}>$ . The phase of zero is defined to be zero.

If *number* is a complex float, the result is a float of the same type as the components of *number*. If *number* is a float, the result is a float of the same type. If *number* is a rational or a complex rational, the result is a single float.

The branch cut for **phase** lies along the negative real axis, continuous with quadrant II. The range consists of that portion of the real axis between  $-<\text{PI}>$  (exclusive) and  $<\text{PI}>$  (inclusive).

The mathematical definition of **phase** is as follows:

```
(phase x) = (atan (imagpart x) (realpart x))
```

**Examples:**

```
(phase 1) => 0.0s0
(phase 0) => 0.0s0
(phase (cis 30)) => -1.4159266
(phase #c(0 1)) => 1.5707964
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:**

Should signal type-error if its argument is not a number. Might signal arithmetic-error.

**See Also:**

**Notes:** None.

## Function PACKAGE-USED-BY-LIST

**Syntax:**

**package-used-by-list** *package* => *used-by-list*

**Arguments and Values:**

*package*—a package designator.

*used-by-list*—a list of package objects.

**Description:**

**package-used-by-list** returns a list of other packages that use *package*.

**Examples:**

```
(package-used-by-list (make-package 'temp)) => ()
(make-package 'trash :use '(temp)) => #<PACKAGE "TRASH">
(package-used-by-list 'temp) => (#<PACKAGE "TRASH">)
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:**

Should signal an error of type type-error if *package* is not a package.

**See Also:**

use-package, unuse-package

**Notes:** None.

## Function PACKAGE-ERROR-PACKAGE

**Syntax:**

**package-error-package** *condition* => *package*

**Arguments and Values:**

*condition*—a condition of type package-error.

*package*—a package designator.

**Description:**

## CLHS: Declaration DYNAMIC-EXTENT

Returns a designator for the offending package in the situation represented by the condition.

### Examples:

```
(package-error-package
  (make-condition 'package-error
    :package (find-package "COMMON-LISP" )))
=> #<Package "COMMON-LISP">
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

### See Also:

#### package-error

**Notes:** None.

## Function PACKAGE-NAME

### Syntax:

**package-name** *package* => *name*

### Arguments and Values:

*package*---a package designator.

*name*---a string or nil.

### Description:

**package-name** returns the string that names *package*, or nil if the *package designator* is a package object that has no name (see the function delete-package).

### Examples:

```
(in-package "COMMON-LISP-USER") => #<PACKAGE "COMMON-LISP-USER">
(package-name *package*) => "COMMON-LISP-USER"
(package-name (symbol-package :test)) => "KEYWORD"
(package-name (find-package 'common-lisp)) => "COMMON-LISP"

(defvar *foo-package* (make-package "FOO"))
(rename-package "FOO" "FOO0")
(package-name *foo-package*) => "FOO0"
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:**

Should signal an error of *type type-error* if *package* is not a *package designator*.

**See Also:** None.

**Notes:** None.

***Function PACKAGE-NICKNAMES*****Syntax:**

**package-nicknames** *package* => *nicknames*

**Arguments and Values:**

*package*—a *package designator*.

*nicknames*—a *list of strings*.

**Description:**

Returns the *list* of nickname *strings* for *package*, not including the name of *package*.

**Examples:**

```
(package-nicknames (make-package 'temporary
                                   :nicknames '("TEMP" "temp"))
=> ("temp" "TEMP") )
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:**

Should signal an error of *type type-error* if *package* is not a *package designator*.

**See Also:** None.

**Notes:** None.

***Function PACKAGE-SHADOWING-SYMBOLS*****Syntax:**

**package-shadowing-symbols** *package* => *symbols*

**Arguments and Values:**

*package*—a *package designator*.

*symbols*---a list of *symbols*.

#### Description:

Returns a list of *symbols* that have been declared as shadowing symbols in *package* by shadow or shadowing-import (or the equivalent defpackage options). All *symbols* on this list are present in *package*.

#### Examples:

```
(package-shadowing-symbols (make-package 'temp)) => ()
(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)
```

**Side Effects:** None.

**Affected By:** None.

#### Exceptional Situations:

Should signal an error of type type-error if *package* is not a package designator.

#### See Also:

shadow, shadowing-import

#### Notes:

Whether the list of *symbols* is fresh is implementation-dependent.

## Function PACKAGE-USE-LIST

#### Syntax:

**package-use-list** *package* => *use-list*

#### Arguments and Values:

*package*---a package designator.

*use-list*---a list of package objects.

#### Description:

Returns a list of other packages used by *package*.

#### Examples:

```
(package-use-list (make-package 'temp)) => (#<PACKAGE "COMMON-LISP">)
(use-package 'common-lisp-user 'temp) => T
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(package-use-list 'temp) => (#<PACKAGE "COMMON-LISP"> #<PACKAGE "COMMON-LISP-USER">)
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:**

Should signal an error of type type-error if *package* is not a package designator.

**See Also:**

use-package, unuse-package

**Notes:** None.

## Function PACKAGEP

**Syntax:**

**packagep** *object* => *generalized-boolean*

**Arguments and Values:**

*object*---an object.

*generalized-boolean*---a generalized boolean.

**Description:**

Returns true if *object* is of type package; otherwise, returns false.

**Examples:**

```
(packagep *package*) => true
(packagep 'common-lisp) => false
(packagep (find-package 'common-lisp)) => true
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:** None.

**Notes:**

```
(packagep object) == (typep object 'package)
```

***Function +*****Syntax:**

`+ &rest numbers => sum`

**Arguments and Values:**

*number*—a number.

*sum*—a number.

**Description:**

Returns the sum of *numbers*, performing any necessary type conversions in the process. If no *numbers* are supplied, 0 is returned.

**Examples:**

```
(+) => 0
(+ 1) => 1
(+ 31/100 69/100) => 1
(+ 1/5 0.8) => 1.0
```

**Affected By:** None.

**Exceptional Situations:**

Might signal type-error if some *argument* is not a number. Might signal arithmetic-error.

**See Also:**

**Notes:** None.

***Function PATHNAME*****Syntax:**

`pathname pathspec => pathname`

**Arguments and Values:**

*pathspec*—a pathname designator.

*pathname*—a pathname.

**Description:**

Returns the pathname denoted by *pathspec*.

## CLHS: Declaration DYNAMIC-EXTENT

If the *pathspec designator* is a *stream*, the *stream* can be either open or closed; in both cases, the **pathname** returned corresponds to the *filename* used to open the *file*. **pathname** returns the same *pathname* for a *file stream* after it is closed as it did when it was open.

If the *pathspec designator* is a *file stream* created by opening a *logical pathname*, a *logical pathname* is returned.

### Examples:

```
; There is a great degree of variability permitted here. The next
; several examples are intended to illustrate just a few of the many
; possibilities. Whether the name is canonicalized to a particular
; case (either upper or lower) depends on both the file system and the
; implementation since two different implementations using the same
; file system might differ on many issues. How information is stored
; internally (and possibly presented in #S notation) might vary,
; possibly requiring `accessors' such as PATHNAME-NAME to perform case
; conversion upon access. The format of a namestring is dependent both
; on the file system and the implementation since, for example, one
; implementation might include the host name in a namestring, and
; another might not. #S notation would generally only be used in a
; situation where no appropriate namestring could be constructed for use
; with #P.

(setq p1 (pathname "test"))
=> #P"CHOCOLATE:TEST" ; with case canonicalization (e.g., VMS)
OR=> #P"VANILLA:test"    ; without case canonicalization (e.g., Unix)
OR=> #P"test"
OR=> #S(PATHNAME :HOST "STRAWBERRY" :NAME "TEST")
OR=> #S(PATHNAME :HOST "BELGIAN-CHOCOLATE" :NAME "test")
  (setq p2 (pathname "test"))
=> #P"CHOCOLATE:TEST"
OR=> #P"VANILLA:test"
OR=> #P"test"
OR=> #S(PATHNAME :HOST "STRAWBERRY" :NAME "TEST")
OR=> #S(PATHNAME :HOST "BELGIAN-CHOCOLATE" :NAME "test")
  (pathnamep p1) => true
  (eq p1 (pathname p1)) => true
  (eq p1 p2)
=> true
OR=> false
  (with-open-file (stream "test" :direction :output)
    (pathname stream))
=> #P"ORANGE-CHOCOLATE:>Gus>test.lisp.newest"
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[pathname](#), [logical-pathname](#), [Section 20.1 \(File System Concepts\)](#), [Section 19.1.2 \(Pathnames as Filenames\)](#)

**Notes:** None.

**Function PATHNAME-HOST, PATHNAME-DEVICE,  
PATHNAME-DIRECTORY, PATHNAME-NAME, PATHNAME-TYPE,  
PATHNAME-VERSION**

**Syntax:**

**pathname-host** *pathname &key case => host*

**pathname-device** *pathname &key case => device*

**pathname-directory** *pathname &key case => directory*

**pathname-name** *pathname &key case => name*

**pathname-type** *pathname &key case => type*

**pathname-version** *pathname => version*

**Arguments and Values:**

*pathname*—a pathname designator.

*case*—one of :local or :common. The default is :local.

*host*—a valid pathname host.

*device*—a valid pathname device.

*directory*—a valid pathname directory.

*name*—a valid pathname name.

*type*—a valid pathname type.

*version*—a valid pathname version.

**Description:**

These functions return the components of *pathname*.

If the *pathname designator* is a *pathname*, it represents the name used to open the file. This may be, but is not required to be, the actual name of the file.

If *case* is supplied, it is treated as described in [Section 19.2.2.1.2 \(Case in Pathname Components\)](#).

**Examples:**

```
(setq q (make-pathname :host "KATHY"
                      :directory "CHAPMAN"
                      :name "LOGIN" :type "COM"))
=> #P"KATHY:[CHAPMAN]LOGIN.COM"
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(pathname-host q) => "KATHY"
(pathname-name q) => "LOGIN"
(pathname-type q) => "COM"

;; Because namestrings are used, the results shown in the remaining
;; examples are not necessarily the only possible results.  Mappings
;; from namestring representation to pathname representation are
;; dependent both on the file system involved and on the implementation
;; (since there may be several implementations which can manipulate the
;; the same file system, and those implementations are not constrained
;; to agree on all details). Consult the documentation for each
;; implementation for specific information on how namestrings are treated
;; that implementation.

;; VMS
(pathname-directory (parse-namestring "[FOO.*.BAR]BAZ.LSP"))
=> (:ABSOLUTE "FOO" "BAR")
(pathname-directory (parse-namestring "[FOO.*.BAR]BAZ.LSP") :case :common)
=> (:ABSOLUTE "FOO" "BAR")

;; Unix
(pathname-directory "foo.l") => NIL
(pathname-device "foo.l") => :UNSPECIFIC
(pathname-name "foo.l") => "foo"
(pathname-name "foo.l" :case :local) => "foo"
(pathname-name "foo.l" :case :common) => "FOO"
(pathname-type "foo.l") => "l"
(pathname-type "foo.l" :case :local) => "l"
(pathname-type "foo.l" :case :common) => "L"
(pathname-type "foo") => :UNSPECIFIC
(pathname-type "foo" :case :common) => :UNSPECIFIC
(pathname-type "foo.") => ""
(pathname-type "foo." :case :common) => ""
(pathname-directory (parse-namestring "/foo/bar/baz.lisp") :case :local)
=> (:ABSOLUTE "foo" "bar")
(pathname-directory (parse-namestring "/foo/bar/baz.lisp") :case :local)
=> (:ABSOLUTE "FOO" "BAR")
(pathname-directory (parse-namestring "../baz.lisp"))
=> (:RELATIVE :UP)
(PATHNAME-DIRECTORY (PARSE-NAMESTRING "/foo/BAR/..Mum/baz"))
=> (:ABSOLUTE "foo" "BAR" :UP "Mum")
(PATHNAME-DIRECTORY (PARSE-NAMESTRING "/foo/BAR/..Mum/baz") :case :common)
=> (:ABSOLUTE "FOO" "bar" :UP "Mum")
(PATHNAME-DIRECTORY (PARSE-NAMESTRING "/foo/*/bar/baz.l"))
=> (:ABSOLUTE "foo" :WILD "bar")
(PATHNAME-DIRECTORY (PARSE-NAMESTRING "/foo/*/bar/baz.l") :case :common)
=> (:ABSOLUTE "FOO" :WILD "BAR")

;; Symbolics LMFS
(pathname-directory (parse-namestring ">foo>**>bar>baz.lisp"))
=> (:ABSOLUTE "foo" :WILD-INFERIORS "bar")
(pathname-directory (parse-namestring ">foo>*>bar>baz.lisp"))
=> (:ABSOLUTE "foo" :WILD "bar")
(pathname-directory (parse-namestring ">foo>*>bar>baz.lisp") :case :common)
=> (:ABSOLUTE "FOO" :WILD "BAR")
(pathname-device (parse-namestring ">foo>baz.lisp")) => :UNSPECIFIC
```

### Affected By:

The implementation and the host file system.

**Exceptional Situations:**

Should signal an error of [type type-error](#) if its first argument is not a [pathname](#).

**See Also:**

[pathname](#), [logical-pathname](#), [Section 20.1 \(File System Concepts\)](#), [Section 19.1.2 \(Pathnames as Filenames\)](#)

**Notes:** None.

## **Function PATHNAME-MATCH-P**

**Syntax:**

[pathname-match-p](#) *pathname wildcard => generalized-boolean*

**Arguments and Values:**

*pathname*—a [pathname designator](#).

*wildcard*—a [designator](#) for a [wild pathname](#).

*generalized-boolean*—a [generalized boolean](#).

**Description:**

[pathname-match-p](#) returns true if *pathname* matches *wildcard*, otherwise [nil](#). The matching rules are [implementation-defined](#) but should be consistent with [directory](#). Missing components of *wildcard* default to `:wild`.

It is valid for *pathname* to be a wild [pathname](#); a wildcard field in *pathname* only matches a wildcard field in *wildcard* (i.e., [pathname-match-p](#) is not commutative). It is valid for *wildcard* to be a non-wild [pathname](#).

**Examples:** None.

**Affected By:** None.

**Exceptional Situations:**

If *pathname* or *wildcard* is not a [pathname](#), [string](#), or [stream associated with a file](#) an error of [type type-error](#) is signaled.

**See Also:**

[directory](#), [pathname](#), [logical-pathname](#), [Section 20.1 \(File System Concepts\)](#), [Section 19.1.2 \(Pathnames as Filenames\)](#)

**Notes:** None.

**Function PATHNAMEP****Syntax:****pathnamep object => generalized-boolean****Arguments and Values:***object*—an object.*generalized-boolean*—a generalized boolean.**Description:**Returns true if *object* is of type **pathname**; otherwise, returns false.**Examples:**

```
(setq q "test")  => "test"
(pathnamep q) => false
(setq q (pathname "test"))
=> #S(PATHNAME :HOST NIL :DEVICE NIL :DIRECTORY NIL :NAME "test" :TYPE NIL
      :VERSION NIL)
(pathnamep q) => true
(setq q (logical-pathname "SYS:SITE;FOO.SYSTEM"))
=> #P"SYS:SITE;FOO.SYSTEM"
(pathnamep q) => true
```

**Side Effects:** None.**Affected By:** None.**Exceptional Situations:** None.**See Also:** None.**Notes:**

(pathnamep object) == (typep object 'pathname)

**Function POSITION, POSITION-IF, POSITION-IF-NOT****Syntax:****position item sequence &key from-end test test-not start end key => position****position-if predicate sequence &key from-end start end key => position****position-if-not predicate sequence &key from-end start end key => position**

**Arguments and Values:**

*item*—an object.

*sequence*—a proper sequence.

*predicate*—a designator for a function of one argument that returns a generalized boolean.

*from-end*—a generalized boolean. The default is false.

*test*—a designator for a function of two arguments that returns a generalized boolean.

*test-not*—a designator for a function of two arguments that returns a generalized boolean.

*start, end*—bounding index designators of *sequence*. The defaults for *start* and *end* are 0 and nil, respectively.

*key*—a designator for a function of one argument, or nil.

*position*—a bounding index of *sequence*, or nil.

**Description:**

**position**, **position-if**, and **position-if-not** each search *sequence* for an element that *satisfies the test*.

The *position* returned is the index within *sequence* of the leftmost (if *from-end* is true) or of the rightmost (if *from-end* is false) element that *satisfies the test*; otherwise nil is returned. The index returned is relative to the left-hand end of the entire *sequence*, regardless of the value of *start*, *end*, or *from-end*.

**Examples:**

```
(position #\a "baobab" :from-end t) => 4
(position-if #'oddp '(1) (2) (3) (4)) :start 1 :key #'car) => 2
(position 595 '()) => NIL
(position-if-not #'integerp '(1 2 3 4 5.0)) => 4
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:**

Should be prepared to signal an error of type type-error if *sequence* is not a proper sequence.

**See Also:**

**find**, [Section 3.6 \(Traversal Rules and Side Effects\)](#)

**Notes:**

The :*test-not* argument is deprecated.

The *function position-if-not* is deprecated.

## Function PPRINT-DISPATCH

### Syntax:

**pprint-dispatch** *object &optional table => function, found-p*

### Arguments and Values:

*object*—an *object*.

*table*—a *pprint dispatch table*, or *nil*. The default is the *value* of \*print-pprint-dispatch\*.

*function*—a *function designator*.

*found-p*—a *generalized boolean*.

### Description:

Retrieves the highest priority function in *table* that is associated with a *type specifier* that matches *object*. The function is chosen by finding all of the *type specifiers* in *table* that match the *object* and selecting the highest priority function associated with any of these *type specifiers*. If there is more than one highest priority function, an arbitrary choice is made. If no *type specifiers* match the *object*, a function is returned that prints *object* using *print-object*.

The *secondary value*, *found-p*, is *true* if a matching *type specifier* was found in *table*, or *false* otherwise.

If *table* is *nil*, retrieval is done in the *initial pprint dispatch table*.

**Examples:** None.

**Side Effects:** None.

### Affected By:

The state of the *table*.

### Exceptional Situations:

Should signal an error of *type type-error* if *table* is neither a *pprint-dispatch-table* nor *nil*.

**See Also:** None.

### Notes:

```
(let ((*print-pretty* t))
  (write object :stream s))
== (funcall (pprint-dispatch object) s object)
```

**Function PPRINT-FILL, PPRINT-LINEAR, PPRINT-TABULAR****Syntax:**

**pprint-fill** *stream object &optional colon-p at-sign-p => nil*

**pprint-linear** *stream object &optional colon-p at-sign-p => nil*

**pprint-tabular** *stream object &optional colon-p at-sign-p tabspace => nil*

**Arguments and Values:**

*stream*—an output stream designator.

*object*—an object.

*colon-p*—a generalized boolean. The default is true.

*at-sign-p*—a generalized boolean. The default is implementation-dependent.

*tabspace*—a non-negative integer. The default is 16.

**Description:**

The functions **pprint-fill**, **pprint-linear**, and **pprint-tabular** specify particular ways of *pretty printing a list* to *stream*. Each function prints parentheses around the output if and only if *colon-p* is true. Each function ignores its *at-sign-p* argument. (Both arguments are included even though only one is needed so that these functions can be used via `~/ . . . /` and as **set-pprint-dispatch** functions, as well as directly.) Each function handles abbreviation and the detection of circularity and sharing correctly, and uses **write** to print *object* when it is a non-list.

If *object* is a list and if the value of **\*print-pretty\*** is false, each of these functions prints *object* using a minimum of whitespace, as described in Section 22.1.3.5 (Printing Lists and Conses). Otherwise (if *object* is a list and if the value of **\*print-pretty\*** is true):

- The function pprint-linear prints a list either all on one line, or with each element on a separate line.
- The function pprint-fill prints a list with as many elements as possible on each line.
- The function pprint-tabular is the same as **pprint-fill** except that it prints the elements so that they line up in columns. The *tabspace* specifies the column spacing in ems, which is the total spacing from the leading edge of one column to the leading edge of the next.

**Examples:**

Evaluating the following with a line length of 25 produces the output shown.

```
(progn (princ "Roads ")
          (pprint-tabular *standard-output* '(elm main maple center) nil nil 8))
Roads ELM      MAIN
      MAPLE    CENTER
```

**Side Effects:**

Performs output to the indicated *stream*.

**Affected By:**

The cursor position on the indicated *stream*, if it can be determined.

**Exceptional Situations:** None.

**See Also:** None.

**Notes:**

The *function pprint-tabular* could be defined as follows:

```
(defun pprint-tabular (s list &optional (colon-p t) at-sign-p (tabsize nil))
  (declare (ignore at-sign-p))
  (when (null tabsize) (setq tabsize 16))
  (pprint-logical-block (s list :prefix (if colon-p "  ")
                           :suffix (if colon-p ")" " ")))
  (pprint-exit-if-list-exhausted)
  (loop (write (pprint-pop) :stream s)
        (pprint-exit-if-list-exhausted)
        (write-char #\Space s)
        (pprint-tab :section-relative 0 tabsize s)
        (pprint-newline :fill s))))
```

Note that it would have been inconvenient to specify this function using *format*, because of the need to pass its *tabsize* argument through to a  $\sim :T$  format directive nested within an iteration over a list.

## Function PPRINT-INDENT

**Syntax:**

**pprint-indent** *relative-to n &optional stream => nil*

**Arguments and Values:**

*relative-to*—either `:block` or `:current`.

*n*—a *real*.

*stream*—an *output stream designator*. The default is *standard output*.

**Description:**

**pprint-indent** specifies the indentation to use in a logical block on *stream*. If *stream* is a *pretty printing stream* and the *value* of *\*print-pretty\** is *true*, **pprint-indent** sets the indentation in the innermost dynamically enclosing logical block; otherwise, **pprint-indent** has no effect.

*N* specifies the indentation in *ems*. If *relative-to* is `:block`, the indentation is set to the horizontal position of the first character in the *dynamically current logical block* plus *n.ems*. If *relative-to* is `:current`, the indentation is set to the current output position plus *n.ems*. (For robustness in the face of variable-width fonts,

## CLHS: Declaration DYNAMIC-EXTENT

it is advisable to use :current with an *n* of zero whenever possible.)

*N* can be negative; however, the total indentation cannot be moved left of the beginning of the line or left of the end of the rightmost per-line prefix---an attempt to move beyond one of these limits is treated the same as an attempt to move to that limit. Changes in indentation caused by *pprint-indent* do not take effect until after the next line break. In addition, in miser mode all calls to **pprint-indent** are ignored, forcing the lines corresponding to the logical block to line up under the first character in the block.

**Examples:** None.

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:**

An error is signaled if *relative-to* is any *object* other than :block or :current.

**See Also:**

**Notes:** None.

## Function PPRINT-NEWLINE

**Syntax:**

**pprint-newline** *kind &optional stream => nil*

**Arguments and Values:**

*kind*---one of :linear, :fill, :miser, or :mandatory.

*stream*---a *stream designator*. The default is *standard output*.

**Description:**

If *stream* is a *pretty printing stream* and the *value* of **\*print-pretty\*** is *true*, a line break is inserted in the output when the appropriate condition below is satisfied; otherwise, **pprint-newline** has no effect.

*Kind* specifies the style of conditional newline. This *parameter* is treated as follows:

:linear

This specifies a "linear-style" *conditional newline*. A line break is inserted if and only if the immediately containing *section* cannot be printed on one line. The effect of this is that line breaks are either inserted at every linear-style conditional newline in a logical block or at none of them.

:miser

This specifies a "miser-style" *conditional newline*. A line break is inserted if and only if the immediately containing *section* cannot be printed on one line and miser style is in effect in the immediately containing logical block. The effect of this is that miser-style conditional newlines act like linear-style conditional newlines, but only when miser style is in effect. Miser style is in effect for a logical block if and only if the starting position of the logical block is less than or equal to

**\*print-miser-width\*** *ems* from the right margin.

:*fill*

This specifies a "fill-style" *conditional newline*. A line break is inserted if and only if either (a) the following *section* cannot be printed on the end of the current line, (b) the preceding *section* was not printed on a single line, or (c) the immediately containing *section* cannot be printed on one line and miser style is in effect in the immediately containing logical block. If a logical block is broken up into a number of subsections by fill-style conditional newlines, the basic effect is that the logical block is printed with as many subsections as possible on each line. However, if miser style is in effect, fill-style conditional newlines act like linear-style conditional newlines.

:*mandatory*

This specifies a "mandatory-style" *conditional newline*. A line break is always inserted. This implies that none of the containing *sections* can be printed on a single line and will therefore trigger the insertion of line breaks at linear-style conditional newlines in these *sections*.

When a line break is inserted by any type of conditional newline, any blanks that immediately precede the conditional newline are omitted from the output and indentation is introduced at the beginning of the next line. By default, the indentation causes the following line to begin in the same horizontal position as the first character in the immediately containing logical block. (The indentation can be changed via **pprint-indent**.)

There are a variety of ways unconditional newlines can be introduced into the output (i.e., via **terpri** or by printing a string containing a newline character). As with mandatory conditional newlines, this prevents any of the containing *sections* from being printed on one line. In general, when an unconditional newline is encountered, it is printed out without suppression of the preceding blanks and without any indentation following it. However, if a per-line prefix has been specified (see **pprint-logical-block**), this prefix will always be printed no matter how a newline originates.

### Examples:

See [Section 22.2.2 \(Examples of using the Pretty Printer\)](#).

### Side Effects:

Output to *stream*.

### Affected By:

**\*print-pretty\***, **\*print-miser\***. The presence of containing logical blocks. The placement of newlines and conditional newlines.

### Exceptional Situations:

An error of ***type type-error*** is signaled if *kind* is not one of :linear, :fill, :miser, or :mandatory.

### See Also:

**Notes:** None.

## Function PPRINT-TAB

### Syntax:

## CLHS: Declaration DYNAMIC-EXTENT

**pprint-tab** *kind colnum colinc &optional stream => nil*

### Arguments and Values:

*kind*—one of :line, :section, :line-relative, or :section-relative.

*colnum*—a non-negative integer.

*colinc*—a non-negative integer.

*stream*—an output stream designator.

### Description:

Specifies tabbing to *stream* as performed by the standard ~T format directive. If *stream* is a pretty printing stream and the value of \*print-pretty\* is true, tabbing is performed; otherwise, **pprint-tab** has no effect.

The arguments *colnum* and *colinc* correspond to the two *parameters* to ~T and are in terms of ems. The *kind* argument specifies the style of tabbing. It must be one of :line (tab as by ~T), :section (tab as by ~:T, but measuring horizontal positions relative to the start of the dynamically enclosing section), :line-relative (tab as by ~@T), or :section-relative (tab as by ~:@T, but measuring horizontal positions relative to the start of the dynamically enclosing section).

**Examples:** None.

**Side Effects:** None.

**Affected By:** None.

### Exceptional Situations:

An error is signaled if *kind* is not one of :line, :section, :line-relative, or :section-relative.

### See Also:

**pprint-logical-block**

**Notes:** None.

## Function PRINT-NOT-READABLE-OBJECT

### Syntax:

**print-not-readable-object** *condition => object*

### Arguments and Values:

*condition*—a condition of type **print-not-readable**.

*object*—an object.

**Description:**

Returns the *object* that could not be printed readably in the situation represented by *condition*.

**Examples:** None.**Affected By:** None.**Exceptional Situations:** None.**See Also:**

[print-not-readable](#), [Section 9 \(Conditions\)](#)

**Notes:** None.

## **Standard Generic Function PRINT-OBJECT**

**Syntax:**

**print-object** *object stream => object*

**Method Signatures:**

**print-object** (*object standard-object*) *stream*

**print-object** (*object structure-object*) *stream*

**Arguments and Values:**

*object*—an *object*.

*stream*—a *stream*.

**Description:**

The generic function print-object writes the printed representation of *object* to *stream*. The function print-object is called by the Lisp printer; it should not be called by the user.

Each implementation is required to provide a method on the class standard-object and on the class structure-object. In addition, each implementation must provide methods on enough other classes so as to ensure that there is always an applicable method. Implementations are free to add methods for other classes. Users may write methods for print-object for their own classes if they do not wish to inherit an implementation-dependent method.

The method on the class structure-object prints the object in the default #S notation; see [Section 22.1.3.12 \(Printing Structures\)](#).

Methods on print-object are responsible for implementing their part of the semantics of the printer control variables, as follows:

**\*print-readably\***

All methods for **print-object** must obey **\*print-readably\***. This includes both user-defined methods and *implementation-defined* methods. Readable printing of *structures* and *standard objects* is controlled by their **print-object** method, not by their **make-load-form method**. *Similarity* for these *objects* is application dependent and hence is defined to be whatever these *methods* do; see Section 3.2.4.2 (Similarity of Literal Objects).

**\*print-escape\***

Each *method* must implement **\*print-escape\***.

**\*print-pretty\***

The *method* may wish to perform specialized line breaking or other output conditional on the *value* of **\*print-pretty\***. For further information, see (for example) the **macro pprint-fill**. See also Section 22.2.1.4 (Pretty Print Dispatch Tables) and Section 22.2.2 (Examples of using the Pretty Printer).

**\*print-length\***

*Methods* that produce output of indefinite length must obey **\*print-length\***. For further information, see (for example) the **macros pprint-logical-block** and **pprint-pop**. See also Section 22.2.1.4 (Pretty Print Dispatch Tables) and Section 22.2.2 (Examples of using the Pretty Printer).

**\*print-level\***

The printer takes care of **\*print-level\*** automatically, provided that each *method* handles exactly one level of structure and calls **write** (or an equivalent *function*) recursively if there are more structural levels. The printer's decision of whether an *object* has components (and therefore should not be printed when the printing depth is not less than **\*print-level\***) is *implementation-dependent*. In some implementations its **print-object** *method* is not called; in others the *method* is called, and the determination that the *object* has components is based on what it tries to write to the *stream*.

**\*print-circle\***

When the *value* of **\*print-circle\*** is *true*, a user-defined **print-object** *method* can print *objects* to the supplied *stream* using **write**, **prin1**, **princ**, or **format** and expect circularities to be detected and printed using the #n# syntax. If a user-defined **print-object** *method* prints to a *stream* other than the one that was supplied, then circularity detection starts over for that *stream*. See **\*print-circle\***.

**\*print-base\*, \*print-radix\*, \*print-case\*, \*print-gensym\*, and \*print-array\***

These *printer control variables* apply to specific types of *objects* and are handled by the *methods* for those *objects*.

If these rules are not obeyed, the results are undefined.

In general, the printer and the **print-object** methods should not rebind the print control variables as they operate recursively through the structure, but this is *implementation-dependent*.

In some implementations the *stream* argument passed to a **print-object** *method* is not the original *stream*, but is an intermediate *stream* that implements part of the printer. *methods* should therefore not depend on the identity of this *stream*.

**Examples:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**pprint-fill**, **pprint-logical-block**, **pprint-pop**, **write**, **\*print-readably\***, **\*print-escape\***, **\*print-pretty\***, **\*print-length\***, Section 22.1.3 (Default Print-Object Methods), Section 22.1.3.12 (Printing Structures),

[Section 22.2.1.4 \(Pretty Print Dispatch Tables\)](#), [Section 22.2.2 \(Examples of using the Pretty Printer\)](#)

**Notes:** None.

## **Function PROBE-FILE**

**Syntax:**

**probe-file** *pathspec => truename*

**Arguments and Values:**

*pathspec*—a pathname designator.

*truename*—a physical pathname or nil.

**Description:**

**probe-file** tests whether a file exists.

**probe-file** returns false if there is no file named *pathspec*, and otherwise returns the truename of *pathspec*.

If the *pathspec designator* is an open stream, then **probe-file** produces the truename of its associated file. If *pathspec* is a stream, whether open or closed, it is coerced to a pathname as if by the function pathname.

**Examples:** None.

**Affected By:**

The host computer's file system.

**Exceptional Situations:**

An error of type file-error is signaled if *pathspec* is wild.

An error of type file-error is signaled if the file system cannot perform the requested operation.

**See Also:**

truename, open, ensure-directories-exist, pathname, logical-pathname, [Section 20.1 \(File System Concepts\)](#), [Section 20.1.2 \(File Operations on Open and Closed Streams\)](#), [Section 19.1.2 \(Pathnames as Filenames\)](#)

**Notes:** None.

## **Function PROCLAIM**

**Syntax:**

**proclaim** *declaration-specifier => implementation-dependent*

**Arguments and Values:**

*declaration-specifier*—a declaration specifier.

**Description:**

*Establishes* the declaration specified by *declaration-specifier* in the global environment.

Such a declaration, sometimes called a global declaration or a proclamation, is always in force unless locally shadowed.

Names of variables and functions within *declaration-specifier* refer to dynamic variables and global function definitions, respectively.

The next figure shows a list of *declaration identifiers* that can be used with proclaim.

<u>declaration</u>	<u>inline</u>	<u>optimize</u>	<u>type</u>
<u>ftype</u>	<u>notinline</u>	<u>special</u>	

**Figure 3–22. Global Declaration Specifiers**

An implementation is free to support other (implementation-defined) *declaration identifiers* as well.

**Examples:**

```
(defun declare-variable-types-globally (type vars)
  (proclaim `(type ,type ,@vars))
  type)

;; Once this form is executed, the dynamic variable *TOLERANCE*
;; must always contain a float.
(declare-variable-types-globally 'float '(*tolerance*))
=> FLOAT
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

declaim, declare, Section 3.2 (Compilation)

**Notes:**

Although the *execution* of a proclaim form has effects that might affect compilation, the compiler does not make any attempt to recognize and specially process proclaim forms. A proclamation such as the following, even if a top level form, does not have any effect until it is executed:

```
(proclaim '(special *x*))
```

If compile time side effects are desired, eval-when may be useful. For example:

```
(eval-when (:execute :compile-toplevel :load-toplevel))
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(proclaim '(special *x*)) )
```

In most such cases, however, it is preferable to use **declaim** for this purpose.

Since **proclaim forms** are ordinary *function forms*, *macro forms* can expand into them.

## **Function PROVIDE, REQUIRE**

### Syntax:

**provide** *module-name* => *implementation-dependent*

**require** *module-name* &*optional pathname-list* => *implementation-dependent*

### Arguments and Values:

*module-name*—a *string designator*.

*pathname-list*—**nil**, or a *designator* for a *non-empty list* of *pathname designators*. The default is **nil**.

### Description:

**provide** adds the *module-name* to the *list* held by **\*modules\***, if such a name is not already present.

**require** tests for the presence of the *module-name* in the *list* held by **\*modules\***. If it is present, **require** immediately returns. Otherwise, an attempt is made to load an appropriate set of *files* as follows: The *pathname-list* argument, if *non-nil*, specifies a list of *pathnames* to be loaded in order, from left to right. If the *pathname-list* is **nil**, an *implementation-dependent* mechanism will be invoked in an attempt to load the module named *module-name*; if no such module can be loaded, an error of **type error** is signaled.

Both functions use **string=** to test for the presence of a *module-name*.

### Examples:

```
;; This illustrates a nonportable use of REQUIRE, because it
;; depends on the implementation-dependent file-loading mechanism.

(require "CALCULUS")

;; This use of REQUIRE is nonportable because of the literal
;; physical pathname.

(require "CALCULUS" "/usr/lib/lisp/calculus")

;; One form of portable usage involves supplying a logical pathname,
;; with appropriate translations defined elsewhere.

(require "CALCULUS" "lib:calculus")

;; Another form of portable usage involves using a variable or
;; table lookup function to determine the pathname, which again
;; must be initialized elsewhere.
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(require "CALCULUS" *calculus-module-pathname*)
```

### Side Effects:

**provide** modifies **\*modules\***.

### Affected By:

The specific action taken by **require** is affected by calls to **provide** (or, in general, any changes to the *value* of **\*modules\***).

### Exceptional Situations:

Should signal an error of *type type-error* if *module-name* is not a *string designator*.

If **require** fails to perform the requested operation due to a problem while interacting with the *file system*, an error of *type file-error* is signaled.

An error of *type file-error* might be signaled if any *pathname* in *pathname-list* is a *designator* for a *wild pathname*.

### See Also:

**\*modules\***, Section 19.1.2 (Pathnames as Filenames)

### Notes:

The functions **provide** and **require** are deprecated.

If a module consists of a single *package*, it is customary for the package and module names to be the same.

## Function RANDOM

### Syntax:

**random** *limit* &*optional random-state => random-number*

### Arguments and Values:

*limit*—a positive *integer*, or a positive *float*.

*random-state*—a *random state*. The default is the *current random state*.

*random-number*—a non-negative *number* less than *limit* and of the same *type* as *limit*.

### Description:

Returns a pseudo-random number that is a non-negative *number* less than *limit* and of the same *type* as *limit*.

The *random-state*, which is modified by this function, encodes the internal state maintained by the random number generator.

## CLHS: Declaration DYNAMIC-EXTENT

An approximately uniform choice distribution is used. If *limit* is an integer, each of the possible results occurs with (approximate) probability 1/*limit*.

### Examples:

```
(<= 0 (random 1000) 1000) => true
(let ((state1 (make-random-state))
      (state2 (make-random-state)))
  (= (random 1000 state1) (random 1000 state2))) => true
```

### Side Effects:

The *random-state* is modified.

**Affected By:** None.

### Exceptional Situations:

Should signal an error of type-type-error if *limit* is not a positive integer or a positive real.

### See Also:

make-random-state, \*random-state\*

### Notes:

See *Common Lisp: The Language* for information about generating random numbers.

## Function RASSOC, RASSOC-IF, RASSOC-IF-NOT

### Syntax:

**rassoc** *item alist &key key test test-not => entry*

**rassoc-if** *predicate alist &key key => entry*

**rassoc-if-not** *predicate alist &key key => entry*

### Arguments and Values:

*item*—an object.

*alist*—an association list.

*predicate*—a designator for a function of one argument that returns a generalized boolean.

*test*—a designator for a function of two arguments that returns a generalized boolean.

*test-not*—a designator for a function of two arguments that returns a generalized boolean.

*key*—a designator for a function of one argument, or nil.

*entry*—a cons that is an element of the *alist*, or nil.

### Description:

**rassoc**, **rassoc-if**, and **rassoc-if-not** return the first cons whose cdr satisfies the test. If no such cons is found, nil is returned.

If nil appears in *alist* in place of a pair, it is ignored.

### 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 'a '((a . b) (b . c) (c . a) (z . a))) => (C . A)
(rassoc-if #'stringp alist) => (1 . "one")
(rassoc-if-not #'vectorp alist) => (3 . 3)
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

### See Also:

assoc, Section 3.6 (Traversal Rules and Side Effects)

### Notes:

The :test-not parameter is deprecated.

The function rassoc-if-not is deprecated.

It is possible to rplaca the result of rassoc, provided that it is not nil, in order to "update" *alist*.

The expressions

```
(rassoc item list :test fn)
```

and

```
(find item list :test fn :key #'cdr)
```

are equivalent in meaning, except when the *item* is nil and nil appears in place of a pair in the *alist*. See the function assoc.

***Function RATIONALP*****Syntax:**

**rationalp** *object* => *generalized-boolean*

**Arguments and Values:**

*object*---an *object*.

*generalized-boolean*---a *generalized boolean*.

**Description:**

Returns *true* if *object* is of *type rational*; otherwise, returns *false*.

**Examples:**

```
(rationalp 12) => true
(rationalp 6/5) => true
(rationalp 1.212) => false
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**rational**

**Notes:**

```
(rationalp object) == (typep object 'rational)
```

***Function RATIONAL, RATIONALIZE*****Syntax:**

**rational** *number* => *rational*

**rationalize** *number* => *rational*

**Arguments and Values:**

*number*---a *real*.

*rational*---a *rational*.

**Description:**

**rational** and **rationalize** convert *reals* to *rationals*.

If *number* is already *rational*, it is returned.

If *number* is a *float*, **rational** returns a *rational* that is mathematically equal in value to the *float*. **rationalize** returns a *rational* that approximates the *float* to the accuracy of the underlying floating-point representation.

**rational** assumes that the *float* is completely accurate.

**rationalize** assumes that the *float* is accurate only to the precision of the floating-point representation.

**Examples:**

```
(rational 0) => 0
(rationalize -11/100) => -11/100
(rational .1) => 13421773/134217728 ;implementation-dependent
(rationalize .1) => 1/10
```

**Side Effects:** None.

**Affected By:**

The *implementation*.

**Exceptional Situations:**

Should signal an error of *type type-error* if *number* is not a *real*. Might signal *arithmetic-error*.

**See Also:** None.

**Notes:**

It is always the case that

```
(float (rational x) x) == x
```

and

```
(float (rationalize x) x) == x
```

**Function READ-BYTE****Syntax:**

**read-byte** *stream* &*optional eof-error-p* *eof-value* => *byte*

**Arguments and Values:**

*stream*—a *binary input stream*.

*eof-error-p*---a generalized boolean. The default is true.

*eof-value*---an object. The default is nil.

*byte*---an integer, or the *eof-value*.

#### Description:

**read-byte** reads and returns one byte from *stream*.

If an end of file[2] occurs and *eof-error-p* is false, the *eof-value* is returned.

#### 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
```

#### Side Effects:

Modifies *stream*.

**Affected By:** None.

#### Exceptional Situations:

Should signal an error of type type-error if *stream* is not a stream.

Should signal an error of type error if *stream* is not a binary input stream.

If there are no bytes remaining in the *stream* and *eof-error-p* is true, an error of type end-of-file is signaled.

#### See Also:

**read-char**, **read-sequence**, **write-byte**

**Notes:** None.

## Function READ-CHAR-NO-HANG

#### Syntax:

**read-char-no-hang** &optional *input-stream* *eof-error-p* *eof-value* *recursive-p* => *char*

#### Arguments and Values:

*input-stream* --- an input stream designator. The default is standard input.

*eof-error-p*---a generalized boolean. The default is true.

*eof-value*---an object. The default is nil.

*recursive-p*---a generalized boolean. The default is false.

*char*---a character or nil or the *eof-value*.

#### Description:

**read-char-no-hang** returns a character from *input-stream* if such a character is available. If no character is available, **read-char-no-hang** returns nil.

If *recursive-p* is true, this call is expected to be embedded in a higher-level call to **read** or a similar function used by the Lisp reader.

If an end of file[2] occurs and *eof-error-p* is false, *eof-value* is returned.

#### Examples:

```
; ; This code assumes an implementation in which a newline is not
; ; required to terminate input from the console.
(defun test-it ()
  (unread-char (read-char))
  (list (read-char-no-hang)
        (read-char-no-hang)
        (read-char-no-hang)))
=> TEST-IT
; ; Implementation A, where a Newline is not required to terminate
; ; interactive input on the console.
(test-it)
>> a
=> (#\a NIL NIL)
; ; Implementation B, where a Newline is required to terminate
; ; interactive input on the console, and where that Newline remains
; ; on the input stream.
(test-it)
>> a<NEWLINE>
=> (#\a #\Newline NIL)
```

#### Affected By:

\*standard-input\*, \*terminal-io\*.

#### Exceptional Situations:

If an end of file[2] occurs when *eof-error-p* is true, an error of type end-of-file is signaled .

#### See Also:

listen

#### Notes:

**read-char-no-hang** is exactly like **read-char**, except that if it would be necessary to wait in order to get a character (as from a keyboard), **nil** is immediately returned without waiting.

## Function READ-CHAR

### Syntax:

**read-char** &optional *input-stream* *eof-error-p* *eof-value* *recursive-p* => *char*

### Arguments and Values:

*input-stream*—an *input stream designator*. The default is *standard input*.

*eof-error-p*—a *generalized boolean*. The default is *true*.

*eof-value*—an *object*. The default is **nil**.

*recursive-p*—a *generalized boolean*. The default is *false*.

*char*—a *character* or the *eof-value*.

### Description:

**read-char** returns the next *character* from *input-stream*.

When *input-stream* is an *echo stream*, the character is echoed on *input-stream* the first time the character is seen. Characters that are not echoed by **read-char** are those that were put there by **unread-char** and hence are assumed to have been echoed already by a previous call to **read-char**.

If *recursive-p* is *true*, this call is expected to be embedded in a higher-level call to **read** or a similar *function* used by the *Lisp reader*.

If an *end of file*[2] occurs and *eof-error-p* is *false*, *eof-value* is returned.

### Examples:

```
(with-input-from-string (is "0123")
  (do ((c (read-char is) (read-char is nil 'the-end)))
       ((not (characterp c)))
    (format t "~S ~A" c))
>> #\0 #\1 #\2 #\3
=> NIL
```

### Affected By:

\***standard-input**\*, \***terminal-io**\*

### Exceptional Situations:

If an *end of file*[2] occurs before a character can be read, and *eof-error-p* is *true*, an error of *type end-of-file* is signaled.

**See Also:**

[read-byte](#), [read-sequence](#), [write-char](#), [read](#)

**Notes:**

The corresponding output function is [write-char](#).

**Function READ-DELIMITED-LIST****Syntax:**

**read-delimited-list** *char* &*optional input-stream recursive-p => list*

**Arguments and Values:**

*char*—a [character](#).

*input-stream*—an [input stream designator](#). The default is [standard input](#).

*recursive-p*—a [generalized boolean](#). The default is [false](#).

*list*—a [list](#) of the [objects](#) read.

**Description:**

**read-delimited-list** reads [objects](#) from *input-stream* until the next character after an [object](#)'s representation (ignoring [whitespace\[2\]](#) characters and comments) is *char*.

**read-delimited-list** looks ahead at each step for the next non-[whitespace\[2\]](#) [character](#) and peeked at it as if with [peek-char](#). If it is *char*, then the [character](#) is consumed and the [list](#) of [objects](#) is returned. If it is a [constituent](#) or [escape character](#), then [read](#) is used to read an [object](#), which is added to the end of the [list](#). If it is a [macro character](#), its [reader macro function](#) is called; if the function returns a [value](#), that [value](#) is added to the [list](#). The peek-ahead process is then repeated.

If *recursive-p* is [true](#), this call is expected to be embedded in a higher-level call to [read](#) or a similar function.

It is an error to reach end-of-file during the operation of **read-delimited-list**.

The consequences are undefined if *char* has a [syntax type](#) of [whitespace\[2\]](#) in the [current readable](#).

**Examples:**

```
(read-delimited-list #\]) 1 2 3 4 5 6 ]
=> (1 2 3 4 5 6)
```

Suppose you wanted  $\#\{a\ b\ c\ \dots\ z\}$  to read as a list of all pairs of the elements *a*, *b*, *c*, ..., *z*, for example.

```
#\{p\ q\ z\ a\}  reads as ((p\ q)\ (p\ z)\ (p\ a)\ (q\ z)\ (q\ a)\ (z\ a))
```

## CLHS: Declaration DYNAMIC-EXTENT

This can be done by specifying a macro-character definition for #{} that does two things: reads in all the items up to the }, and constructs the pairs. read-delimited-list performs the first task.

```
(defun |#{-reader| (stream char arg)
  (declare (ignore char arg))
  (mapcon #'(lambda (x)
    (mapcar #'(lambda (y) (list (car x) y)) (cdr x)))
    (read-delimited-list #\} stream t))) => |#{-reader|
(set-dispatch-macro-character #\# #\{ #'|#{-reader|) => T
(set-macro-character #\} (get-macro-character #\) nil))
```

Note that true is supplied for the *recursive-p* argument.

It is necessary here to give a definition to the character } as well to prevent it from being a constituent. If the line

```
(set-macro-character #\} (get-macro-character #\) nil))
```

shown above were not included, then the } in

```
#{ p q z a}
```

would be considered a constituent character, part of the symbol named a}. This could be corrected by putting a space before the }, but it is better to call set-macro-character.

Giving } the same definition as the standard definition of the character ) has the twin benefit of making it terminate tokens for use with read-delimited-list and also making it invalid for use in any other context. Attempting to read a stray } will signal an error.

### Affected By:

\*standard-input\*, \*readtable\*, \*terminal-io\*.

**Exceptional Situations:** None.

### See Also:

read, peek-char, read-char, unread-char.

### Notes:

read-delimited-list is intended for use in implementing reader macros. Usually it is desirable for *char* to be a terminating macro character so that it can be used to delimit tokens; however, read-delimited-list makes no attempt to alter the syntax specified for *char* by the current readtable. The caller must make any necessary changes to the readtable syntax explicitly.

## Function READ-FROM-STRING

### Syntax:

**read-from-string** *string* &*optional eof-error-p* *eof-value* &*key start end preserve-whitespace*

=> *object, position*

#### Arguments and Values:

*string*---a string.

*eof-error-p*---a generalized boolean. The default is true.

*eof-value*---an object. The default is nil.

*start, end*---bounding index designators of *string*. The defaults for *start* and *end* are 0 and nil, respectively.

*preserve-whitespace*---a generalized boolean. The default is false.

*object*---an object (parsed by the Lisp reader) or the *eof-value*.

*position*---an integer greater than or equal to zero, and less than or equal to one more than the length of the *string*.

#### Description:

Parses the printed representation of an object from the subsequence of *string bounded* by *start* and *end*, as if read had been called on an input stream containing those same characters.

If *preserve-whitespace* is true, the operation will preserve whitespace[2] as read-preserving-whitespace would do.

If an object is successfully parsed, the primary value, *object*, is the object that was parsed. If *eof-error-p* is false and if the end of the *substring* is reached, *eof-value* is returned.

The secondary value, *position*, is the index of the first character in the bounded string that was not read. The *position* may depend upon the value of *preserve-whitespace*. If the entire *string* was read, the *position* returned is either the length of the *string* or one greater than the length of the *string*.

#### Examples:

```
(read-from-string " 1 3 5" t nil :start 2) => 3, 5
(read-from-string "(a b c)" ) => (A B C), 7
```

**Side Effects:** None.

**Affected By:** None.

#### Exceptional Situations:

If the end of the supplied substring occurs before an object can be read, an error is signaled if *eof-error-p* is true. An error is signaled if the end of the *substring* occurs in the middle of an incomplete object.

#### See Also:

read, read-preserving-whitespace

**Notes:**

The reason that *position* is allowed to be beyond the *length* of the *string* is to permit (but not require) the *implementation* to work by simulating the effect of a trailing delimiter at the end of the *bounded string*. When *preserve-whitespace* is *true*, the *position* might count the simulated delimiter.

**Function READ-LINE****Syntax:**

```
read-line &optional input-stream eof-error-p eof-value recursive-p  
=> line missing-newline-p
```

**Arguments and Values:**

*input-stream*—an *input stream designator*. The default is *standard input*.

*eof-error-p*—a *generalized boolean*. The default is *true*.

*eof-value*—an *object*. The default is *nil*.

*recursive-p*—a *generalized boolean*. The default is *false*.

*line*—a *string* or the *eof-value*.

*missing-newline-p*—a *generalized boolean*.

**Description:**

Reads from *input-stream* a line of text that is terminated by a *newline* or *end of file*.

If *recursive-p* is *true*, this call is expected to be embedded in a higher-level call to **read** or a similar *function* used by the *Lisp reader*.

The *primary value*, *line*, is the line that is read, represented as a *string* (without the trailing *newline*, if any). If *eof-error-p* is *false* and the *end of file* for *input-stream* is reached before any *characters* are read, *eof-value* is returned as the *line*.

The *secondary value*, *missing-newline-p*, is a *generalized boolean* that is *false* if the *line* was terminated by a *newline*, or *true* if the *line* was terminated by the *end of file* for *input-stream* (or if the *line* is the *eof-value*).

**Examples:**

```
(setq a "line 1  
line2")  
=> "line 1  
line2"  
(read-line (setq input-stream (make-string-input-stream a)))  
=> "line 1", false  
(read-line input-stream)  
=> "line2", true
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(read-line input-stream nil nil)
=> NIL, true
```

**Side Effects:** None.

**Affected By:**

\*standard-input\*, \*terminal-io\*.

**Exceptional Situations:**

If an end-of-file[2] occurs before any characters are read in the line, an error is signaled if eof-error-p is true.

**See Also:**

read

**Notes:**

The corresponding output function is write-line.

## Function READ, READ-PRESERVING-WHITESPACE

**Syntax:**

**read** &*optional input-stream eof-error-p eof-value recursive-p => object*

**read-preserving-whitespace** &*optional input-stream eof-error-p eof-value recursive-p => object*

**Arguments and Values:**

*input-stream*—an input stream designator.

*eof-error-p*—a generalized boolean. The default is true.

*eof-value*—an object. The default is nil.

*recursive-p*—a generalized boolean. The default is false.

*object*—an object (parsed by the Lisp reader) or the *eof-value*.

**Description:**

**read** parses the printed representation of an object from *input-stream* and builds such an object.

**read-preserving-whitespace** is like **read** but preserves any whitespace[2] character that delimits the printed representation of the object. **read-preserving-whitespace** is exactly like **read** when the *recursive-p argument* to **read-preserving-whitespace** is true.

## CLHS: Declaration DYNAMIC-EXTENT

When **\*read-suppress\*** is *false*, **read** throws away the delimiting *character* required by certain printed representations if it is a *whitespace*[2] *character*; but **read** preserves the character (using **unread-char**) if it is syntactically meaningful, because it could be the start of the next expression.

If a file ends in a *symbol* or a *number* immediately followed by an *end of file*[1], **read** reads the *symbol* or *number* successfully; when called again, it sees the *end of file*[1] and only then acts according to *eof-error-p*. If a file contains ignorable text at the end, such as blank lines and comments, **read** does not consider it to end in the middle of an *object*.

If *recursive-p* is *true*, the call to **read** is expected to be made from within some function that itself has been called from **read** or from a similar input function, rather than from the top level.

Both functions return the *object* read from *input-stream*. *Eof-value* is returned if *eof-error-p* is *false* and end of file is reached before the beginning of an *object*.

### 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 t nil t) (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
```

As an example, consider this *reader macro* definition:

```
(defun slash-reader (stream char)
  (declare (ignore char))
  `(path . ,(loop for dir = (read-preserving-whitespace stream t nil t)
                 then (progn (read-char stream t nil t)
                               (read-preserving-whitespace stream t nil t))
                 collect dir
                 while (eql (peek-char nil stream nil nil t) #\/))))
  (set-macro-character #\/ #'slash-reader))
```

Consider now calling **read** on this expression:

```
(zyedh /usr/games/zork /usr/games/boggle)
```

The `/` macro reads objects separated by more `/` characters; thus `/usr/games/zork` is intended to read as `(path usr games zork)`. The entire example expression should therefore be read as

```
(zyedh (path usr games zork) (path usr games boggle))
```

However, if **read** had been used instead of **read-preserving-whitespace**, then after the reading of the symbol `zork`, the following space would be discarded; the next call to **peek-char** would see the following `/`, and the loop would continue, producing this interpretation:

```
(zyedh (path usr games zork usr games boggle))
```

## CLHS: Declaration DYNAMIC-EXTENT

There are times when whitespace[2] should be discarded. If a command interpreter takes single-character commands, but occasionally reads an object then if the whitespace[2] after a symbol is not discarded it might be interpreted as a command some time later after the symbol had been read.

### Affected By:

\*standard-input\*, \*terminal-io\*, \*readtable\*, \*read-default-float-format\*, \*read-base\*,  
\*read-suppress\*, \*package\*, \*read-eval\*.

### Exceptional Situations:

read signals an error of type end-of-file, regardless of eof-error-p, if the file ends in the middle of an object representation. For example, if a file does not contain enough right parentheses to balance the left parentheses in it, read signals an error. This is detected when read or read-preserving-whitespace is called with recursive-p and eof-error-p non-nil, and end-of-file is reached before the beginning of an object.

If eof-error-p is true, an error of type end-of-file is signaled at the end of file.

### See Also:

peek-char, read-char, unread-char, read-from-string, read-delimited-list, parse-integer, Section 2 (Syntax), Section 23.1 (Reader Concepts)

**Notes:** None.

## Function READ-SEQUENCE

### Syntax:

**read-sequence** *sequence stream &key start end => position*

*sequence*—a sequence.

*stream*—an input stream.

*start*, *end*—bounding index designators of *sequence*. The defaults for *start* and *end* are 0 and nil, respectively.

*position*—an integer greater than or equal to zero, and less than or equal to the length of the *sequence*.

### Description:

Destructively modifies *sequence* by replacing the elements of *sequence bounded* by *start* and *end* with elements read from *stream*.

*Sequence* is destructively modified by copying successive elements into it from *stream*. If the end of file for *stream* is reached before copying all elements of the subsequence, then the extra elements near the end of *sequence* are not updated.

*Position* is the index of the first element of *sequence* that was not updated, which might be less than *end* because the end of file was reached.

**Examples:**

```
(defvar *data* (make-array 15 :initial-element nil))
(values (read-sequence *data* (make-string-input-stream "test string")) *data*)
=> 11, #(#\t#\e#\s#\t#\Space#\s#\t#\r#\i#\n#\g NIL NIL NIL NIL)
```

**Side Effects:**

Modifies *stream* and *sequence*.

**Affected By:** None.

**Exceptional Situations:**

Should be prepared to signal an error of type type-error if *sequence* is not a proper sequence. Should signal an error of type type-error if *start* is not a non-negative integer. Should signal an error of type type-error if *end* is not a non-negative integer or nil.

Might signal an error of type type-error if an element read from the *stream* is not a member of the element type of the *sequence*.

**See Also:****Notes:**

read-sequence is identical in effect to iterating over the indicated subsequence and reading one element at a time from *stream* and storing it into *sequence*, but may be more efficient than the equivalent loop. An efficient implementation is more likely to exist for the case where the *sequence* is a vector with the same element type as the *stream*.

**Function READTABLEP****Syntax:**

**readtablep** *object* => *generalized-boolean*

**Arguments and Values:**

*object*---an object.

*generalized-boolean*---a generalized boolean.

**Description:**

Returns true if *object* is of type readable; otherwise, returns false.

**Examples:**

```
(readtablep *readtable*) => true
(readtablep (copy-readtable)) => true
(readtablep '*readtable*) => false
```

**Side Effects:** None.**Affected By:** None.**Exceptional Situations:** None.**See Also:** None.**Notes:**

```
(readtablep object) == (typep object 'readtable)
```

**Accessor READTABLE-CASE****Syntax:****readtable-case** *readtable* => *mode*(setf (**readtable-case** *readtable*) *mode*)**Arguments and Values:***readtable*—a readtable.*mode*—a case sensitivity mode.**Description:**

Accesses the readtable case of *readtable*, which affects the way in which the Lisp Reader reads symbols and the way in which the Lisp Printer writes symbols.

**Examples:**

See [Section 23.1.2.1 \(Examples of Effect of Readtable Case on the Lisp Reader\)](#) and [Section 22.1.3.3.2.1 \(Examples of Effect of Readtable Case on the Lisp Printer\)](#).

**Affected By:** None.**Exceptional Situations:**

Should signal an error of type type-error if *readtable* is not a readtable. Should signal an error of type type-error if *mode* is not a case sensitivity mode.

**See Also:**

[\*\*\\*readtable\\*\*\*](#), [\*\*\\*print-escape\\*\*\*](#), [Section 2.2 \(Reader Algorithm\)](#), [Section 23.1.2 \(Effect of Readtable Case on the Lisp Reader\)](#), [Section 22.1.3.3.2 \(Effect of Readtable Case on the Lisp Printer\)](#)

**Notes:**

**copy-readtable** copies the readtable case of the *readtable*.

***Function REALP*****Syntax:**

**realp** *object* => *generalized-boolean*

**Arguments and Values:**

*object*---an *object*.

*generalized-boolean*---a *generalized boolean*.

**Description:**

Returns *true* if *object* is of *type real*; otherwise, returns *false*.

**Examples:**

```
(realp 12) => true
(realp #c(5/3 7.2)) => false
(realp nil) => false
(realp (cons 1 2)) => false
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:** None.

**Notes:**

```
(realp object) == (typep object 'real)
```

***Function REALPART, IMAGPART*****Syntax:**

**realpart** *number* => *real*

**imagpart** *number* => *real*

**Arguments and Values:**

*number*---a *number*.

*real*---a *real*.

**Description:**

## CLHS: Declaration DYNAMIC-EXTENT

**realpart** and **imagpart** return the real and imaginary parts of *number* respectively. If *number* is real, then **realpart** returns *number* and **imagpart** returns (\* 0 *number*), which has the effect that the imaginary part of a rational is 0 and that of a float is a floating-point zero of the same format.

### 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
```

**Side Effects:** None.

**Affected By:** None.

### Exceptional Situations:

Should signal an error of type type-error if *number* is not a number.

### See Also:

**complex**

**Notes:** None.

## Function REDUCE

### Syntax:

**reduce** *function sequence &key key from-end start end initial-value => result*

### Arguments and Values:

*function*---a designator for a function that might be called with either zero or two arguments.

*sequence*---a proper sequence.

*key*---a designator for a function of one argument, or nil.

*from-end*---a generalized boolean. The default is false.

*start, end*---bounding index designators of *sequence*. The defaults for *start* and *end* are 0 and nil, respectively.

*initial-value*---an object.

*result*---an object.

### Description:

**reduce** uses a binary operation, *function*, to combine the elements of *sequence bounded* by *start* and *end*.

## CLHS: Declaration DYNAMIC-EXTENT

The *function* must accept as *arguments* two *elements* of *sequence* or the results from combining those *elements*. The *function* must also be able to accept no arguments.

If *key* is supplied, it is used to extract the values to reduce. The *key* function is applied exactly once to each element of *sequence* in the order implied by the reduction order but not to the value of *initial-value*, if supplied. The *key* function typically returns part of the *element* of *sequence*. If *key* is not supplied or is *nil*, the *sequence element* itself is used.

The reduction is left-associative, unless *from-end* is *true* in which case it is right-associative.

If *initial-value* is supplied, it is logically placed before the subsequence (or after it if *from-end* is *true*) and included in the reduction operation.

In the normal case, the result of *reduce* is the combined result of *function*'s being applied to successive pairs of *elements* of *sequence*. If the subsequence contains exactly one *element* and no *initial-value* is given, then that *element* is returned and *function* is not called. If the subsequence is empty and an *initial-value* is given, then the *initial-value* is returned and *function* is not called. If the subsequence is empty and no *initial-value* is given, then the *function* is called with zero arguments, and *reduce* returns whatever *function* does. This is the only case where the *function* is called with other than two arguments.

### Examples:

```
(reduce #'* '(1 2 3 4 5)) => 120
(reduce #'append '((1) (2)) :initial-value '(i n i t)) => (I N I T 1 2)
(reduce #'append '((1) (2)) :from-end t
            :initial-value '(i n i t)) => (1 2 I N I T)
(reduce #'- '(1 2 3 4)) == (- (- (- 1 2) 3) 4) => -8
(reduce #'- '(1 2 3 4) :from-end t) ;Alternating sum.
== (- 1 (- 2 (- 3 4))) => -2
(reduce #'+ '()) => 0
(reduce #'+ '(3)) => 3
(reduce #'+ '(foo)) => FOO
(reduce #'list '(1 2 3 4)) => (((1 2) 3) 4)
(reduce #'list '(1 2 3 4) :from-end t) => (1 (2 (3 4)))
(reduce #'list '(1 2 3 4) :initial-value 'foo) => (((foo 1) 2) 3) 4)
(reduce #'list '(1 2 3 4)
         :from-end t :initial-value 'foo) => (1 (2 (3 (4 foo))))
```

**Side Effects:** None.

**Affected By:** None.

### Exceptional Situations:

Should be prepared to signal an error of *type type-error* if *sequence* is not a *proper sequence*.

### See Also:

[Section 3.6 \(Traversal Rules and Side Effects\)](#)

**Notes:** None.

## **Standard Generic Function REINITIALIZE-INSTANCE**

### Syntax:

**reinitialize-instance** *instance* &*rest initargs* &*key allow-other-keys* => *instance*

### Method Signatures:

**reinitialize-instance** (*instance standard-object*) &*rest initargs*

### Arguments and Values:

*instance*—an *object*.

*initargs*—an *initialization argument list*.

### Description:

The *generic function reinitialize-instance* can be used to change the values of *local slots* of an *instance* according to *initargs*. This *generic function* can be called by users.

The system-supplied primary *method* for **reinitialize-instance** checks the validity of *initargs* and signals an error if an *initarg* is supplied that is not declared as valid. The *method* then calls the generic function **shared-initialize** with the following arguments: the *instance*, **nil** (which means no *slots* should be initialized according to their initforms), and the *initargs* it received.

**Examples:** None.

### Side Effects:

The *generic function reinitialize-instance* changes the values of *local slots*.

**Affected By:** None.

### Exceptional Situations:

The system-supplied primary *method* for **reinitialize-instance** signals an error if an *initarg* is supplied that is not declared as valid.

### See Also:

[initialize-instance](#), [shared-initialize](#), [update-instance-for-redefined-class](#),  
[update-instance-for-different-class](#), [slot-boundp](#), [slot-makunbound](#), [Section 7.3 \(Reinitializing an Instance\)](#), [Section 7.1.4 \(Rules for Initialization Arguments\)](#), [Section 7.1.2 \(Declaring the Validity of Initialization Arguments\)](#)

### Notes:

*Initargs* are declared as valid by using the :*initarg* option to [defclass](#), or by defining *methods* for **reinitialize-instance** or **shared-initialize**. The keyword name of each keyword parameter specifier in the *lambda list* of any *method* defined on **reinitialize-instance** or **shared-initialize** is declared as a valid

initialization argument name for all classes for which that method is applicable.

## **Function REMHASH**

### **Syntax:**

**remhash** *key hash-table => generalized-boolean*

### **Arguments and Values:**

*key*—an object.

*hash-table*—a hash table.

*generalized-boolean*—a generalized boolean.

### **Description:**

Removes the entry for *key* in *hash-table*, if any. Returns true if there was such an entry, or false otherwise.

### **Examples:**

```
(setq table (make-hash-table)) => #<HASH-TABLE EQL 0/120 32115666>
(setf (gethash 100 table) "C") => "C"
(gethash 100 table) => "C", true
(remhash 100 table) => true
(gethash 100 table) => NIL, false
(remhash 100 table) => false
```

### **Side Effects:**

The *hash-table* is modified.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:** None.

**Notes:** None.

## **Function REMPROP**

### **Syntax:**

**remprop** *symbol indicator => generalized-boolean*

### **Arguments and Values:**

*symbol*—a symbol.

*indicator*—an object.

*generalized-boolean*—a generalized boolean.

### Description:

**remprop** removes from the property list[2] of symbol a property[1] with a property indicator identical to *indicator*. If there are multiple properties[1] with the identical key, **remprop** only removes the first such property. **remprop** returns false if no such property was found, or true if a property was found.

The property indicator and the corresponding property value are removed in an undefined order by destructively splicing the property list. The permissible side-effects correspond to those permitted for **remf**, such that:

```
(remprop x y) == (remf (symbol-plist x) y)
```

### Examples:

```
(setq test (make-symbol "PSEUDO-PI")) => #:PSEUDO-PI
(symbol-plist test) => ()
(setf (get test 'constant) t) => T
(setf (get test 'approximation) 3.14) => 3.14
(setf (get test 'error-range) 'noticeable) => NOTICEABLE
(symbol-plist test)
=> (ERROR-RANGE NOTICEABLE APPROXIMATION 3.14 CONSTANT T)
(setf (get test 'approximation) nil) => NIL
(symbol-plist test)
=> (ERROR-RANGE NOTICEABLE APPROXIMATION NIL CONSTANT T)
(get test 'approximation) => NIL
(remprop test 'approximation) => true
(get test 'approximation) => NIL
(symbol-plist test)
=> (ERROR-RANGE NOTICEABLE CONSTANT T)
(remprop test 'approximation) => NIL
(symbol-plist test)
=> (ERROR-RANGE NOTICEABLE CONSTANT T)
(remprop test 'error-range) => true
(setf (get test 'approximation) 3) => 3
(symbol-plist test)
=> (APPROXIMATION 3 CONSTANT T)
```

### Side Effects:

The property list of symbol is modified.

**Affected By:** None.

### Exceptional Situations:

Should signal an error of type type-error if symbol is not a symbol.

### See Also:

remf, symbol-plist

**Notes:**

Numbers and characters are not recommended for use as *indicators* in portable code since remprop tests with eq rather than eql, and consequently the effect of using such *indicators* is implementation-dependent. Of course, if you've gotten as far as needing to remove such a property, you don't have much choice---the time to have been thinking about this was when you used setf of get to establish the property.

**Function REPLACE****Syntax:**

```
replace sequence-1 sequence-2 &key start1 end1 start2 end2 => sequence-1
```

**Arguments and Values:**

*sequence-1*---a sequence.

*sequence-2*---a sequence.

*start1*, *end1*---bounding index designators of *sequence-1*. The defaults for *start1* and *end1* are 0 and nil, respectively.

*start2*, *end2*---bounding index designators of *sequence-2*. The defaults for *start2* and *end2* are 0 and nil, respectively.

**Description:**

Destructively modifies *sequence-1* by replacing the elements of subsequence-1 bounded by *start1* and *end1* with the elements of subsequence-2 bounded by *start2* and *end2*.

*Sequence-1* is destructively modified by copying successive elements into it from *sequence-2*. Elements of the subsequence of *sequence-2* bounded by *start2* and *end2* are copied into the subsequence of *sequence-1* bounded by *start1* and *end1*. If these subsequences are not of the same length, then the shorter length determines how many elements are copied; the extra elements near the end of the longer subsequence are not involved in the operation. The number of elements copied can be expressed as:

```
(min (- end1 start1) (- end2 start2))
```

If *sequence-1* and *sequence-2* are the same object and the region being modified overlaps the region being copied from, then it is as if the entire source region were copied to another place and only then copied back into the target region. However, if *sequence-1* and *sequence-2* are not the same, but the region being modified overlaps the region being copied from (perhaps because of shared list structure or displaced arrays), then after the replace operation the subsequence of *sequence-1* being modified will have unpredictable contents. It is an error if the elements of *sequence-2* are not of a type that can be stored into *sequence-1*.

**Examples:**

```
(replace "abcdefghijkl" "0123456789" :start1 4 :end1 7 :start2 4)
=> "abcd456hij"
(setq lst "012345678") => "012345678"
(replace lst lst :start1 2 :start2 0) => "010123456"
lst => "010123456"
```

**Side Effects:**

The *sequence-l* is modified.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

fill

**Notes:** None.

**Accessor REST****Syntax:**

**rest** *list* => *tail*

(**setf** (**rest** *list*) *new-tail*)

**Arguments and Values:**

*list*—a list, which might be a dotted list or a circular list.

*tail*—an object.

**Description:**

**rest** performs the same operation as **cdr**, but mnemonically complements **first**. Specifically,

```
(rest list) == (cdr list)
(setf (rest list) new-tail) == (setf (cdr list) new-tail)
```

**Examples:**

```
(rest '(1 2)) => (2)
(rest '(1 . 2)) => 2
(rest '(1)) => NIL
(setq *cons* '(1 . 2)) => (1 . 2)
(setf (rest *cons*) "two") => "two"
*cons* => (1 . "two")
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**cdr, nthcdr****Notes:**

rest is often preferred stylistically over cdr when the argument is to be subjectively viewed as a list rather than as a cons.

**Function REVAPPEND, NRECONC****Syntax:**

**revappend** *list tail* => *result-list*

**nreconc** *list tail* => *result-list*

**Arguments and Values:**

*list*—a proper list.

*tail*—an object.

*result-list*—an object.

**Description:**

**revappend** constructs a copy[2] of *list*, but with the elements in reverse order. It then appends (as if by nconc) the *tail* to that reversed list and returns the result.

**nreconc** reverses the order of elements in *list* (as if by nreverse). It then appends (as if by nconc) the *tail* to that reversed list and returns the result.

The resulting *list* shares list structure with *tail*.

**Examples:**

```
(let ((list-1 (list 1 2 3))
      (list-2 (list 'a 'b 'c)))
  (print (revappend list-1 list-2)))
  (print (equal list-1 '(1 2 3)))
  (print (equal list-2 '(a b c))))
>> (3 2 1 A B C)
>> T
>> T
=> T

  (revappend '(1 2 3) '()) => (3 2 1)
  (revappend '(1 2 3) '(a . b)) => (3 2 1 A . B)
  (revappend '() '(a b c)) => (A B C)
  (revappend '(1 2 3) 'a) => (3 2 1 . A)
  (revappend '() 'a) => A      ;degenerate case

  (let ((list-1 '(1 2 3))
        (list-2 '(a b c)))
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(print (nreconc list-1 list-2))
(print (equal list-1 '(1 2 3)))
(print (equal list-2 '(a b c)))
>> (3 2 1 A B C)
>> NIL
>> T
=> T
```

### Side Effects:

**revappend** does not modify either of its *arguments*. **nreconc** is permitted to modify *list* but not *tail*.

Although it might be implemented differently, **nreconc** is constrained to have side-effect behavior equivalent to:

```
(nconc (nreverse list) tail)
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**reverse, nreverse, nconc**

**Notes:**

The following functional equivalences are true, although good *implementations* will typically use a faster algorithm for achieving the same effect:

```
(revappend list tail) == (nconc (reverse list) tail)
(nreconc list tail) == (nconc (nreverse list) tail)
```

## Function REVERSE, NREVERSE

**Syntax:**

**reverse** *sequence* => *reversed-sequence*

**nreverse** *sequence* => *reversed-sequence*

**Arguments and Values:**

*sequence*—a *proper sequence*.

*reversed-sequence*—a *sequence*.

**Description:**

**reverse** and **nreverse** return a new *sequence* of the same kind as *sequence*, containing the same *elements*, but

in reverse order.

**reverse** and **nreverse** differ in that **reverse** always creates and returns a new *sequence*, whereas **nreverse** might modify and return the given *sequence*. **reverse** never modifies the given *sequence*.

For **reverse**, if *sequence* is a *vector*, the result is a *fresh simple array* of *rank* one that has the same *actual array element type* as *sequence*. If *sequence* is a *list*, the result is a *fresh list*.

For **nreverse**, if *sequence* is a *vector*, the result is a *vector* that has the same *actual array element type* as *sequence*. If *sequence* is a *list*, the result is a *list*.

For **nreverse**, *sequence* might be destroyed and re-used to produce the result. The result might or might not be *identical* to *sequence*. Specifically, when *sequence* is a *list*, **nreverse** is permitted to **setf** any part, **car** or **cdr**, of any **cons** that is part of the *list structure* of *sequence*. When *sequence* is a *vector*, **nreverse** is permitted to re-order the elements of *sequence* in order to produce the resulting *vector*.

### Examples:

```
(setq str "abc") => "abc"
(reverse str) => "cba"
str => "abc"
(setq str (copy-seq str)) => "abc"
(nreverse str) => "cba"
str => implementation-dependent
(setq l (list 1 2 3)) => (1 2 3)
(nreverse l) => (3 2 1)
l => implementation-dependent
```

### Side Effects:

**nreverse** might either create a new *sequence*, modify the argument *sequence*, or both. (**reverse** does not modify *sequence*.)

**Affected By:** None.

### Exceptional Situations:

Should be prepared to signal an error of *type type-error* if *sequence* is not a *proper sequence*.

**See Also:** None.

**Notes:** None.

## Function REMOVE-DUPLICATES, DELETE-DUPLICATES

### Syntax:

**remove-duplicates** *sequence* &*key* *from-end test test-not start end key*

=> *result-sequence*

**delete-duplicates** *sequence &key from-end test test-not start end key*

=> *result-sequence*

#### Arguments and Values:

*sequence*—a *proper sequence*.

*from-end*—a *generalized boolean*. The default is *false*.

*test*—a *designator* for a *function* of two *arguments* that returns a *generalized boolean*.

*test-not*—a *designator* for a *function* of two *arguments* that returns a *generalized boolean*.

*start, end*—*bounding index designators* of *sequence*. The defaults for *start* and *end* are 0 and *nil*, respectively.

*key*—a *designator* for a *function* of one argument, or *nil*.

*result-sequence*—a *sequence*.

#### Description:

**remove-duplicates** returns a modified copy of *sequence* from which any element that matches another element occurring in *sequence* has been removed.

If *sequence* is a *vector*, the result is a *vector* that has the same *actual array element type* as *sequence*. If *sequence* is a *list*, the result is a *list*.

**delete-duplicates** is like **remove-duplicates**, but **delete-duplicates** may modify *sequence*.

The elements of *sequence* are compared *pairwise*, and if any two match, then the one occurring earlier in *sequence* is discarded, unless *from-end* is *true*, in which case the one later in *sequence* is discarded.

**remove-duplicates** and **delete-duplicates** return a *sequence* of the same *type* as *sequence* with enough elements removed so that no two of the remaining elements match. The order of the elements remaining in the result is the same as the order in which they appear in *sequence*.

**remove-duplicates** returns a *sequence* that may share with *sequence* or may be *identical* to *sequence* if no elements need to be removed.

**delete-duplicates**, when *sequence* is a *list*, is permitted to *setf* any part, *car* or *cdr*, of the top-level list structure in that *sequence*. When *sequence* is a *vector*, **delete-duplicates** is permitted to change the dimensions of the *vector* and to slide its elements into new positions without permuting them to produce the resulting *vector*.

#### Examples:

```
(remove-duplicates "aBcDAbCd" :test #'char-equal :from-end t) => "aBcD"
(remove-duplicates '(a b c b d d e)) => (A C B D E)
(remove-duplicates '(a b c b d d e) :from-end t) => (A B C D E)
(remove-duplicates '((foo #\a) (bar #\%) (baz #\A)))
```

## CLHS: Declaration DYNAMIC-EXTENT

```
:test #'char-equal :key #'cadr => ((BAR #\%) (BAZ #\A))
(remove-duplicates '((foo #\a) (bar #\%) (baz #\A)))
  :test #'char-equal :key #'cadr :from-end t) => ((FOO #\a) (BAR #\%))
(setq tester (list 0 1 2 3 4 5 6))
(delete-duplicates tester :key #'oddp :start 1 :end 6) => (0 4 5 6)
```

### Side Effects:

**delete-duplicates** might destructively modify *sequence*.

**Affected By:** None.

### Exceptional Situations:

Should signal an error of **type type-error** if *sequence* is not a *proper sequence*.

### See Also:

[Section 3.2.1 \(Compiler Terminology\)](#), [Section 3.6 \(Traversal Rules and Side Effects\)](#)

### Notes:

If *sequence* is a *vector*, the result might or might not be simple, and might or might not be *identical* to *sequence*.

The **:test-not argument** is deprecated.

These functions are useful for converting *sequence* into a canonical form suitable for representing a set.

## Standard Generic Function REMOVE-METHOD

### Syntax:

**remove-method** *generic-function method* => *generic-function*

### Method Signatures:

**remove-method** (*generic-function standard-generic-function*) *method*

### Arguments and Values:

*generic-function*—a *generic function*.

*method*—a *method*.

### Description:

The *generic function remove-method* removes a *method* from *generic-function* by modifying the *generic-function* (if necessary).

**remove-method** must not signal an error if the *method* is not one of the *methods* on the *generic-function*.

**Examples:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**find-method**

**Notes:** None.

## **Function REMOVE, REMOVE-IF, REMOVE-IF-NOT, DELETE, DELETE-IF, DELETE-IF-NOT**

**Syntax:**

**remove** *item sequence &key from-end test test-not start end count key => result-sequence*

**remove-if** *test sequence &key from-end start end count key => result-sequence*

**remove-if-not** *test sequence &key from-end start end count key => result-sequence*

**delete** *item sequence &key from-end test test-not start end count key => result-sequence*

**delete-if** *test sequence &key from-end start end count key => result-sequence*

**delete-if-not** *test sequence &key from-end start end count key => result-sequence*

**Arguments and Values:**

*item*—an object.

*sequence*—a proper sequence.

*test*—a designator for a function of one argument that returns a generalized boolean.

*from-end*—a generalized boolean. The default is false.

*test*—a designator for a function of two arguments that returns a generalized boolean.

*test-not*—a designator for a function of two arguments that returns a generalized boolean.

*start, end*—bounding index designators of *sequence*. The defaults for *start* and *end* are 0 and nil, respectively.

## CLHS: Declaration DYNAMIC-EXTENT

*count*—an integer or nil. The default is nil.

*key*—a designator for a function of one argument, or nil.

*result-sequence*—a sequence.

### Description:

remove, remove-if, and remove-if-not return a sequence from which the elements that satisfy the test have been removed.

delete, delete-if, and delete-if-not are like remove, remove-if, and remove-if-not respectively, but they may modify sequence.

If *sequence* is a vector, the result is a vector that has the same actual array element type as *sequence*. If *sequence* is a list, the result is a list.

Supplying a *from-end* of true matters only when the *count* is provided; in that case only the rightmost *count* elements *satisfying the test* are deleted.

*Count*, if supplied, limits the number of elements removed or deleted; if more than *count* elements satisfy the test, then of these elements only the leftmost or rightmost, depending on *from-end*, are deleted or removed, as many as specified by *count*. If *count* is supplied and negative, the behavior is as if zero had been supplied instead. If *count* is nil, all matching items are affected.

For all these functions, elements not removed or deleted occur in the same order in the result as they did in *sequence*.

remove, remove-if, remove-if-not return a sequence of the same type as *sequence* that has the same elements except that those in the subsequence bounded by *start* and *end* and *satisfying the test* have been removed. This is a non-destructive operation. If any elements need to be removed, the result will be a copy. The result of remove may share with *sequence*; the result may be identical to the input *sequence* if no elements need to be removed.

delete, delete-if, and delete-if-not return a sequence of the same type as *sequence* that has the same elements except that those in the subsequence bounded by *start* and *end* and *satisfying the test* have been deleted. *Sequence* may be destroyed and used to construct the result; however, the result might or might not be identical to *sequence*.

delete, when *sequence* is a list, is permitted to setf any part, car or cdr, of the top-level list structure in that *sequence*. When *sequence* is a vector, delete is permitted to change the dimensions of the vector and to slide its elements into new positions without permuting them to produce the resulting vector.

delete-if is constrained to behave exactly as follows:

```
(delete nil sequence
      :test #'(lambda (ignore item) (funcall test item))
      ...)
```

### Examples:

```
(remove 4 '(1 3 4 5 9)) => (1 3 5 9)
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(remove 4 '(1 2 4 1 3 4 5)) => (1 2 1 3 5)
(remove 4 '(1 2 4 1 3 4 5) :count 1) => (1 2 1 3 4 5)
(remove 4 '(1 2 4 1 3 4 5) :count 1 :from-end t) => (1 2 4 1 3 5)
(remove 3 '(1 2 4 1 3 4 5) :test #'>) => (4 3 4 5)
(setq lst '(list of four elements)) => (LIST OF FOUR ELEMENTS)
(setq lst2 (copy-seq lst)) => (LIST OF FOUR ELEMENTS)
(setq lst3 (delete 'four lst)) => (LIST OF ELEMENTS)
(equal lst lst2) => false
(remove-if #'oddp '(1 2 4 1 3 4 5)) => (2 4 4)
(remove-if #'evenp '(1 2 4 1 3 4 5) :count 1 :from-end t)
=> (1 2 4 1 3 5)
(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)
(setq tester (list 1 2 4 1 3 4 5)) => (1 2 4 1 3 4 5)
(delete 4 tester) => (1 2 1 3 5)
(setq tester (list 1 2 4 1 3 4 5)) => (1 2 4 1 3 4 5)
(delete 4 tester :count 1) => (1 2 1 3 4 5)
(setq tester (list 1 2 4 1 3 4 5)) => (1 2 4 1 3 4 5)
(delete 4 tester :count 1 :from-end t) => (1 2 4 1 3 5)
(setq tester (list 1 2 4 1 3 4 5)) => (1 2 4 1 3 4 5)
(delete 3 tester :test #'>) => (4 3 4 5)
(setq tester (list 1 2 4 1 3 4 5)) => (1 2 4 1 3 4 5)
(delete-if #'oddp tester) => (2 4 4)
(setq tester (list 1 2 4 1 3 4 5)) => (1 2 4 1 3 4 5)
(delete-if #'evenp tester :count 1 :from-end t) => (1 2 4 1 3 5)
(setq tester (list 1 2 3 4 5 6)) => (1 2 3 4 5 6)
(delete-if #'evenp tester) => (1 3 5)
tester => implementation-dependent

(setq foo (list 'a 'b 'c)) => (A B C)
(setq bar (cdr foo)) => (B C)
(setq foo (delete 'b foo)) => (A C)
bar => ((C)) or ...
(eq (cdr foo) (car bar)) => T or ...
```

### Side Effects:

For delete, delete-if, and delete-if-not, *sequence* may be destroyed and used to construct the result.

**Affected By:** None.

### Exceptional Situations:

Should be prepared to signal an error of type type-error if *sequence* is not a proper sequence.

### See Also:

[Section 3.2.1 \(Compiler Terminology\)](#), [Section 3.6 \(Traversal Rules and Side Effects\)](#)

### Notes:

If *sequence* is a vector, the result might or might not be simple, and might or might not be identical to *sequence*.

The :test-not argument is deprecated.

The functions **delete-if-not** and **remove-if-not** are deprecated.

## Function RENAME-FILE

### Syntax:

**rename-file** *filespec new-name => defaulted-new-name, old-trlename, new-trlename*

### Arguments and Values:

*filespec*—a pathname designator.

*new-name*—a pathname designator other than a stream.

*defaulted-new-name*—a pathname

*old-trlename*—a physical pathname.

*new-trlename*—a physical pathname.

### Description:

**rename-file** modifies the file system in such a way that the file indicated by *filespec* is renamed to *defaulted-new-name*.

It is an error to specify a filename containing a wild component, for *filespec* to contain a nil component where the file system does not permit a nil component, or for the result of defaulting missing components of *new-name* from *filespec* to contain a nil component where the file system does not permit a nil component.

If *new-name* is a logical pathname, **rename-file** returns a logical pathname as its primary value.

**rename-file** returns three values if successful. The primary value, *defaulted-new-name*, is the resulting name which is composed of *new-name* with any missing components filled in by performing a merge-pathnames operation using *filespec* as the defaults. The secondary value, *old-trlename*, is the trlename of the file before it was renamed. The tertiary value, *new-trlename*, is the trlename of the file after it was renamed.

If the *filespec designator* is an open stream, then the stream itself and the file associated with it are affected (if the file system permits).

### Examples:

```
; ; An example involving logical pathnames.
(with-open-file (stream "sys:chemistry;lead.text"
                        :direction :output :if-exists :error)
  (princ "eureka" stream)
  (values (pathname stream) (trlename stream)))
=> #P"SYS:CHEMISTRY;LEAD.TEXT.NEWEST", #P"Q:>sys>chem>lead.text.1"
(rename-file "sys:chemistry;lead.text" "gold.text")
=> #P"SYS:CHEMISTRY;GOLD.TEXT.NEWEST",
#P"Q:>sys>chem>lead.text.1",
#P"Q:>sys>chem>gold.text.1"
```

**Affected By:** None.

**Exceptional Situations:**

If the renaming operation is not successful, an error of *type file-error* is signaled.

An error of *type file-error* might be signaled if *filespec* is *wild*.

**See Also:**

[truename](#), [pathname](#), [logical-pathname](#), [Section 20.1 \(File System Concepts\)](#), [Section 19.1.2 \(Pathnames as Filenames\)](#)

**Notes:** None.

## Function RENAME-PACKAGE

**Syntax:**

**rename-package** *package new-name &optional new-nicknames => package-object*

**Arguments and Values:**

*package*—a *package designator*.

*new-name*—a *package designator*.

*new-nicknames*—a *list* of *string designators*. The default is the *empty list*.

*package-object*—the renamed *package object*.

**Description:**

Replaces the name and nicknames of *package*. The old name and all of the old nicknames of *package* are eliminated and are replaced by *new-name* and *new-nicknames*.

The consequences are undefined if *new-name* or any *new-nickname* conflicts with any existing package names.

**Examples:**

```
(make-package 'temporary :nicknames '("TEMP")) => #<PACKAGE "TEMPORARY">
(rename-package 'temp 'ephemeral) => #<PACKAGE "EPHEMERAL">
(package-nicknames (find-package 'ephemeral)) => ()
(find-package 'temporary) => NIL
(rename-package 'ephemeral 'temporary '(temp fleeting))
=> #<PACKAGE "TEMPORARY">
(package-nicknames (find-package 'temp)) => ("TEMP" "FLEETING")
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**make-package**

**Notes:** None.

## **Function RANDOM-STATE-P**

**Syntax:**

**random-state-p** *object* => *generalized-boolean*

**Arguments and Values:**

*object*—an *object*.

*generalized-boolean*—a *generalized boolean*.

**Description:**

Returns *true* if *object* is of *type random-state*; otherwise, returns *false*.

**Examples:**

```
(random-state-p *random-state*) => true
(random-state-p (make-random-state)) => true
(random-state-p 'test-function) => false
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**make-random-state, \*random-state\***

**Notes:**

```
(random-state-p object) == (typep object 'random-state)
```

## **Function ROOM**

**Syntax:**

**room** &*optional x* => *implementation-dependent*

**Arguments and Values:**

*x*—one of **t**, **nil**, or **:default**.

#### Description:

**room** prints, to *standard output*, information about the state of internal storage and its management. This might include descriptions of the amount of memory in use and the degree of memory compaction, possibly broken down by internal data type if that is appropriate. The nature and format of the printed information is *implementation-dependent*. The intent is to provide information that a *programmer* might use to tune a *program* for a particular *implementation*.

(**room nil**) prints out a minimal amount of information. (**room t**) prints out a maximal amount of information. (**room**) or (**room :default**) prints out an intermediate amount of information that is likely to be useful.

**Examples:** None.

#### Side Effects:

Output to *standard output*.

#### Affected By:

**\*standard-output\***.

**Exceptional Situations:** None.

**See Also:** None.

**Notes:** None.

## Accessor ROW-MAJOR-AREF

#### Syntax:

**row-major-aref** *array index => element*

(**setf** (**row-major-aref** *array index*) *new-element*)

#### Arguments and Values:

*array*—an *array*.

*index*—a *valid array row-major index* for the *array*.

*element, new-element*—an *object*.

#### Description:

Considers *array* as a *vector* by viewing its *elements* in row-major order, and returns the *element* of that *vector* which is referred to by the given *index*.

**row-major-aref** is valid for use with **setf**.

**Examples:** None.

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**aref, array-row-major-index**

**Notes:**

```
(row-major-aref array index) ==
  (aref (make-array (array-total-size array)
                    :displaced-to array
                    :element-type (array-element-type array)))
  index)

(aref array i1 i2 ...) ==
  (row-major-aref array (array-row-major-index array i1 i2))
```

## **Function RPLACA, RPLACD**

**Syntax:**

**rplaca** *cons object => cons*

**rplacd** *cons object => cons*

**Pronunciation:**

**rplaca**: [,ree'plakuh] or [,ruh'plakuh]

**rplacd**: [,ree'plakduh] or [,ruh'plakduh] or [,ree'plakdee] or [,ruh'plakdee]

**Arguments and Values:**

*cons*---a cons.

*object*---an object.

**Description:**

**rplaca** replaces the car of the *cons* with *object*.

**rplacd** replaces the cdr of the *cons* with *object*.

**Examples:**

```
(defparameter *some-list* (list* 'one 'two 'three 'four)) => *some-list*
*some-list* => (ONE TWO THREE . FOUR)
(rplaca *some-list* 'uno) => (UNO TWO THREE . FOUR)
*some-list* => (UNO TWO THREE . FOUR)
(rplacd (last *some-list*) (list 'IV)) => (THREE IV)
*some-list* => (UNO TWO THREE IV)
```

**Side Effects:**

The *cons* is modified.

**Affected By:** None.

**Exceptional Situations:** None.

Should signal an error of type type-error if *cons* is not a cons.

**See Also:** None.

**Notes:** None.

## **Function RESTART-NAME**

**Syntax:**

**restart-name** *restart* => *name*

**Arguments and Values:**

*restart*—a restart.

*name*—a symbol.

**Description:**

Returns the name of the *restart*, or nil if the *restart* is not named.

**Examples:**

```
(restart-case
  (loop for restart in (compute-restarts)
        collect (restart-name restart))
  (casel () :report "Return 1." 1)
  (nil () :report "Return 2." 2)
  (case3 () :report "Return 3." 3)
  (casel () :report "Return 4." 4))
=> (CASE1 NIL CASE3 CASE1 ABORT)
;; In the example above the restart named ABORT was not created
;; explicitly, but was implicitly supplied by the system.
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**compute-restarts** **find-restart**

**Notes:** None.

## **Function SUBSTITUTE, SUBSTITUTE-IF, SUBSTITUTE-IF-NOT, NSUBSTITUTE, NSUBSTITUTE-IF, NSUBSTITUTE-IF-NOT**

**Syntax:**

**substitute** *newitem olditem sequence &key from-end test test-not start end count key*

=> *result-sequence*

**substitute-if** *newitem predicate sequence &key from-end start end count key*

=> *result-sequence*

**substitute-if-not** *newitem predicate sequence &key from-end start end count key*

=> *result-sequence*

**nsubstitute** *newitem olditem sequence &key from-end test test-not start end count key*

=> *sequence*

**nsubstitute-if** *newitem predicate sequence &key from-end start end count key*

=> *sequence*

**nsubstitute-if-not** *newitem predicate sequence &key from-end start end count key*

=> *sequence*

**Arguments and Values:**

*newitem*—an object.

*olditem*—an object.

## CLHS: Declaration DYNAMIC-EXTENT

*sequence*—a *proper sequence*.

*predicate*—a *designator* for a *function* of one *argument* that returns a *generalized boolean*.

*from-end*—a *generalized boolean*. The default is *false*.

*test*—a *designator* for a *function* of two *arguments* that returns a *generalized boolean*.

*test-not*—a *designator* for a *function* of two *arguments* that returns a *generalized boolean*.

*start, end*—*bounding index designators* of *sequence*. The defaults for *start* and *end* are 0 and *nil*, respectively.

*count*—an *integer* or *nil*. The default is *nil*.

*key*—a *designator* for a *function* of one argument, or *nil*.

*result-sequence*—a *sequence*.

### Description:

**substitute**, **substitute-if**, and **substitute-if-not** return a copy of *sequence* in which each *element* that *satisfies the test* has been replaced with *newitem*.

**nsubstitute**, **nsubstitute-if**, and **nsubstitute-if-not** are like **substitute**, **substitute-if**, and **substitute-if-not** respectively, but they may modify *sequence*.

If *sequence* is a *vector*, the result is a *vector* that has the same *actual array element type* as *sequence*. If *sequence* is a *list*, the result is a *list*.

*Count*, if supplied, limits the number of elements altered; if more than *count elements satisfy the test*, then of these *elements* only the leftmost or rightmost, depending on *from-end*, are replaced, as many as specified by *count*. If *count* is supplied and negative, the behavior is as if zero had been supplied instead. If *count* is *nil*, all matching items are affected.

Supplying a *from-end* of *true* matters only when the *count* is provided (and *non-nil*); in that case, only the rightmost *count elements satisfying the test* are removed (instead of the leftmost).

*predicate*, *test*, and *test-not* might be called more than once for each *sequence element*, and their side effects can happen in any order.

The result of all these functions is a *sequence* of the same *type* as *sequence* that has the same elements except that those in the subsequence *bounded* by *start* and *end* and *satisfying the test* have been replaced by *newitem*.

**substitute**, **substitute-if**, and **substitute-if-not** return a *sequence* which can share with *sequence* or may be *identical* to the input *sequence* if no elements need to be changed.

**nsubstitute** and **nsubstitute-if** are required to *setf* any *car* (if *sequence* is a *list*) or *aref* (if *sequence* is a *vector*) of *sequence* that is required to be replaced with *newitem*. If *sequence* is a *list*, none of the *cdrs* of the top-level *list* can be modified.

## CLHS: Declaration DYNAMIC-EXTENT

### Examples:

```
(substitute #\_. #\SPACE "0 2 4 6") => "0.2.4.6"
(substitute 9 4 '(1 2 4 1 3 4 5)) => (1 2 9 1 3 9 5)
(substitute 9 4 '(1 2 4 1 3 4 5) :count 1) => (1 2 9 1 3 4 5)
(substitute 9 4 '(1 2 4 1 3 4 5) :count 1 :from-end t)
=> (1 2 4 1 3 9 5)
(substitute 9 3 '(1 2 4 1 3 4 5) :test #'>) => (9 9 4 9 3 4 5)

(substitute-if 0 #'evenp '((1) (2) (3) (4)) :start 2 :key #'car)
=> ((1) (2) (3) 0)
(substitute-if 9 #'oddp '(1 2 4 1 3 4 5)) => (9 2 4 9 9 4 9)
(substitute-if 9 #'evenp '(1 2 4 1 3 4 5) :count 1 :from-end t)
=> (1 2 4 1 3 9 5)

(setq some-things (list 'a 'car 'b 'cdr 'c)) => (A CAR B CDR C)
(nsubstitute-if "function was here" #'fboundp some-things
                 :count 1 :from-end t) => (A CAR B "function was here" C)
some-things => (A CAR B "function was here" C)
(setq alpha-tester (copy-seq "ab ")) => "ab "
(nsubstitute-if-not #\z #'alpha-char-p alpha-tester) => "abz"
alpha-tester => "abz"
```

### Side Effects:

nsubstitute, nsubstitute-if, and nsubstitute-if-not modify *sequence*.

Affected By: None.

### Exceptional Situations:

Should be prepared to signal an error of type type-error if *sequence* is not a proper sequence.

### See Also:

subst, nsubst, Section 3.2.1 (Compiler Terminology), Section 3.6 (Traversal Rules and Side Effects)

### Notes:

If *sequence* is a vector, the result might or might not be simple, and might or might not be identical to *sequence*.

The :test-not argument is deprecated.

The functions substitute-if-not and nsubstitute-if-not are deprecated.

nsubstitute and nsubstitute-if can be used in for-effect-only positions in code.

Because the side-effecting variants (e.g., nsubstitute) potentially change the path that is being traversed, their effects in the presence of shared or circular structure may vary in surprising ways when compared to their non-side-effecting alternatives. To see this, consider the following side-effect behavior, which might be exhibited by some implementations:

```
(defun test-it (fn)
  (let ((x (cons 'b nil)))
```

```
(rplacd x x)
(funcall fn 'a 'b x :count 1)))
(test-it #'substitute) => (A . #1=(B . #1#))
(test-it #'nsubstitute) => (A . #1#)
```

## Function SEARCH

### Syntax:

**search** *sequence-1 sequence-2 &key from-end test test-not key start1 start2 end1 end2*

=> *position*

### Arguments and Values:

*Sequence-1*—a sequence.

*Sequence-2*—a sequence.

*from-end*—a generalized boolean. The default is false.

*test*—a designator for a function of two arguments that returns a generalized boolean.

*test-not*—a designator for a function of two arguments that returns a generalized boolean.

*key*—a designator for a function of one argument, or nil.

*start1, end1*—bounding index designators of *sequence-1*. The defaults for *start1* and *end1* are 0 and nil, respectively.

*start2, end2*—bounding index designators of *sequence-2*. The defaults for *start2* and *end2* are 0 and nil, respectively.

*position*—a bounding index of *sequence-2*, or nil.

### Description:

Searches *sequence-2* for a subsequence that matches *sequence-1*.

The implementation may choose to search *sequence-2* in any order; there is no guarantee on the number of times the test is made. For example, when *start-end* is true, the *sequence* might actually be searched from left to right instead of from right to left (but in either case would return the rightmost matching subsequence). If the search succeeds, **search** returns the offset into *sequence-2* of the first element of the leftmost or rightmost matching subsequence, depending on *from-end*; otherwise **search** returns nil.

If *from-end* is true, the index of the leftmost element of the rightmost matching subsequence is returned.

### Examples:

```
(search "dog" "it's a dog's life") => 7
(search '(0 1) '(2 4 6 1 3 5) :key #'oddp) => 2
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[Section 3.6 \(Traversal Rules and Side Effects\)](#)

**Notes:**

The `:test-not argument` is deprecated.

## Function SET

**Syntax:**

`set symbol value => value`

**Arguments and Values:**

*symbol*—a symbol.

*value*—an object.

**Description:**

`set` changes the contents of the *value cell* of *symbol* to the given *value*.

```
(set symbol value) == (setf (symbol-value symbol) value)
```

**Examples:**

```
(setf (symbol-value 'n) 1) => 1
(set 'n 2) => 2
(symbol-value 'n) => 2
(let ((n 3))
  (declare (special n))
  (setq n (+ n 1))
  (setf (symbol-value 'n) (* n 10))
  (set 'n (+ (symbol-value 'n) n))
  n) => 80
n => 2
(let ((n 3))
  (setq n (+ n 1))
  (setf (symbol-value 'n) (* n 10))
  (set 'n (+ (symbol-value 'n) n))
  n) => 4
n => 44
(defvar *n* 2)
(let ((*n* 3))
  (setq *n* (+ *n* 1))
  (setf (symbol-value '*n*) (* *n* 10)))
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(set '*n* (+ (symbol-value '*n*) *n*))
*n*) => 80
*n* => 2
(defvar *even-count* 0) => *EVEN-COUNT*
(defvar *odd-count* 0) => *ODD-COUNT*
(defun tally-list (list)
  (dolist (element list)
    (set (if (evenp element) '*even-count* '*odd-count*)
         (+ element (if (evenp element) *even-count* *odd-count*)))))
(tally-list '(1 9 4 3 2 7)) => NIL
*even-count* => 6
*odd-count* => 20
```

### Side Effects:

The value of *symbol* is changed.

Affected By: None.

Exceptional Situations: None.

### See Also:

[setq](#), [progv](#), [symbol-value](#)

### Notes:

The function set is deprecated.

set cannot change the value of a lexical variable.

## **Function SET-DISPATCH-MACRO-CHARACTER, GET-DISPATCH-MACRO-CHARACTER**

### Syntax:

**get-dispatch-macro-character** *disp-char sub-char &optional readtable => function*

**set-dispatch-macro-character** *disp-char sub-char new-function &optional readtable => t*

### Arguments and Values:

*disp-char*—a character.

*sub-char*—a character.

*readtable*—a readtable designator. The default is the current readtable.

*function*—a function designator or nil.

*new-function*—a function designator.

**Description:**

**set-dispatch-macro-character** causes *new-function* to be called when *disp-char* followed by *sub-char* is read. If *sub-char* is a lowercase letter, it is converted to its uppercase equivalent. It is an error if *sub-char* is one of the ten decimal digits.

**set-dispatch-macro-character** installs a *new-function* to be called when a particular *dispatching macro character* pair is read. *New-function* is installed as the dispatch function to be called when *readtable* is in use and when *disp-char* is followed by *sub-char*.

For more information about how the *new-function* is invoked, see [Section 2.1.4.4 \(Macro Characters\)](#).

**get-dispatch-macro-character** retrieves the dispatch function associated with *disp-char* and *sub-char* in *readtable*.

**get-dispatch-macro-character** returns the macro-character function for *sub-char* under *disp-char*, or **nil** if there is no function associated with *sub-char*. If *sub-char* is a decimal digit, **get-dispatch-macro-character** returns **nil**.

**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
```

If it is desired that #\$\_*foo* : as if it were (dollar\$\_*foo*).

```
(defun |#$-reader| (stream subchar arg)
  (declare (ignore subchar arg))
  (list 'dollars (read stream t nil t))) => |#$-reader|
(set-dispatch-macro-character #\$ #'|#$-reader|) => T
```

**See Also:****Side Effects:**

The *readtable* is modified.

**Affected By:**

**\*readtable\***.

**Exceptional Situations:**

For either function, an error is signaled if *disp-char* is not a *dispatching macro character* in *readtable*.

**See Also:****\*readtable\*****Notes:****Function SET-DIFFERENCE, NSET-DIFFERENCE****Syntax:****set-difference** *list-1 list-2 &key key test test-not => result-list***nset-difference** *list-1 list-2 &key key test test-not => result-list***Arguments and Values:***list-1*—a proper list.*list-2*—a proper list.*test*—a designator for a function of two arguments that returns a generalized boolean.*test-not*—a designator for a function of two arguments that returns a generalized boolean.*key*—a designator for a function of one argument, or nil.*result-list*—a list.**Description:****set-difference** returns a list of elements of *list-1* that do not appear in *list-2*.**nset-difference** is the destructive version of **set-difference**. It may destroy *list-1*.

For all possible ordered pairs consisting of one element from *list-1* and one element from *list-2*, the `:test` or `:test-not` function is used to determine whether they satisfy the test. The first argument to the `:test` or `:test-not` function is the part of an element of *list-1* that is returned by the `:key` function (if supplied); the second argument is the part of an element of *list-2* that is returned by the `:key` function (if supplied).

If `:key` is supplied, its argument is a *list-1* or *list-2* element. The `:key` function typically returns part of the supplied element. If `:key` is not supplied, the *list-1* or *list-2* element is used.

An element of *list-1* appears in the result if and only if it does not match any element of *list-2*.

There is no guarantee that the order of elements in the result will reflect the ordering of the arguments in any particular way. The result list may share cells with, or be eq to, either of *list-1* or *list-2*, if appropriate.

**Examples:**

## CLHS: Declaration DYNAMIC-EXTENT

```
(setq lst1 (list "A" "b" "C" "d")
  lst2 (list "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")
(setq lst1 '(("a" . "b") ("c" . "d") ("e" . "f")))
=> (("a" . "b") ("c" . "d") ("e" . "f"))
(setq lst2 '(("c" . "a") ("e" . "b") ("d" . "a")))
=> (("c" . "a") ("e" . "b") ("d" . "a"))
(nset-difference lst1 lst2 :test #'string= :key #'cdr)
=> ("(c" . "d") ("e" . "f"))
lst1 => (("a" . "b") ("c" . "d") ("e" . "f"))
lst2 => (("c" . "a") ("e" . "b") ("d" . "a"))

;; Remove all flavor names that contain "c" or "w".
(set-difference '("strawberry" "chocolate" "banana"
  "lemon" "pistachio" "rhubarb")
  '(#\c #\w)
  :test #'(lambda (s c) (find c s)))
=> ("banana" "rhubarb" "lemon") ;One possible ordering.
```

### Side Effects:

**nset-difference** may destroy *list-1*.

**Affected By:** None.

### Exceptional Situations:

Should be prepared to signal an error of **type-type-error** if *list-1* and *list-2* are not **proper lists**.

### See Also:

[Section 3.2.1 \(Compiler Terminology\)](#), [Section 3.6 \(Traversal Rules and Side Effects\)](#)

### Notes:

The **:test-not** parameter is deprecated.

## Function SET-EXCLUSIVE-OR, NSET-EXCLUSIVE-OR

### Syntax:

**set-exclusive-or** *list-1* *list-2* &**key** *key test test-not* => *result-list*

**nset-exclusive-or** *list-1* *list-2* &**key** *key test test-not* => *result-list*

### Arguments and Values:

*list-1*—a **proper list**.

*list-2*—a **proper list**.

*test*—a designator for a function of two arguments that returns a generalized boolean.

*test-not*—a designator for a function of two arguments that returns a generalized boolean.

*key*—a designator for a function of one argument, or nil.

*result-list*—a list.

### Description:

**set-exclusive-or** returns a list of elements that appear in exactly one of *list-1* and *list-2*.

**nset-exclusive-or** is the destructive version of **set-exclusive-or**.

For all possible ordered pairs consisting of one element from *list-1* and one element from *list-2*, the :*test* or :*test-not* function is used to determine whether they satisfy the test.

If :*key* is supplied, it is used to extract the part to be tested from the *list-1* or *list-2* element. The first argument to the :*test* or :*test-not* function is the part of an element of *list-1* extracted by the :*key* function (if supplied); the second argument is the part of an element of *list-2* extracted by the :*key* function (if supplied). If :*key* is not supplied or nil, the *list-1* or *list-2* element is used.

The result contains precisely those elements of *list-1* and *list-2* that appear in no matching pair.

The result list of **set-exclusive-or** might share storage with one of *list-1* or *list-2*.

### Examples:

```
(setq lst1 (list 1 "a" "b")
      lst2 (list 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")
(setq lst1 (list ((&a) . "b") ("c" . "d") ("e" . "f")))
=> ((&a) . "b") ("c" . "d") ("e" . "f"))
(setq lst2 (list ((&c) . "a") ("e" . "b") ("d" . "a")))
=> ((&c) . "a") ("e" . "b") ("d" . "a"))
(nset-exclusive-or lst1 lst2 :test #'string= :key #'cdr)
=> ((&c) . "d") ("e" . "f") ("c" . "a") ("d" . "a"))
lst1 => ((&a) . "b") ("c" . "d") ("e" . "f"))
lst2 => ((&c) . "a") ("d" . "a"))
```

### Side Effects:

**nset-exclusive-or** is permitted to modify any part, car or cdr, of the list structure of *list-1* or *list-2*.

**Affected By:** None.

### Exceptional Situations:

Should be prepared to signal an error of type type-error if *list-1* and *list-2* are not proper lists.

**See Also:**

[Section 3.2.1 \(Compiler Terminology\)](#), [Section 3.6 \(Traversal Rules and Side Effects\)](#)

**Notes:**

The `:test-not` parameter is deprecated.

Since the `nset-exclusive-or` side effect is not required, it should not be used in for-effect-only positions in portable code.

## Function SET-MACRO-CHARACTER, GET-MACRO-CHARACTER

**Syntax:**

**get-macro-character** *char &optional readtable => function, non-terminating-p*

**set-macro-character** *char new-function &optional non-terminating-p readtable => t*

**Arguments and Values:**

*char*—a character.

*non-terminating-p*—a generalized boolean. The default is false.

*readtable*—a readtable designator. The default is the current readtable.

*function*—nil, or a designator for a function of two arguments.

*new-function*—a function designator.

**Description:**

**get-macro-character** returns as its primary value, *function*, the reader macro function associated with *char* in *readtable* (if any), or else nil if *char* is not a macro character in *readtable*. The secondary value, *non-terminating-p*, is true if *char* is a non-terminating macro character; otherwise, it is false.

**set-macro-character** causes *char* to be a macro character associated with the reader macro function *new-function* (or the designator for *new-function*) in *readtable*. If *non-terminating-p* is true, *char* becomes a non-terminating macro character; otherwise it becomes a terminating macro character.

**Examples:**

```
(get-macro-character #\{} => NIL, false
(not (get-macro-character #\;)) => false
```

The following is a possible definition for the single-quote reader macro in standard syntax:

```
(defun single-quote-reader (stream char)
  (declare (ignore char)))
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(list 'quote (read stream t nil t))) => SINGLE-QUOTE-READER
(set-macro-character #'\` #'single-quote-reader) => T
```

Here `single-quote-reader` reads an *object* following the *single-quote* and returns a *list* of `quote` and that *object*. The *char* argument is ignored.

The following is a possible definition for the *semicolon reader macro* in *standard syntax*:

```
(defun semicolon-reader (stream char)
  (declare (ignore char))
  ; First swallow the rest of the current input line.
  ; End-of-file is acceptable for terminating the comment.
  (do () ((char= (read-char stream nil #\Newline t) #\Newline)))
  ; Return zero values.
  (values)) => SEMICOLON-READER
(set-macro-character #'\: #'semicolon-reader) => T
```

### Side Effects:

The *readtable* is modified.

**Affected By:** None.

**Exceptional Situations:** None.

### See Also:

**\*readtable\***

**Notes:** None.

## Function SET-PPRINT-DISPATCH

### Syntax:

**set-pprint-dispatch** *type-specifier function* &*optional priority table => nil*

### Arguments and Values:

*type-specifier*—a *type specifier*.

*function*—a *function*, a *function name*, or **nil**.

*priority*—a *real*. The default is 0.

*table*—a *pprint dispatch table*. The default is the *value* of **\*print-pprint-dispatch\***.

### Description:

Installs an entry into the *pprint dispatch table* which is *table*.

*Type-specifier* is the *key* of the entry. The first action of **set-pprint-dispatch** is to remove any pre-existing

## CLHS: Declaration DYNAMIC-EXTENT

entry associated with *type-specifier*. This guarantees that there will never be two entries associated with the same *type specifier* in a given *pprint dispatch table*. Equality of *type specifiers* is tested by *equal*.

Two values are associated with each *type specifier* in a *pprint dispatch table*: a *function* and a *priority*. The *function* must accept two arguments: the *stream* to which output is sent and the *object* to be printed. The *function* should *pretty print* the *object* to the *stream*. The *function* can assume that object satisfies the *type* given by *type-specifier*. The *function* must obey *\*print-readably\**. Any values returned by the *function* are ignored.

*Priority* is a priority to resolve conflicts when an object matches more than one entry.

It is permissible for *function* to be *nil*. In this situation, there will be no *type-specifier* entry in *table* after *set-pprint-dispatch* returns.

**Examples:** None.

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:**

An error is signaled if *priority* is not a *real*.

**See Also:** None.

**Notes:**

Since *pprint dispatch tables* are often used to control the pretty printing of Lisp code, it is common for the *type-specifier* to be an *expression* of the form

```
(cons car-type cdr-type)
```

This signifies that the corresponding object must be a cons cell whose *car* matches the *type specifier* *car-type* and whose *cdr* matches the *type specifier* *cdr-type*. The *cdr-type* can be omitted in which case it defaults to *t*.

## Function SET-SYNTAX-FROM-CHAR

**Syntax:**

**set-syntax-from-char** *to-char from-char &optional to-readtable from-readtable => t*

**Arguments and Values:**

*to-char*—a *character*.

*from-char*—a *character*.

*to-readtable*—a *readtable*. The default is the *current readtable*.

*from-readtable*—a *readtable designator*. The default is the *standard readtable*.

**Description:**

**set-syntax-from-char** makes the syntax of *to-char* in *to-readtable* be the same as the syntax of *from-char* in *from-readtable*.

**set-syntax-from-char** copies the *syntax types* of *from-char*. If *from-char* is a *macro character*, its *reader macro function* is copied also. If the character is a *dispatching macro character*, its entire dispatch table of *reader macro functions* is copied. The *constituent traits* of *from-char* are not copied.

A macro definition from a character such as " can be copied to another character; the standard definition for " looks for another character that is the same as the character that invoked it. The definition of ( can not be meaningfully copied to {, on the other hand. The result is that *lists* are of the form { a b c ), not { a b c }, because the definition always looks for a closing parenthesis, not a closing brace.

**Examples:**

```
(set-syntax-from-char #\7 #\: ) => T
123579 => 1235
```

**Side Effects:**

The *to-readtable* is modified.

**Affected By:**

The existing values in the *from-readtable*.

**Exceptional Situations:** None.

**See Also:**

**set-macro-character**, **make-dispatch-macro-character**, [Section 2.1.4 \(Character Syntax Types\)](#)

**Notes:**

The *constituent traits* of a *character* are "hard wired" into the parser for extended *tokens*. For example, if the definition of S is copied to \*, then \* will become a *constituent* that is *alphabetic*[2] but that cannot be used as a *short float exponent marker*. For further information, see [Section 2.1.4.2 \(Constituent Traits\)](#).

## Function SHADOW

**Syntax:**

**shadow** *symbol-names* &*optional package* => *t*

**Arguments and Values:**

*symbol-names*—a *designator* for a *list* of *string designators*.

*package*—a *package designator*. The default is the *current package*.

**Description:**

**shadow** assures that *symbols* with names given by *symbol-names* are *present* in the *package*.

Specifically, *package* is searched for *symbols* with the *names* supplied by *symbol-names*. For each such *name*, if a corresponding *symbol* is not *present* in *package* (directly, not by inheritance), then a corresponding *symbol* is created with that *name*, and inserted into *package* as an *internal symbol*. The corresponding *symbol*, whether pre-existing or newly created, is then added, if not already present, to the *shadowing symbols list* of *package*.

**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)

(make-package 'test-1) => #<PACKAGE "TEST-1">
(intern "TEST" (find-package 'test-1)) => TEST-1::TEST, NIL
(shadow 'test-1::test (find-package 'test-1)) => T
(shadow 'TEST (find-package 'test-1)) => T
(assert (not (null (member 'test-1::test (package-shadowing-symbols
                                         (find-package 'test-1))))))

(make-package 'test-2) => #<PACKAGE "TEST-2">
(intern "TEST" (find-package 'test-2)) => TEST-2::TEST, NIL
(export 'test-2::test (find-package 'test-2)) => T
(use-package 'test-2 (find-package 'test-1)) ;should not error
```

**Side Effects:**

**shadow** changes the state of the package system in such a way that the package consistency rules do not hold across the change.

**Affected By:**

Current state of the package system.

**Exceptional Situations:** None.

**See Also:**

[package-shadowing-symbols](#), [Section 11.1 \(Package Concepts\)](#)

**Notes:**

If a *symbol* with a name in *symbol-names* already exists in *package*, but by inheritance, the inherited symbol becomes *shadowed*[3] by a newly created *internal symbol*.

**Standard Generic Function SHARED-INITIALIZE****Syntax:**

**shared-initialize** *instance slot-names &rest initargs &key &allow-other-keys => instance*

**Method Signatures:**

**shared-initialize** (*instance standard-object*) *slot-names &rest initargs*

**Arguments and Values:**

*instance*—an *object*.

*slot-names*—a *list* or *t*.

*initargs*—a *list* of *keyword/value pairs* (of initialization argument *names* and *values*).

**Description:**

The generic function **shared-initialize** is used to fill the *slots* of an *instance* using *initargs* and :*initform* forms. It is called when an instance is created, when an instance is re-initialized, when an instance is updated to conform to a redefined *class*, and when an instance is updated to conform to a different *class*. The generic function **shared-initialize** is called by the system-supplied primary *method* for **initialize-instance**, **reinitialize-instance**, **update-instance-for-redefined-class**, and **update-instance-for-different-class**.

The generic function **shared-initialize** takes the following arguments: the *instance* to be initialized, a specification of a set of *slot-names accessible* in that *instance*, and any number of *initargs*. The arguments after the first two must form an *initialization argument list*. The system-supplied primary *method* on **shared-initialize** initializes the *slots* with values according to the *initargs* and supplied :*initform* forms. *Slot-names* indicates which *slots* should be initialized according to their :*initform* forms if no *initargs* are provided for those *slots*.

The system-supplied primary *method* behaves as follows, regardless of whether the *slots* are local or shared:

- If an *initarg* in the *initialization argument list* specifies a value for that *slot*, that value is stored into the *slot*, even if a value has already been stored in the *slot* before the *method* is run.
- Any *slots* indicated by *slot-names* that are still unbound at this point are initialized according to their :*initform* forms. For any such *slot* that has an :*initform* form, that *form* is evaluated in the lexical environment of its defining *defclass form* and the result is stored into the *slot*. For example, if a *before method* stores a value in the *slot*, the :*initform* form will not be used to supply a value for the *slot*.
- The rules mentioned in [Section 7.1.4 \(Rules for Initialization Arguments\)](#) are obeyed.

The *slots-names* argument specifies the *slots* that are to be initialized according to their :*initform* forms if no initialization arguments apply. It can be a *list* of *slot names*, which specifies the set of those *slot names*; or it can be the *symbol t*, which specifies the set of all of the *slots*.

**Examples:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[initialize-instance](#), [reinitialize-instance](#), [update-instance-for-redefined-class](#),  
[update-instance-for-different-class](#), [slot-boundp](#), [slot-makunbound](#), [Section 7.1 \(Object Creation and Initialization\)](#), [Section 7.1.4 \(Rules for Initialization Arguments\)](#), [Section 7.1.2 \(Declaring the Validity of Initialization Arguments\)](#)

**Notes:**

*Initargs* are declared as valid by using the `:initarg` option to [defclass](#), or by defining *methods* for [shared-initialize](#). The keyword name of each keyword parameter specifier in the *lambda list* of any *method* defined on [shared-initialize](#) is declared as a valid *initarg* name for all *classes* for which that *method* is applicable.

Implementations are permitted to optimize `:initform` forms that neither produce nor depend on side effects, by evaluating these *forms* and storing them into slots before running any [initialize-instance](#) methods, rather than by handling them in the primary [initialize-instance](#) method. (This optimization might be implemented by having the [allocate-instance](#) method copy a prototype instance.)

Implementations are permitted to optimize default initial value forms for *initargs* associated with slots by not actually creating the complete initialization argument *list* when the only *method* that would receive the complete *list* is the *method* on [standard-object](#). In this case default initial value forms can be treated like `:initform` forms. This optimization has no visible effects other than a performance improvement.

## Function SHADOWING-IMPORT

**Syntax:**

**shadowing-import** *symbols* &*optional package* => *t*

**Arguments and Values:**

*symbols* --- a *designator* for a *list of symbols*.

*package* --- a *package designator*. The default is the *current package*.

**Description:**

[shadowing-import](#) is like [import](#), but it does not signal an error even if the importation of a *symbol* would shadow some *symbol* already *accessible* in *package*.

[shadowing-import](#) inserts each of *symbols* into *package* as an internal symbol, regardless of whether another *symbol* of the same name is shadowed by this action. If a different *symbol* of the same name is already *present* in *package*, that *symbol* is first *uninterned* from *package*. The new *symbol* is added to *package*'s shadowing-symbols list.

[shadowing-import](#) does name-conflict checking to the extent that it checks whether a distinct existing *symbol* with the same name is *accessible*; if so, it is shadowed by the new *symbol*, which implies that it must be uninterned if it was *present* in *package*.

**Examples:**

```
(in-package "COMMON-LISP-USER") => #<PACKAGE "COMMON-LISP-USER">
(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)
```

**Side Effects:**

**shadowing-import** changes the state of the package system in such a way that the consistency rules do not hold across the change.

*package*'s shadowing-symbols list is modified.

**Affected By:**

Current state of the package system.

**Exceptional Situations:** None.

**See Also:**

**import, unintern, package-shadowing-symbols**

**Notes:** None.

## **Function SHORT-SITE-NAME, LONG-SITE-NAME**

**Syntax:**

**short-site-name** <no arguments> => *description*

**long-site-name** <no arguments> => *description*

**Arguments and Values:**

*description*—a string or nil.

**Description:**

**short-site-name** and **long-site-name** return a string that identifies the physical location of the computer hardware, or nil if no appropriate *description* can be produced.

**Examples:**

```
(short-site-name)
=> "MIT AI Lab"
OR=> "CMU-CSD"
(long-site-name)
```

## CLHS: Declaration DYNAMIC-EXTENT

```
=> "MIT Artificial Intelligence Laboratory"  
OR=> "CMU Computer Science Department"
```

**Side Effects:** None.

**Affected By:**

The implementation, the location of the computer hardware, and the installation/configuration process.

**Exceptional Situations:** None.

**See Also:** None.

**Notes:** None.

## Function SIGNAL

**Syntax:**

**signal** *datum &rest arguments => nil*

**Arguments and Values:**

*datum, arguments*—designators for a condition of default type simple-condition.

**Description:**

Signals the condition denoted by the given *datum* and *arguments*. If the condition is not handled, signal returns nil.

**Examples:**

```
(defun handle-division-conditions (condition)  
  (format t "Considering condition for division condition handling~%")  
  (when (and (typep condition 'arithmetic-error)  
            (eq '/ (arithmetic-error-operation condition)))  
    (invoke-debugger condition)))  
HANDLE-DIVISION-CONDITIONS  
(defun handle-other-arithmetic-errors (condition)  
  (format t "Considering condition for arithmetic condition handling~%")  
  (when (typep condition 'arithmetic-error)  
    (abort)))  
HANDLE-OTHER-ARITHMETIC-ERRORS  
(define-condition a-condition-with-no-handler (condition) ())  
A-CONDITION-WITH-NO-HANDLER  
(signal 'a-condition-with-no-handler)  
NIL  
(handler-bind ((condition #'handle-division-conditions)  
              (condition #'handle-other-arithmetic-errors))  
  (signal 'a-condition-with-no-handler))  
Considering condition for division condition handling  
Considering condition for arithmetic condition handling  
NIL  
(handler-bind ((arithmetic-error #'handle-division-conditions)  
              (arithmetic-error #'handle-other-arithmetic-errors))
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(signal 'arithmetic-error :operation '* :operands '(1.2 b))
Considering condition for division condition handling
Considering condition for arithmetic condition handling
Back to Lisp Toplevel
```

### Side Effects:

The debugger might be entered due to **\*break-on-signals\***.

Handlers for the condition being signaled might transfer control.

### Affected By:

Existing handler bindings.

### **\*break-on-signals\***

**Exceptional Situations:** None.

### See Also:

**\*break-on-signals\***, **error**, **simple-condition**, Section 9.1.4 (Signaling and Handling Conditions)

### Notes:

If `(typep datum *break-on-signals*)` *yields true*, the debugger is entered prior to beginning the signaling process. The **continue restart** can be used to continue with the signaling process. This is also true for all other *functions* and *macros* that should, might, or must *signal conditions*.

## Function SIGNUM

### Syntax:

**signum** *number* => *signed-prototype*

### Arguments and Values:

*number*—*a number*.

*signed-prototype*—*a number*.

### Description:

**signum** determines a numerical value that indicates whether *number* is negative, zero, or positive.

For a *rational*, **signum** returns one of -1, 0, or 1 according to whether *number* is negative, zero, or positive. For a *float*, the result is a *float* of the same format whose value is minus one, zero, or one. For a *complex* number *z*, (**signum** *z*) is a complex number of the same phase but with unit magnitude, unless *z* is a complex zero, in which case the result is *z*.

For *rational arguments*, **signum** is a rational function, but it may be irrational for *complex arguments*.

## CLHS: Declaration DYNAMIC-EXTENT

If *number* is a float, the result is a float. If *number* is a rational, the result is a rational. If *number* is a complex float, the result is a complex float. If *number* is a complex rational, the result is a complex, but it is implementation-dependent whether that result is a complex rational or a complex float.

### Examples:

```
(signum 0) => 0
(signum 99) => 1
(signum 4/5) => 1
(signum -99/100) => -1
(signum 0.0) => 0.0
(signum #c(0 33)) => #C(0.0 1.0)
(signum #c(7.5 10.0)) => #C(0.6 0.8)
(signum #c(0.0 -14.7)) => #C(0.0 -1.0)
(eql (signum -0.0) -0.0) => true
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**Notes:**

```
(signum x) == (if (zerop x) x (/ x (abs x)))
```

## Function SIN, COS, TAN

### Syntax:

**sin radians => number**

**cos radians => number**

**tan radians => number**

### Arguments and Values:

*radians*—a *number* given in radians.

*number*—a *number*.

### Description:

**sin**, **cos**, and **tan** return the sine, cosine, and tangent, respectively, of *radians*.

### Examples:

## CLHS: Declaration DYNAMIC-EXTENT

```
(sin 0) => 0.0
(cos 0.7853982) => 0.707107
(tan #c(0 1)) => #C(0.0 0.761594)
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:**

Should signal an error of **type-type-error** if *radians* is not a **number**. Might signal **arithmetic-error**.

**See Also:**

[asin](#), [acos](#), [atan](#), [Section 12.1.3.3 \(Rule of Float Substitutability\)](#)

**Notes:** None.

## **Function SINH, COSH, TANH, ASINH, ACOSH, ATANH**

**Syntax:**

**sinh** *number* => *result*

**cosh** *number* => *result*

**tanh** *number* => *result*

**asinh** *number* => *result*

**acosh** *number* => *result*

**atanh** *number* => *result*

**Arguments and Values:**

*number*—a **number**.

*result*—a **number**.

**Description:**

These functions compute the hyperbolic sine, cosine, tangent, arc sine, arc cosine, and arc tangent functions, which are mathematically defined for an argument *x* as given in the next figure.

Function	Definition
<b>sinh</b>	$\sinh(x) = \frac{e^x - e^{-x}}{2}$
<b>cosh</b>	$\cosh(x) = \frac{e^x + e^{-x}}{2}$
<b>tanh</b>	$\tanh(x) = \frac{\sinh(x)}{\cosh(x)}$
<b>asinh</b>	$\text{asinh}(x) = \ln(x + \sqrt{x^2 + 1})$
<b>acosh</b>	$\text{acosh}(x) = \ln(x + \sqrt{x^2 - 1})$
<b>atanh</b>	$\text{atanh}(x) = \frac{1}{2} \ln\left(\frac{1+x}{1-x}\right)$

## CLHS: Declaration DYNAMIC-EXTENT

Hyperbolic sine	$(e^x - e^{-x})/2$
Hyperbolic cosine	$(e^x + e^{-x})/2$
Hyperbolic tangent	$(e^x - e^{-x})/(e^x + e^{-x})$
Hyperbolic arc sine	$\log(x + \sqrt{1+x^2})$
Hyperbolic arc cosine	$2 \log(\sqrt{(x+1)/2} + \sqrt{(x-1)/2})$
Hyperbolic arc tangent	$(\log(1+x) - \log(1-x))/2$

**Figure 12–16. Mathematical definitions for hyperbolic functions**

The following definition for the inverse hyperbolic cosine determines the range and branch cuts:

$$\operatorname{arccosh} z = 2 \log(\sqrt{(z+1)/2} + \sqrt{(z-1)/2}).$$

The branch cut for the inverse hyperbolic cosine function lies along the real axis to the left of 1 (inclusive), extending indefinitely along the negative real axis, continuous with quadrant II and (between 0 and 1) with quadrant I. The range is that half-strip of the complex plane containing numbers whose real part is non-negative and whose imaginary part is between  $-\pi$  (exclusive) and  $\pi$  (inclusive). A number with real part zero is in the range if its imaginary part is between zero (inclusive) and  $\pi$  (inclusive).

The following definition for the inverse hyperbolic sine determines the range and branch cuts:

$$\operatorname{arcsinh} z = \log(z + \sqrt{1+z^2}).$$

The branch cut for the inverse hyperbolic sine function is in two pieces: one along the positive imaginary axis above  $i$  (inclusive), continuous with quadrant I, and one along the negative imaginary axis below  $-i$  (inclusive), continuous with quadrant III. The range is that strip of the complex plane containing numbers whose imaginary part is between  $-\pi/2$  and  $\pi/2$ . A number with imaginary part equal to  $-\pi/2$  is in the range if and only if its real part is non-positive; a number with imaginary part equal to  $\pi/2$  is in the range if and only if its imaginary part is non-negative.

The following definition for the inverse hyperbolic tangent determines the range and branch cuts:

$$\operatorname{arctanh} z = \log(1+z) - \log(1-z)/2.$$

Note that:

$$i \operatorname{arctan} z = \operatorname{arctanh} iz.$$

The branch cut for the inverse hyperbolic tangent function is in two pieces: one along the negative real axis to the left of  $-1$  (inclusive), continuous with quadrant III, and one along the positive real axis to the right of  $1$  (inclusive), continuous with quadrant I. The points  $-1$  and  $1$  are excluded from the domain. The range is that strip of the complex plane containing numbers whose imaginary part is between  $-\pi/2$  and  $\pi/2$ . A number with imaginary part equal to  $-\pi/2$  is in the range if and only if its real part is strictly negative; a number with imaginary part equal to  $\pi/2$  is in the range if and only if its imaginary part is strictly positive. Thus the range of the inverse hyperbolic tangent function is identical to that of the inverse hyperbolic sine function with the points  $-\pi/2i$  and  $\pi/2i$  excluded.

### Examples:

```
(sinh 0) => 0.0
(cosh (complex 0 -1)) => #C(0.540302 -0.0)
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:**

Should signal an error of type type-error if *number* is not a number. Might signal arithmetic-error.

**See Also:**

[log](#), [sqrt](#), [Section 12.1.3.3 \(Rule of Float Substitutability\)](#)

**Notes:**

The result of acosh may be a complex even if *number* is not a complex; this occurs when *number* is less than one. Also, the result of atanh may be a complex even if *number* is not a complex; this occurs when the absolute value of *number* is greater than one.

The branch cut formulae are mathematically correct, assuming completely accurate computation. Implementors should consult a good text on numerical analysis. The formulae given above are not necessarily the simplest ones for real-valued computations; they are chosen to define the branch cuts in desirable ways for the complex case.

## **Function /**

**Syntax:**

*/ number => reciprocal*

*/ numerator &rest denominators+ => quotient*

**Arguments and Values:**

*number, denominator*—a non-zero number.

*numerator, quotient, reciprocal*—a number.

**Description:**

The function / performs division or reciprocation.

If no *denominators* are supplied, the function / returns the reciprocal of *number*.

If at least one *denominator* is supplied, the function / divides the *numerator* by all of the *denominators* and returns the resulting *quotient*.

If each argument is either an integer or a ratio, and the result is not an integer, then it is a ratio.

The function / performs necessary type conversions.

## CLHS: Declaration DYNAMIC-EXTENT

If any *argument* is a *float* then the rules of floating-point contagion apply; see [Section 12.1.4 \(Floating-point Computations\)](#).

### Examples:

```
(/ 12 4) => 3
(/ 13 4) => 13/4
(/ -8) => -1/8
(/ 3 4 5) => 3/20
(/ 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)
```

**Affected By:** None.

### Exceptional Situations:

The consequences are unspecified if any *argument* other than the first is zero. If there is only one *argument*, the consequences are unspecified if it is zero.

Might signal **type-error** if some *argument* is not a *number*. Might signal **division-by-zero** if division by zero is attempted. Might signal **arithmetic-error**.

### See Also:

[floor](#), [ceiling](#), [truncate](#), [round](#)

**Notes:** None.

## Function SLEEP

### Syntax:

**sleep** *seconds* => *nil*

### Arguments and Values:

*seconds*—a non-negative *real*.

### Description:

Causes execution to cease and become dormant for approximately the seconds of real time indicated by *seconds*, whereupon execution is resumed.

### Examples:

```
(sleep 1) => NIL
;; Actually, since SLEEP is permitted to use approximate timing,
;; this might not always yield true, but it will often enough that
;; we felt it to be a productive example of the intent.
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(let ((then (get-universal-time))
      (now  (progn (sleep 10) (get-universal-time))))
  (>= (- now then) 10))
=> true
```

### Side Effects:

Causes processing to pause.

### Affected By:

The granularity of the scheduler.

### Exceptional Situations:

Should signal an error of type type-error if *seconds* is not a non-negative real.

**See Also:** None.

**Notes:** None.

## Function SLOT-BOUNDP

### Syntax:

**slot-boundp** *instance slot-name => generalized-boolean*

### Arguments and Values:

*instance*---an object.

*slot-name*---a symbol naming a slot of *instance*.

*generalized-boolean*---a generalized boolean.

### Description:

Returns true if the slot named *slot-name* in *instance* is bound; otherwise, returns false.

**Examples:** None.

**Affected By:** None.

### Exceptional Situations:

If no slot of the name *slot-name* exists in the *instance*, **slot-missing** is called as follows:

```
(slot-missing (class-of instance)
              instance
              slot-name
              'slot-boundp)
```

## CLHS: Declaration DYNAMIC-EXTENT

(If slot-missing is invoked and returns a value, a boolean equivalent to its primary value is returned by slot-boundp.)

The specific behavior depends on *instance's metaclass*. An error is never signaled if *instance* has metaclass standard-class. An error is always signaled if *instance* has metaclass built-in-class. The consequences are undefined if *instance* has any other metaclass—an error might or might not be signaled in this situation. Note in particular that the behavior for conditions and structures is not specified.

**See Also:**

slot-makunbound, slot-missing

**Notes:**

The function slot-boundp allows for writing after methods on initialize-instance in order to initialize only those slots that have not already been bound.

Although no implementation is required to do so, implementors are strongly encouraged to implement the function slot-boundp using the function slot-boundp-using-class described in the Metaobject Protocol.

## Function SLOT-EXISTS-P

**Syntax:**

**slot-exists-p** *object slot-name => generalized-boolean*

**Arguments and Values:**

*object*—an object.

*slot-name*—a symbol.

*generalized-boolean*—a generalized boolean.

**Description:**

Returns true if the *object* has a slot named *slot-name*.

**Examples:** None.

**Affected By:**

defclass, defstruct

**Exceptional Situations:** None.

**See Also:**

defclass, slot-missing

**Notes:**

Although no implementation is required to do so, implementors are strongly encouraged to implement the function slot-exists-p using the function slot-exists-p-using-class described in the Metaobject Protocol.

## **Function SLOT-MAKUNBOUND**

**Syntax:**

**slot-makunbound** *instance slot-name => instance*

**Arguments and Values:**

*instance* — instance.

*Slot-name* — a symbol.

**Description:**

The function slot-makunbound restores a slot of the name *slot-name* in an *instance* to the unbound state.

**Examples:** None.

**Affected By:** None.

**Exceptional Situations:**

If no slot of the name *slot-name* exists in the *instance*, slot-missing is called as follows:

```
(slot-missing (class-of instance)
             instance
             slot-name
             'slot-makunbound)
```

(Any values returned by slot-missing in this case are ignored by slot-makunbound.)

The specific behavior depends on *instance*'s metaclass. An error is never signaled if *instance* has metaclass standard-class. An error is always signaled if *instance* has metaclass built-in-class. The consequences are undefined if *instance* has any other metaclass—an error might or might not be signaled in this situation. Note in particular that the behavior for conditions and structures is not specified.

**See Also:**

slot-boundp, slot-missing

**Notes:**

Although no implementation is required to do so, implementors are strongly encouraged to implement the function slot-makunbound using the function slot-makunbound-using-class described in the Metaobject Protocol.

**Standard Generic Function SLOT-MISSING****Syntax:**

**slot-missing** *class object slot-name operation &optional new-value => result\**

**Method Signatures:**

**slot-missing** (*class t*) *object slot-name operation &optional new-value*

**Arguments and Values:**

*class*—the class of *object*.

*object*—an object.

*slot-name*—a symbol (the name of a would-be slot).

*operation*—one of the symbols **setf**, **slot-boundp**, **slot-makunbound**, or **slot-value**.

*new-value*—an object.

*result*—an object.

**Description:**

The generic function **slot-missing** is invoked when an attempt is made to access a slot in an *object* whose metaclass is **standard-class** and the slot of the name *slot-name* is not a name of a slot in that class. The default method signals an error.

The generic function **slot-missing** is not intended to be called by programmers. Programmers may write methods for it.

The generic function **slot-missing** may be called during evaluation of **slot-value**, (**setf slot-value**), **slot-boundp**, and **slot-makunbound**. For each of these operations the corresponding symbol for the *operation* argument is **slot-value**, **setf**, **slot-boundp**, and **slot-makunbound** respectively.

The optional *new-value* argument to **slot-missing** is used when the operation is attempting to set the value of the slot.

If **slot-missing** returns, its values will be treated as follows:

- If the *operation* is **setf** or **slot-makunbound**, any values will be ignored by the caller.
- If the *operation* is **slot-value**, only the primary value will be used by the caller, and all other values will be ignored.
- If the *operation* is **slot-boundp**, any boolean equivalent of the primary value of the method might be used, and all other values will be ignored.

**Examples:** None.

**Affected By:** None.

**Exceptional Situations:**

The default method on slot-missing signals an error of type error.

**See Also:**

defclass, slot-exists-p, slot-value

**Notes:**

The set of arguments (including the class of the instance) facilitates defining methods on the metaclass for slot-missing.

**Standard Generic Function SLOT-UNBOUND****Syntax:**

**slot-unbound** *class* *instance* *slot-name* => *result*\*

**Method Signatures:**

**slot-unbound** (*class* *t*) *instance* *slot-name*

**Arguments and Values:**

*class*—the class of the *instance*.

*instance*—the *instance* in which an attempt was made to read the unbound slot.

*slot-name*—the name of the unbound slot.

*result*—an object.

**Description:**

The generic function **slot-unbound** is called when an unbound slot is read in an *instance* whose metaclass is standard-class. The default method signals an error of type unbound-slot. The name slot of the unbound-slot condition is initialized to the name of the offending variable, and the instance slot of the unbound-slot condition is initialized to the offending instance.

The generic function **slot-unbound** is not intended to be called by programmers. Programmers may write methods for it. The function slot-unbound is called only indirectly by slot-value.

If **slot-unbound** returns, only the primary value will be used by the caller, and all other values will be ignored.

**Examples:** None.

**Affected By:** None.

**Exceptional Situations:**

The default method on **slot-unbound** signals an error of type unbound-slot.

**See Also:**

### **slot-makunbound**

**Notes:**

An unbound slot may occur if no :initform form was specified for the slot and the slot value has not been set, or if **slot-makunbound** has been called on the slot.

## **Function SLOT-VALUE**

**Syntax:**

**slot-value** *object slot-name => value*

**Arguments and Values:**

*object*—an object.

*name*—a symbol.

*value*—an object.

**Description:**

The function slot-value returns the value of the slot named slot-name in the object. If there is no slot named slot-name, **slot-missing** is called. If the slot is unbound, **slot-unbound** is called.

The macro setf can be used with **slot-value** to change the value of a slot.

**Examples:**

```
(defclass foo ()
  ((a :accessor foo-a :initarg :a :initform 1)
   (b :accessor foo-b :initarg :b)
   (c :accessor foo-c :initform 3)))
=> #<STANDARD-CLASS FOO 244020371>
(setq fool (make-instance 'foo :a 'one :b 'two))
=> #<FOO 36325624>
(slot-value fool 'a) => ONE
(slot-value fool 'b) => TWO
(slot-value fool 'c) => 3
(setf (slot-value fool 'a) 'uno) => UNO
(slot-value fool 'a) => UNO
(defmethod foo-method ((x foo))
  (slot-value x 'a))
=> #<STANDARD-METHOD FOO-METHOD (FOO) 42720573>
(foo-method fool) => UNO
```

**Affected By:** None.

**Exceptional Situations:**

If an attempt is made to read a slot and no slot of the name *slot-name* exists in the *object*, **slot-missing** is called as follows:

```
(slot-missing (class-of instance)
              instance
              slot-name
              'slot-value)
```

(If **slot-missing** is invoked, its *primary value* is returned by **slot-value**.)

If an attempt is made to write a slot and no slot of the name *slot-name* exists in the *object*, **slot-missing** is called as follows:

```
(slot-missing (class-of instance)
              instance
              slot-name
              'setf
              new-value)
```

(If **slot-missing** returns in this case, any *values* are ignored.)

The specific behavior depends on *object's metaclass*. An error is never signaled if *object* has metaclass standard-class. An error is always signaled if *object* has metaclass built-in-class. The consequences are unspecified if *object* has any other metaclass—an error might or might not be signaled in this situation. Note in particular that the behavior for conditions and structures is not specified.

**See Also:**

**slot-missing**, **slot-unbound**, **with-slots**

**Notes:**

Although no implementation is required to do so, implementors are strongly encouraged to implement the function slot-value using the function slot-value-using-class described in the Metaobject Protocol.

Implementations may optimize **slot-value** by compiling it inline.

## **Function SIMPLE-BIT-VECTOR-P**

**Syntax:**

**simple-bit-vector-p** *object* => *generalized-boolean*

**Arguments and Values:**

*object*---an object.

*generalized-boolean*---a generalized boolean.

**Description:**

## CLHS: Declaration DYNAMIC-EXTENT

Returns true if *object* is of type simple-bit-vector; otherwise, returns false.

### Examples:

```
(simple-bit-vector-p (make-array 6)) => false
(simple-bit-vector-p #*) => true
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

### See Also:

[simple-vector-p](#)

### Notes:

```
(simple-bit-vector-p object) == (typep object 'simple-bit-vector)
```

## **Function SIMPLE-CONDITION-FORMAT-CONTROL, SIMPLE-CONDITION-FORMAT-ARGUMENTS**

### Syntax:

**simple-condition-format-control** *condition* => *format-control*

**simple-condition-format-arguments** *condition* => *format-arguments*

### Arguments and Values:

*condition*—a condition of type simple-condition.

*format-control*—a format control.

*format-arguments*—a list.

### Description:

**simple-condition-format-control** returns the format control needed to process the *condition's format arguments*.

**simple-condition-format-arguments** returns a list of format arguments needed to process the *condition's format control*.

### Examples:

```
(setq foo (make-condition 'simple-condition
                           :format-control "Hi ~S"
```

## CLHS: Declaration DYNAMIC-EXTENT

```
:format-arguments '(ho))  
=> #<SIMPLE-CONDITION 26223553>  
(apply #'format nil (simple-condition-format-control foo)  
       (simple-condition-format-arguments foo))  
=> "Hi HO"
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[simple-condition](#), [Section 9.1 \(Condition System Concepts\)](#)

**Notes:** None.

## Function SIMPLE-STRING-P

**Syntax:**

`simple-string-p object => generalized-boolean`

**Arguments and Values:**

*object*—an [object](#).

*generalized-boolean*—a [generalized boolean](#).

**Description:**

Returns [true](#) if *object* is of [type simple-string](#); otherwise, returns [false](#).

**Examples:**

```
(simple-string-p "aaaaaa") => true  
(simple-string-p (make-array 6  
                      :element-type 'character  
                      :fill-pointer t)) => false
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:** None.

**Notes:**

```
(simple-string-p object) == (typep object 'simple-string)
```

**Function SIMPLE-VECTOR-P****Syntax:**

**simple-vector-p** *object* => *generalized-boolean*

**Arguments and Values:**

*object*---an *object*.

*generalized-boolean*---a *generalized boolean*.

**Description:**

Returns *true* if *object* is of type **simple-vector**; otherwise, returns *false*.

**Examples:**

```
(simple-vector-p (make-array 6)) => true
(simple-vector-p "aaaaaa") => false
(simple-vector-p (make-array 6 :fill-pointer t)) => false
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**simple-vector**

**Notes:**

```
(simple-vector-p object) == (typep object 'simple-vector)
```

**Function SORT, STABLE-SORT****Syntax:**

**sort** *sequence predicate &key key* => *sorted-sequence*

**stable-sort** *sequence predicate &key key* => *sorted-sequence*

**Arguments and Values:**

*sequence*---a *proper sequence*.

*predicate*---a *designator* for a *function* of two arguments that returns a *generalized boolean*.

*key*—a designator for a function of one argument, or nil.

*sorted-sequence*—a sequence.

### Description:

sort and stable-sort destructively sort *sequences* according to the order determined by the *predicate* function.

If *sequence* is a vector, the result is a vector that has the same actual array element type as *sequence*. If *sequence* is a list, the result is a list.

sort determines the relationship between two elements by giving keys extracted from the elements to the *predicate*. The first argument to the *predicate* function is the part of one element of *sequence* extracted by the *key* function (if supplied); the second argument is the part of another element of *sequence* extracted by the *key* function (if supplied). *Predicate* should return true if and only if the first argument is strictly less than the second (in some appropriate sense). If the first argument is greater than or equal to the second (in the appropriate sense), then the *predicate* should return false.

The argument to the *key* function is the *sequence* element. The return value of the *key* function becomes an argument to *predicate*. If *key* is not supplied or nil, the *sequence* element itself is used. There is no guarantee on the number of times the *key* will be called.

If the *key* and *predicate* always return, then the sorting operation will always terminate, producing a sequence containing the same elements as *sequence* (that is, the result is a permutation of *sequence*). This is guaranteed even if the *predicate* does not really consistently represent a total order (in which case the elements will be scrambled in some unpredictable way, but no element will be lost). If the *key* consistently returns meaningful keys, and the *predicate* does reflect some total ordering criterion on those keys, then the elements of the *sorted-sequence* will be properly sorted according to that ordering.

The sorting operation performed by sort is not guaranteed stable. Elements considered equal by the *predicate* might or might not stay in their original order. The *predicate* is assumed to consider two elements *x* and *y* to be equal if (*funcall predicate x y*) and (*funcall predicate y x*) are both false. stable-sort guarantees stability.

The sorting operation can be destructive in all cases. In the case of a vector argument, this is accomplished by permuting the elements in place. In the case of a list, the list is destructively reordered in the same manner as for nreverse.

### Examples:

```
(setq tester (copy-seq "lkjashd")) => "lkjashd"
(sort tester #'char-lessp) => "adhjkls"
(setq tester (list '(1 2 3) '(4 5 6) '(7 8 9))) => ((1 2 3) (4 5 6) (7 8 9))
(sort tester #'> :key #'car) => ((7 8 9) (4 5 6) (1 2 3))
(setq tester (list 1 2 3 4 5 6 7 8 9 0)) => (1 2 3 4 5 6 7 8 9 0)
(stable-sort tester #'(lambda (x y) (and (oddp x) (evenp y))))
=> (1 3 5 7 9 2 4 6 8 0)
(sort (setq committee-data
          (vector (list (list "JonL" "White") "Iteration")
                  (list (list "Dick" "Waters") "Iteration")
                  (list (list "Dick" "Gabriel") "Objects")
                  (list (list "Kent" "Pitman") "Conditions")
                  (list (list "Gregor" "Kiczales") "Objects")))
          committee-data)
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(list (list "David" "Moon") "Objects")
      (list (list "Kathy" "Chapman") "Editorial")
      (list (list "Larry" "Masinter") "Cleanup")
      (list (list "Sandra" "Loosemore") "Compiler")))
      #'string-lessp :key #'cadar)
=> #((( "Kathy" "Chapman") "Editorial")
     (( "Dick" "Gabriel") "Objects")
     (( "Gregor" "Kiczales") "Objects")
     (( "Sandra" "Loosemore") "Compiler")
     (( "Larry" "Masinter") "Cleanup")
     (( "David" "Moon") "Objects")
     (( "Kent" "Pitman") "Conditions")
     (( "Dick" "Waters") "Iteration")
     (( "JonL" "White") "Iteration"))
;; Note that individual alphabetical order within `committees'
;; is preserved.
(setq committee-data
      (stable-sort committee-data #'string-lessp :key #'cadr))
=> #((( "Larry" "Masinter") "Cleanup")
     (( "Sandra" "Loosemore") "Compiler")
     (( "Kent" "Pitman") "Conditions")
     (( "Kathy" "Chapman") "Editorial")
     (( "Dick" "Waters") "Iteration")
     (( "JonL" "White") "Iteration")
     (( "Dick" "Gabriel") "Objects")
     (( "Gregor" "Kiczales") "Objects")
     (( "David" "Moon") "Objects"))
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:**

Should be prepared to signal an error of *type type–error* if *sequence* is not a *proper sequence*.

**See Also:**

[merge](#), [Section 3.2.1 \(Compiler Terminology\)](#), [Section 3.6 \(Traversal Rules and Side Effects\)](#), [Section 3.7 \(Destructive Operations\)](#)

**Notes:**

If *sequence* is a *vector*, the result might or might not be simple, and might or might not be *identical* to *sequence*.

## Function SPECIAL-OPERATOR-P

**Syntax:**

**special-operator-p symbol => generalized-boolean**

**Arguments and Values:**

*symbol*—a *symbol*.

*generalized-boolean*---a generalized boolean.

**Description:**

Returns true if *symbol* is a special operator; otherwise, returns false.

**Examples:**

```
(special-operator-p 'if) => true
(special-operator-p 'car) => false
(special-operator-p 'one) => false
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:**

Should signal type-error if its argument is not a symbol.

**See Also:** None.

**Notes:**

Historically, this function was called `special-form-p`. The name was finally declared a misnomer and changed, since it returned true for special operators, not special forms.

## Function SQRT, ISQRT

**Syntax:**

**sqrt** *number* => *root*

**isqrt** *natural* => *natural-root*

**Arguments and Values:**

*number*, *root*---a number.

*natural*, *natural-root*---a non-negative integer.

**Description:**

**sqrt** and **isqrt** compute square roots.

**sqrt** returns the principal square root of *number*. If the *number* is not a complex but is negative, then the result is a complex.

**isqrt** returns the greatest integer less than or equal to the exact positive square root of *natural*.

## CLHS: Declaration DYNAMIC-EXTENT

If *number* is a positive *rational*, it is *implementation-dependent* whether *root* is a *rational* or a *float*. If *number* is a negative *rational*, it is *implementation-dependent* whether *root* is a *complex rational* or a *complex float*.

The mathematical definition of complex square root (whether or not minus zero is supported) follows:

```
(sqrt x) = (exp (/ (log x) 2))
```

The branch cut for square root lies along the negative real axis, continuous with quadrant II. The range consists of the right half-plane, including the non-negative imaginary axis and excluding the negative imaginary axis.

### Examples:

```
(sqrt 9.0) => 3.0
(sqrt -9.0) => #C(0.0 3.0)
(isqrt 9) => 3
(sqrt 12) => 3.4641016
(isqrt 12) => 3
(isqrt 300) => 17
(isqrt 325) => 18
(sqrt 25)
=> 5
OR=> 5.0
(isqrt 25) => 5
(sqrt -1) => #C(0.0 1.0)
(sqrt #c(0 2)) => #C(1.0 1.0)
```

**Side Effects:** None.

**Affected By:** None.

### Exceptional Situations:

The *function sqrt* should signal *type-error* if its argument is not a *number*.

The *function isqrt* should signal *type-error* if its argument is not a non-negative *integer*.

The functions *sqrt* and *isqrt* might signal *arithmetic-error*.

### See Also:

[exp](#), [log](#), [Section 12.1.3.3 \(Rule of Float Substitutability\)](#)

### Notes:

```
(isqrt x) == (values (floor (sqrt x)))
```

but it is potentially more efficient.

***Function \******Syntax:**

\* &rest numbers => product

**Arguments and Values:**

number---a number.

product---a number.

**Description:**

Returns the product of *numbers*, performing any necessary type conversions in the process. If no *numbers* are supplied, 1 is returned.

**Examples:**

```
(*) => 1
(*) 3 5) => 15
(* 1.0 #c(22 33) 55/98) => #C(12.346938775510203 18.520408163265305)
```

**Affected By:** None.

**Exceptional Situations:**

Might signal type-error if some argument is not a number. Might signal arithmetic-error.

**See Also:**

**Notes:** None.

***Function STANDARD-CHAR-P*****Syntax:**

standard-char-p character => generalized-boolean

**Arguments and Values:**

character---a character.

generalized-boolean---a generalized boolean.

**Description:**

Returns true if *character* is of type standard-char; otherwise, returns false.

**Examples:**

## CLHS: Declaration DYNAMIC-EXTENT

```
(standard-char-p #\Space) => true
(standard-char-p #\~) => true
;; This next example presupposes an implementation
;; in which #\Bell is a defined character.
(standard-char-p #\Bell) => false
```

**Affected By:** None.

**Exceptional Situations:**

Should signal an error of type type-error if *character* is not a character.

**See Also:** None.

**Notes:** None.

## Function STRING-TRIM, STRING-LEFT-TRIM, STRING-RIGHT-TRIM

**Syntax:**

**string-trim** *character-bag string* => *trimmed-string*

**string-left-trim** *character-bag string* => *trimmed-string*

**string-right-trim** *character-bag string* => *trimmed-string*

**Arguments and Values:**

*character-bag*—a sequence containing characters.

*string*—a string designator.

*trimmed-string*—a string.

**Description:**

**string-trim** returns a substring of *string*, with all characters in *character-bag* stripped off the beginning and end. **string-left-trim** is similar but strips characters off only the beginning; **string-right-trim** strips off only the end.

If no characters need to be trimmed from the *string*, then either *string* itself or a copy of it may be returned, at the discretion of the implementation.

All of these functions observe the fill pointer.

**Examples:**

```
(string-trim "abc" "abcaakaaakabcaaa") => "kaaak"
(string-trim '(\#\Space #\Tab #\Newline) " garbanzo beans
" ) => "garbanzo beans"
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(string-trim " (*)" " (*three (silly) words* ) ")
=> "three (silly) words"

(string-left-trim "abc" "labcabcbabc") => "labcabcbabc"
(string-left-trim " (*)" " (*three (silly) words* ) ")
=> "three (silly) words* ) "

(string-right-trim " (*)" " (*three (silly) words* ) ")
=> " (*three (silly) words"
```

**Side Effects:** None.

**Affected By:**

The implementation.

**Exceptional Situations:** None.

**See Also:** None.

**Notes:** None.

## **Function STRING-UPCASE, STRING-DOWNCASE, STRING-CAPITALIZE, NSTRING-UPCASE, NSTRING-DOWNCASE, NSTRING-CAPITALIZE**

**Syntax:**

**string-upcase** *string* &**key** *start end* => *cased-string*

**string-downcase** *string* &**key** *start end* => *cased-string*

**string-capitalize** *string* &**key** *start end* => *cased-string*

**nstring-upcase** *string* &**key** *start end* => *string*

**nstring-downcase** *string* &**key** *start end* => *string*

**nstring-capitalize** *string* &**key** *start end* => *string*

**Arguments and Values:**

*string*—a string designator. For **nstring-upcase**, **nstring-downcase**, and **nstring-capitalize**, the *string designator* must be a string.

*start, end*—bounding index designators of *string*. The defaults for *start* and *end* are 0 and **nil**, respectively.

*cased-string*—a string.

Function STRING-UPCASE, STRING-DOWNCASE, STRING-CAPITALIZE, NSTRING-UPCASE, NSTRING-DOWNCASE, NSTRING-CAPITALIZE

## CLHS: Declaration DYNAMIC-EXTENT

### Description:

**string-upcase, string-downcase, string-capitalize, nstring-upcase, nstring-downcase, nstring-capitalize**

nstring-capitalize change the case of the subsequence of *string bounded* by *start* and *end* as follows:

#### **string-upcase**

**string-upcase** returns a *string* just like *string* with all lowercase characters replaced by the corresponding uppercase characters. More precisely, each character of the result *string* is produced by applying the *function char-upcase* to the corresponding character of *string*.

#### **string-downcase**

**string-downcase** is like **string-upcase** except that all uppercase characters are replaced by the corresponding lowercase characters (using **char-downcase**).

#### **string-capitalize**

**string-capitalize** produces a copy of *string* such that, for every word in the copy, the first *character* of the "word," if it has *case*, is *uppercase* and any other *characters* with *case* in the word are *lowercase*. For the purposes of **string-capitalize**, a "word" is defined to be a consecutive subsequence consisting of *alphanumeric characters*, delimited at each end either by a non-*alphanumeric character* or by an end of the *string*.

#### **nstring-upcase, nstring-downcase, nstring-capitalize**

**nstring-upcase, nstring-downcase, and nstring-capitalize** are identical to **string-upcase, string-downcase, and string-capitalize** respectively except that they modify *string*.

For **string-upcase, string-downcase, and string-capitalize**, *string* is not modified. However, if no characters in *string* require conversion, the result may be either *string* or a copy of it, at the implementation's discretion.

### Examples:

```
(string-upcase "abcde") => "ABCDE"
(string-upcase "Dr. Livingston, I presume?")
=> "DR. LIVINGSTON, I PRESUME?"
(string-upcase "Dr. Livingston, I presume?" :start 6 :end 10)
=> "Dr. LiVINGston, I presume?"
(string-downcase "Dr. Livingston, I presume?")
=> "dr. livingston, i presume?"

(string-capitalize "elm 13c arthur;fig don't") => "Elm 13c Arthur;Fig Don'T"
(string-capitalize " hello ") => " Hello "
(string-capitalize "occluDeD cASEmenTs FOreSTAll iNADVertent DEFenestrATION")
=> "Occluded Casements Forestall Inadvertent Defenestration"
(string-capitalize 'kludgy-hash-search) => "Kludgy-Hash-Search"
(string-capitalize "DON'T!") => "Don'T!" ;not "Don't!"
(string-capitalize "pipe 13a, fool6c") => "Pipe 13a, Fool6c"

(setq str (copy-seq "0123ABCD890a")) => "0123ABCD890a"
(nstring-downcase str :start 5 :end 7) => "0123AbcD890a"
str => "0123AbcD890a"
```

### Side Effects:

**nstring-upcase, nstring-downcase, and nstring-capitalize** modify *string* as appropriate rather than constructing a new *string*.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[char-upcase](#), [char-downcase](#)

**Notes:**

The result is always of the same length as *string*.

**Function STRING=, STRING/=, STRING<, STRING>, STRING<=, STRING>=, STRING-EQUAL, STRING-NOT-EQUAL, STRING-LESSP, STRING-GREATERP, STRING-NOT-GREATERP, STRING-NOT-LESSP**

**Syntax:**

**string=** *string1 string2 &key start1 end1 start2 end2 => generalized-boolean*

**string/=** *string1 string2 &key start1 end1 start2 end2 => mismatch-index*

**string<** *string1 string2 &key start1 end1 start2 end2 => mismatch-index*

**string>** *string1 string2 &key start1 end1 start2 end2 => mismatch-index*

**string<=** *string1 string2 &key start1 end1 start2 end2 => mismatch-index*

**string>=** *string1 string2 &key start1 end1 start2 end2 => mismatch-index*

**string-equal** *string1 string2 &key start1 end1 start2 end2 => generalized-boolean*

**string-not-equal** *string1 string2 &key start1 end1 start2 end2 => mismatch-index*

**string-lessp** *string1 string2 &key start1 end1 start2 end2 => mismatch-index*

**string-greaterp** *string1 string2 &key start1 end1 start2 end2 => mismatch-index*

**string-not-greaterp** *string1 string2 &key start1 end1 start2 end2 => mismatch-index*

**string-not-lessp** *string1 string2 &key start1 end1 start2 end2 => mismatch-index*

Function STRING=, STRING/=, STRING<, STRING>, STRING<=, STRING>=, STRING-EQUAL43 STRING-

**Arguments and Values:**

*string1*—a string designator.

*string2*—a string designator.

*start1, end1*—bounding index designators of *string1*. The defaults for *start* and *end* are 0 and nil, respectively.

*start2, end2*—bounding index designators of *string2*. The defaults for *start* and *end* are 0 and nil, respectively.

*generalized-boolean*—a generalized boolean.

*mismatch-index*—a bounding index of *string1*, or nil.

**Description:**

These functions perform lexicographic comparisons on *string1* and *string2*. string= and string-equal are called equality functions; the others are called inequality functions. The comparison operations these functions perform are restricted to the subsequence of *string1 bounded* by *start1* and *end1* and to the subsequence of *string2 bounded* by *start2* and *end2*.

A string *a* is equal to a string *b* if it contains the same number of characters, and the corresponding characters are the same under char= or char-equal, as appropriate.

A string *a* is less than a string *b* if in the first position in which they differ the character of *a* is less than the corresponding character of *b* according to char< or char-lessp as appropriate, or if string *a* is a proper prefix of string *b* (of shorter length and matching in all the characters of *a*).

The equality functions return a *generalized boolean* that is true if the strings are equal, or false otherwise.

The inequality functions return a *mismatch-index* that is true if the strings are not equal, or false otherwise. When the *mismatch-index* is true, it is an integer representing the first character position at which the two substrings differ, as an offset from the beginning of *string1*.

The comparison has one of the following results:

string=

string= is true if the supplied substrings are of the same length and contain the same characters in corresponding positions; otherwise it is false.

string/=

string/= is true if the supplied substrings are different; otherwise it is false.

string-equal

string-equal is just like string= except that differences in case are ignored; two characters are considered to be the same if char-equal is true of them.

string<

string< is true if substring1 is less than substring2; otherwise it is false.

string>

string> is true if substring1 is greater than substring2; otherwise it is false.

string-lessp, string-greaterp

string-lessp and string-greaterp are exactly like string< and string>, respectively, except that distinctions between uppercase and lowercase letters are ignored. It is as if char-lessp were used instead of char< for comparing characters.

**string<=**

string<= is true if substring1 is less than or equal to substring2; otherwise it is false.

**string>=**

string>= is true if substring1 is greater than or equal to substring2; otherwise it is false.

**string-not-greaterp, string-not-lessp**

string-not-greaterp and string-not-lessp are exactly like string<= and string>=, respectively, except that distinctions between uppercase and lowercase letters are ignored. It is as if char-lessp were used instead of char< for comparing characters.

**Examples:**

```
(string= "foo" "foo") => true
(string= "foo" "Foo") => false
(string= "foo" "bar") => false
(string= "together" "frog" :start1 1 :end1 3 :start2 2) => true
(string-equal "foo" "Foo") => true
(string= "abcd" "01234abcd9012" :start2 5 :end2 9) => true
(string< "aaaa" "aaab") => 3
(string>= "aaaaaa" "aaaa") => 4
(string-not-greaterp "Abcde" "abcdE") => 5
(string-lessp "012AAAA789" "01aaab6" :start1 3 :end1 7
              :start2 2 :end2 6) => 6
(string-not-equal "AAAA" "aaaA") => false
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**char=**

**Notes:**

equal calls string= if applied to two strings.

**Function STRINGP**

**Syntax:**

**stringp** *object* => *generalized-boolean*

**Arguments and Values:**

*object*—an object.

*generalized-boolean*—a generalized boolean.

**Description:**

Returns true if *object* is of type string; otherwise, returns false.

**Examples:**

```
(stringp "aaaaaa") => true
(stringp #\a) => false
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[typep, string \(type\)](#)

**Notes:**

```
(stringp object) == (typep object 'string)
```

**Function STREAM-ELEMENT-TYPE****Syntax:**

**stream-element-type** *stream* => *typespec*

**Arguments and Values:**

*stream*—a stream.

*typespec*—a type specifier.

**Description:**

**stream-element-type** returns a type specifier that indicates the types of objects that may be read from or written to *stream*.

Streams created by open have an element type restricted to integer or a subtype of type character.

**Examples:**

```
; ; Note that the stream must accomodate at least the specified type,
; ; but might accomodate other types. Further note that even if it does
; ; accomodate exactly the specified type, the type might be specified in
; ; any of several ways.
(with-open-file (s "test" :element-type '(integer 0 1)
                     :if-exists :error
                     :direction :output)
  (stream-element-type s))
=> INTEGER
OR=> (UNSIGNED-BYTE 16)
OR=> (UNSIGNED-BYTE 8)
```

```
OR=> BIT
OR=> (UNSIGNED-BYTE 1)
OR=> (INTEGER 0 1)
OR=> (INTEGER 0 (2))
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:**

Should signal an error of type type-error if *stream* is not a stream.

**See Also:** None.

**Notes:** None.

## **Function STREAM-ERROR-STREAM**

**Syntax:**

**stream-error-stream** *condition => stream*

**Arguments and Values:**

*condition*—a condition of type stream-error.

*stream*—a stream.

**Description:**

Returns the offending stream of a condition of type stream-error.

**Examples:**

```
(with-input-from-string (s "(FOO")
  (handler-case (read s)
    (end-of-file (c)
      (format nil "~&End of file on ~S." (stream-error-stream c))))))
"End of file on #<String Stream>."
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

stream-error, Section 9 (Conditions)

**Notes:** None.

***Function STREAM-EXTERNAL-FORMAT*****Syntax:**

**stream-external-format** *stream* => *format*

**Arguments and Values:**

*stream*---a file stream.

*format*---an external file format.

**Description:**

Returns an external file format designator for the *stream*.

**Examples:**

```
(with-open-file (stream "test" :direction :output)
  (stream-external-format stream))
=> :DEFAULT
OR=> :ISO8859/1-1987
OR=> (:ASCII :SAIL)
OR=> ACME::PROPRIETARY-FILE-FORMAT-17
OR=> #<FILE-FORMAT :ISO646-1983 2343673>
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

the :external-format argument to the function open and the with-open-file macro.

**Notes:**

The *format* returned is not necessarily meaningful to other implementations.

***Function STREAMP*****Syntax:**

**streamp** *object* => *generalized-boolean*

**Arguments and Values:**

*object*---an object.

*generalized-boolean*---a generalized boolean.

**Description:**

Returns true if *object* is of type stream; otherwise, returns false.

streamp is unaffected by whether *object*, if it is a stream, is open or closed.

**Examples:**

```
(streamp *terminal-io*) => true
(streamp 1) => false
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:** None.

**Notes:**

```
(streamp object) == (typep object 'stream)
```

**Function STRING****Syntax:**

**string** *x* => *string*

**Arguments and Values:**

*x*—a string, a symbol, or a character.

*string*—a string.

**Description:**

Returns a string described by *x*; specifically:

- If *x* is a string, it is returned.
- If *x* is a symbol, its name is returned.
- If *x* is a character, then a string containing that one character is returned.
- string might perform additional, implementation-defined conversions.

**Examples:**

```
(string "already a string") => "already a string"
(string 'elm) => "ELM"
(string #\c) => "c"
```

**Affected By:** None.

**Exceptional Situations:**

In the case where a conversion is defined neither by this specification nor by the *implementation*, an error of ***type type-error*** is signaled.

**See Also:**

**coerce, string (type)**.

**Notes:**

**coerce** can be used to convert a *sequence* of *characters* to a *string*.

**prin1-to-string, princ-to-string, write-to-string**, or **format** (with a first argument of **nil**) can be used to get a *string* representation of a *number* or any other *object*.

## **Function SUBLIS, NSUBLIS**

**Syntax:**

**sublis** *alist tree &key key test test-not => new-tree*

**nsublis** *alist tree &key key test test-not => new-tree*

**Arguments and Values:**

*alist*—an *association list*.

*tree*—a *tree*.

*test*—a *designator* for a *function* of two *arguments* that returns a *generalized boolean*.

*test-not*—a *designator* for a *function* of two *arguments* that returns a *generalized boolean*.

*key*—a *designator* for a *function* of one argument, or **nil**.

*new-tree*—a *tree*.

**Description:**

**sublis** makes substitutions for *objects* in *tree* (a structure of *conses*). **nsublis** is like **sublis** but destructively modifies the relevant parts of the *tree*.

**sublis** looks at all subtrees and leaves of *tree*; if a subtree or leaf appears as a key in *alist* (that is, the key and the subtree or leaf *satisfy the test*), it is replaced by the *object* with which that key is associated. This operation is non-destructive. In effect, **sublis** can perform several **subst** operations simultaneously.

If **sublis** succeeds, a new copy of *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* is left unchanged, but the result tree may share cells with it.

## CLHS: Declaration DYNAMIC-EXTENT

**nsublis** is permitted to modify *tree* but otherwise returns the same values as **sublis**.

### Examples:

```
(sublis '(((x . 100) (z . zprime))
          '(plus x (minus g z x p) 4 . x))
=> (PLUS 100 (MINUS G ZPRIME 100 P) 4 . 100)
(sublis '(((+ x y) . (- x y)) ((- x y) . (+ x y)))
          '(* (/ (+ x y) (+ x p)) (- x y))
          :test #'equal)
=> (* (/ (- X Y) (+ X P)) (+ X Y))
(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))))
(setq tree2 ("one" ("one" "two") (("one" "Two" "three"))))
=> ("one" ("one" "two") (("one" "Two" "three")))
(sublis '((("two" . 2)) tree2)
=> ("one" ("one" "two") (("one" "Two" "three")))
tree2 => ("one" ("one" "two") (("one" "Two" "three")))
(sublis '((("two" . 2)) tree2 :test 'equal)
=> ("one" ("one" 2) (("one" "Two" "three")))

(nsulblis '((t . 'temp))
           tree1
           :key #'(lambda (x) (or (atom x) (< (list-length x) 3))))
=> ((QUOTE TEMP) (QUOTE TEMP) QUOTE TEMP)
```

### Side Effects:

**nsulblis** modifies *tree*.

**Affected By:** None.

**Exceptional Situations:** None.

### See Also:

**subst**, [Section 3.2.1 \(Compiler Terminology\)](#), [Section 3.6 \(Traversal Rules and Side Effects\)](#)

### Notes:

The `:test-not` parameter is deprecated.

Because the side-effecting variants (e.g., **nsulblis**) potentially change the path that is being traversed, their effects in the presence of shared or circular structure may vary in surprising ways when compared to their non-side-effecting alternatives. To see this, consider the following side-effect behavior, which might be exhibited by some implementations:

```
(defun test-it (fn)
  (let* ((shared-piece (list 'a 'b)))
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(data (list shared-piece shared-piece)))
(funcall fn '((a . b) (b . a)) data))
(test-it #'sublis) => ((B A) (B A))
(test-it #'nsublis) => ((A B) (A B))
```

## Accessor SUBSEQ

### Syntax:

**subseq** *sequence start &optional end => subsequence*

(**setf** (**subseq** *sequence start &optional end*) *new-subsequence*)

### Arguments and Values:

*sequence*—a *proper sequence*.

*start, end*—*bounding index designators* of *sequence*. The default for *end* is *nil*.

*subsequence*—a *proper sequence*.

*new-subsequence*—a *proper sequence*.

### Description:

**subseq** creates a *sequence* that is a copy of the subsequence of *sequence* bounded by *start* and *end*.

*Start* specifies an offset into the original *sequence* and marks the beginning position of the subsequence. *end* marks the position following the last element of the subsequence.

**subseq** always allocates a new *sequence* for a result; it never shares storage with an old *sequence*. The result subsequence is always of the same *type* as *sequence*.

If *sequence* is a *vector*, the result is a *fresh simple array* of *rank* one that has the same *actual array element type* as *sequence*. If *sequence* is a *list*, the result is a *fresh list*.

**setf** may be used with **subseq** to destructively replace *elements* of a subsequence with *elements* taken from a *sequence* of new values. 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 are not modified 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"
```

**Side Effects:** None.

**Affected By:** None.

### Exceptional Situations:

Should be prepared to signal an error of type type-error if *sequence* is not a proper sequence. Should be prepared to signal an error of type type-error if *new-subsequence* is not a proper sequence.

**See Also:**

replace

**Notes:** None.

## Function SUBSETP

### Syntax:

**subsetp** *list-1* *list-2* &**key** *key* **test** *test-not* => *generalized-boolean*

### Arguments and Values:

*list-1*---a proper list.

*list-2*---a proper list.

*test*---a designator for a function of two arguments that returns a generalized boolean.

*test-not*---a designator for a function of two arguments that returns a generalized boolean.

*key*---a designator for a function of one argument, or nil.

*generalized-boolean*---a generalized boolean.

### Description:

**subsetp** returns true if every element of *list-1* matches some element of *list-2*, and false otherwise.

Whether a list element is the same as another list element is determined by the functions specified by the keyword arguments. The first argument to the :**test** or :**test-not** function is typically part of an element of *list-1* extracted by the :**key** function; the second argument is typically part of an element of *list-2* extracted by the :**key** function.

The argument to the :**key** function is an element of either *list-1* or *list-2*; the return value is part of the element of the supplied list element. If :**key** is not supplied or nil, the *list-1* or *list-2* element itself is supplied to the :**test** or :**test-not** function.

### Examples:

```
(setq cosmos '(1 "a" (1 2))) => (1 "a" (1 2))
(subsetp '(1) cosmos) => true
(subsetp '((1 2)) cosmos) => false
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(subsetp '((1 2)) cosmos :test 'equal) => true
(subsetp '(1 "A") cosmos :test #'equalp) => true
(subsetp '(((1) (2)) '(((1) (2)))) => false
(subsetp '(((1) (2)) '(((1) (2)) :key #'car)) => true
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:**

Should be prepared to signal an error of type type-error if *list-1* and *list-2* are not proper lists.

**See Also:**

[Section 3.6 \(Traversal Rules and Side Effects\)](#)

**Notes:**

The :test-not parameter is deprecated.

## **Function SUBST, SUBST-IF, SUBST-IF-NOT, NSUBST, NSUBST-IF, NSUBST-IF-NOT**

**Syntax:**

**subst** *new old tree &key key test test-not => new-tree*

**subst-if** *new predicate tree &key key => new-tree*

**subst-if-not** *new predicate tree &key key => new-tree*

**ns subst** *new old tree &key key test test-not => new-tree*

**ns subst-if** *new predicate tree &key key => new-tree*

**ns subst-if-not** *new predicate tree &key key => new-tree*

**Arguments and Values:**

*new*—an object.

*old*—an object.

*predicate*—a symbol that names a function, or a function of one argument that returns a generalized boolean value.

*tree*—a tree.

*test*—a designator for a function of two arguments that returns a generalized boolean.

*test-not*—a designator for a function of two arguments that returns a generalized boolean.

*key*—a designator for a function of one argument, or nil.

*new-tree*—a tree.

### Description:

**subst**, **subst-if**, and **subst-if-not** perform substitution operations on *tree*. Each function searches *tree* for occurrences of a particular *old* item of an element or subexpression that *satisfies the test*.

**nsubst**, **nsubst-if**, and **nsubst-if-not** are like **subst**, **subst-if**, and **subst-if-not** respectively, except that the original *tree* is modified.

**subst** makes a copy of *tree*, substituting *new* for every subtree or leaf of *tree* (whether the subtree or leaf is a car or a cdr of its parent) such that *old* and the subtree or leaf satisfy the test.

**nsubst** is a destructive version of **subst**. The list structure of *tree* is altered by destructively replacing with *new* each leaf of the *tree* such that *old* and the leaf satisfy the test.

For **subst**, **subst-if**, and **subst-if-not**, 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* is left unchanged, but the result tree may share storage with it.

For **nsubst**, **nsubst-if**, and **nsubst-if-not** the original *tree* is modified and returned as the function result, but the result may not be eq to *tree*.

### 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)) => implementation-dependent
(subst 'tempest 'hurricane
      '(shakespeare wrote (the hurricane)))
=> (SHAKESPEARE WROTE (THE TEMPEST))
(subst 'foo 'nil '(shakespeare wrote (twelfth night)))
=> (SHAKESPEARE WROTE (TWELFTH NIGHT . FOO) . FOO)
(subst '(a . cons) '(old . pair)
      '((old . spice) ((old . shoes) old . pair) (old . pair)))
      :test #'equal)
=> ((OLD . SPICE) ((OLD . SHOES) A . CONS) (A . CONS))

(subst-if 5 #'listp tree1) => 5
(subst-if-not '(x) #'consp tree1)
=> (1 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)
```

**Side Effects:**

**nsubst, nsubst-if, and nsubst-if-not** might alter the *tree structure* of *tree*.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**substitute, nsubstitute, Section 3.2.1 (Compiler Terminology), Section 3.6 (Traversal Rules and Side Effects)**

**Notes:**

The :test-not parameter is deprecated.

The functions **subst-if-not** and **nsubst-if-not** are deprecated.

One possible definition of **subst**:

```
(defun subst (old new tree &rest x &key test test-not key)
  (cond ((satisfies-the-test old tree :test test
                               :test-not test-not :key key)
          new)
        ((atom tree) tree)
        (t (let ((a (apply #'subst old new (car tree) x))
                 (d (apply #'subst old new (cdr tree) x)))
            (if (and (eql a (car tree))
                     (eql d (cdr tree)))
                tree
                (cons a d))))))
```

**Function SUBTYPEP****Syntax:**

**subtypep type-1 type-2 &optional environment => subtype-p, valid-p**

**Arguments and Values:**

*type-1*—a *type specifier*.

*type-2*—a *type specifier*.

*environment*—an *environment object*. The default is **nil**, denoting the *null lexical environment* and the current *global environment*.

*subtype-p*—a *generalized boolean*.

*valid-p*—a generalized boolean.

### Description:

If *type-1* is a recognizable subtype of *type-2*, the first value is true. Otherwise, the first value is false, indicating that either *type-1* is not a subtype of *type-2*, or else *type-1* is a subtype of *type-2* but is not a recognizable subtype.

A second value is also returned indicating the ‘certainty’ of the first value. If this value is true, then the first value is an accurate indication of the subtype relationship. (The second value is always true when the first value is true.)

The next figure summarizes the possible combinations of values that might result.

Value 1	Value 2	Meaning
<u>true</u>	<u>true</u>	<i>type-1</i> is definitely a <u>subtype</u> of <i>type-2</i> .
<u>false</u>	<u>true</u>	<i>type-1</i> is definitely not a <u>subtype</u> of <i>type-2</i> .
<u>false</u>	<u>false</u>	<u>subtypep</u> could not determine the relationship, so <i>type-1</i> might or might not be a <u>subtype</u> of <i>type-2</i> .

**Figure 4–9. Result possibilities for subtypep**

subtypep is permitted to return the values false and false only when at least one argument involves one of these type specifiers: **and**, **eql**, the list form of **function**, **member**, **not**, **or**, **satisfies**, or **values**. (A type specifier ‘involves’ such a symbol if, after being type expanded, it contains that symbol in a position that would call for its meaning as a type specifier to be used.) One consequence of this is that if neither *type-1* nor *type-2* involves any of these type specifiers, then subtypep is obliged to determine the relationship accurately. In particular, subtypep returns the values true and true if the arguments are equal and do not involve any of these type specifiers.

subtypep never returns a second value of nil when both *type-1* and *type-2* involve only the names in Figure 4–2, or names of types defined by **defstruct**, **define-condition**, or **defclass**, or derived types that expand into only those names. While type specifiers listed in Figure 4–2 and names of **defclass** and **defstruct** can in some cases be implemented as derived types, subtypep regards them as primitive.

The relationships between types reflected by subtypep are those specific to the particular implementation. For example, if an implementation supports only a single type of floating-point numbers, in that implementation (subtypep ‘float’ ‘long-float’) returns the values true and true (since the two types are identical).

For all *T1* and *T2* other than \*, (array *T1*) and (array *T2*) are two different type specifiers that always refer to the same sets of things if and only if they refer to arrays of exactly the same specialized representation, i.e., if (upgraded-array-element-type ‘*T1*’) and (upgraded-array-element-type ‘*T2*’) return two different type specifiers that always refer to the same sets of objects. This is another way of saying that `(array type-specifier) and `(array , (upgraded-array-element-type ‘type-specifier’)) refer to the same set of specialized array representations. For all *T1* and *T2* other than \*, the intersection of (array *T1*) and (array *T2*) is the empty set if and only if they refer to arrays of different, distinct specialized representations.

Therefore,

```
(subtypep '(array T1) '(array T2)) => true
```

## CLHS: Declaration DYNAMIC-EXTENT

if and only if

```
(upgraded-array-element-type 'T1)  and  
(upgraded-array-element-type 'T2)
```

return two different type specifiers that always refer to the same sets of objects.

For all type-specifiers *T1* and *T2* other than \*,

```
(subtypep '(complex T1) '(complex T2)) => true, true
```

if:

1. *T1* is a subtype of *T2*, or
2. (*upgraded-complex-part-type* '*T1*) and (*upgraded-complex-part-type* '*T2*) return two different type specifiers that always refer to the same sets of objects; in this case, (*complex* *T1*) and (*complex* *T2*) both refer to the same specialized representation.

The values are false and true otherwise.

The form

```
(subtypep '(complex single-float) '(complex float))
```

must return true in all implementations, but

```
(subtypep '(array single-float) '(array float))
```

returns true only in implementations that do not have a specialized array representation for single floats distinct from that for other floats.

### Examples:

```
(subtypep 'compiled-function 'function) => true, true  
(subtypep 'null 'list) => true, true  
(subtypep 'null 'symbol) => true, true  
(subtypep 'integer 'string) => false, true  
(subtypep '(satisfies dummy) nil) => false, implementation-dependent  
(subtypep '(integer 1 3) '(integer 1 4)) => true, true  
(subtypep '(integer (0) (0)) 'nil) => true, true  
(subtypep 'nil '(integer (0) (0))) => true, true  
(subtypep '(integer (0) (0)) '(member)) => true, true ;or false, false  
(subtypep '(member) 'nil) => true, true ;or false, false  
(subtypep 'nil '(member)) => true, true ;or false, false
```

Let <aet-x> and <aet-y> be two distinct type specifiers that do not always refer to the same sets of objects in a given implementation, but for which make-array, will return an object of the same array type.

Thus, in each case,

```
(subtypep (array-element-type (make-array 0 :element-type '<aet-x>'))  
          (array-element-type (make-array 0 :element-type '<aet-y>')))  
=> true, true
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(subtypep (array-element-type (make-array 0 :element-type '<aet-y>))
          (array-element-type (make-array 0 :element-type '<aet-x>)))
=> true, true
```

If (array <aet-x>) and (array <aet-y>) are different names for exactly the same set of *objects*, these names should always refer to the same sets of *objects*. That implies that the following set of tests are also true:

```
(subtypep '(array <aet-x>) '(array <aet-y>)) => true, true
(subtypep '(array <aet-y>) '(array <aet-x>)) => true, true
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**Notes:**

The small differences between the **subtypep** specification for the **array** and **complex** types are necessary because there is no creation function for **complexes** which allows the specification of the resultant part type independently of the actual types of the parts. Thus in the case of the **type complex**, the actual type of the parts is referred to, although a **number** can be a member of more than one **type**. For example, 17 is of **type (mod 18)** as well as **type (mod 256)** and **type integer**; and 2.3f5 is of **type single-float** as well as **type float**.

## Accessor SVREF

**Syntax:**

```
svref simple-vector index => element
(setf (svref simple-vector index) new-element)
```

**Arguments and Values:**

*simple-vector*—a **simple vector**.

*index*—a **valid array index** for the **simple-vector**.

*element, new-element*—an **object** (whose **type** is a **subtype** of the **array element type** of the **simple-vector**).

**Description:**

**Accesses** the **element** of **simple-vector** specified by **index**.

**Examples:**

```
(simple-vector-p (setq v (vector 1 2 'sirens))) => true
(svref v 0) => 1
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(svref v 2) => SIRENS
(setf (svref v 1) 'newcomer) => NEWCOMER
v => #(1 NEWCOMER SIRENS)
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[aref, sbit, schar, vector](#), [Section 3.2.1 \(Compiler Terminology\)](#)

**Notes:**

**svref** is identical to **aref** except that it requires its first argument to be a *simple vector*.

```
(svref v i) == (aref (the simple-vector v) i)
```

## Function SOFTWARE-TYPE, SOFTWARE-VERSION

**Syntax:**

**software-type** <no arguments> => *description*

**software-version** <no arguments> => *description*

**Arguments and Values:**

*description*—a *string* or **nil**.

**Description:**

**software-type** returns a *string* that identifies the generic name of any relevant supporting software, or **nil** if no appropriate or relevant result can be produced.

**software-version** returns a *string* that identifies the version of any relevant supporting software, or **nil** if no appropriate or relevant result can be produced.

**Examples:**

```
(software-type) => "Multics"
(software-version) => "1.3x"
```

**Side Effects:** None.

**Affected By:**

Operating system environment.

**Exceptional Situations:** None.

**See Also:** None.

**Notes:**

This information should be of use to maintainers of the implementation.

## Function SXHASH

**Syntax:**

**sxhash** *object* => *hash-code*

**Arguments and Values:**

*object*—an object.

*hash-code*—a non-negative fixnum.

**Description:**

**sxhash** returns a hash code for *object*.

The manner in which the hash code is computed is implementation-dependent, but subject to certain constraints:

1. (*equal* *x* *y*) implies (= (sxhash *x*) (sxhash *y*)).
2. For any two objects, *x* and *y*, both of which are bit vectors, characters, conses, numbers, pathnames, strings, or symbols, and which are similar, (sxhash *x*) and (sxhash *y*) yield the same mathematical value even if *x* and *y* exist in different Lisp images of the same implementation. See Section 3.2.4 (Literal Objects in Compiled Files).
3. The hash-code for an object is always the same within a single session provided that the object is not visibly modified with regard to the equivalence test equal. See Section 18.1.2 (Modifying Hash Table Keys).
4. The hash-code is intended for hashing. This places no verifiable constraint on a conforming implementation, but the intent is that an implementation should make a good-faith effort to produce hash-codes that are well distributed within the range of non-negative fixnums.
5. Computation of the hash-code must terminate, even if the object contains circularities.

**Examples:**

```
(= (sxhash (list 'list "ab")) (sxhash (list 'list "ab")))) => true
(= (sxhash "a") (sxhash (make-string 1 :initial-element #\a))) => true
(let ((r (make-random-state)))
  (= (sxhash r) (sxhash (make-random-state r))))
=> implementation-dependent
```

**Side Effects:** None.

**Affected By:**

The implementation.

**Exceptional Situations:** None.

**See Also:** None.

#### Notes:

Many common hashing needs are satisfied by make-hash-table and the related functions on hash tables. sxhash is intended for use where the pre-defined abstractions are insufficient. Its main intent is to allow the user a convenient means of implementing more complicated hashing paradigms than are provided through hash tables.

The hash codes returned by sxhash are not necessarily related to any hashing strategy used by any other function in Common Lisp.

For objects of types that equal compares with eq, item 3 requires that the hash-code be based on some immutable quality of the identity of the object. Another legitimate implementation technique would be to have sxhash assign (and cache) a random hash code for these objects, since there is no requirement that similar but non-eq objects have the same hash code.

Although similarity is defined for symbols in terms of both the symbol's name and the packages in which the symbol is accessible, item 3 disallows using package information to compute the hash code, since changes to the package status of a symbol are not visible to equal.

## Accessor SYMBOL-FUNCTION

#### Syntax:

**symbol-function** *symbol* => *contents*

(**setf** (**symbol-function** *symbol*) *new-contents*)

#### Arguments and Values:

*symbol*---a symbol.

*contents*--- If the symbol is globally defined as a macro or a special operator, an object of implementation-dependent nature and identity is returned. If the symbol is not globally defined as either a macro or a special operator, and if the symbol is fbound, a function object is returned.

*new-contents*---a function.

#### Description:

Accesses the symbol's function cell.

#### Examples:

```
(symbol-function 'car) => #<FUNCTION CAR>
(symbol-function 'twice) is an error ;because TWICE isn't defined.
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(defun twice (n) (* n 2)) => TWICE
(symbol-function 'twice) => #<FUNCTION TWICE>
(list (twice 3)
      (funcall (function twice) 3)
      (funcall (symbol-function 'twice) 3))
=> (6 6 6)
(flet ((twice (x) (list x x)))
  (list (twice 3)
        (funcall (function twice) 3)
        (funcall (symbol-function 'twice) 3)))
=> ((3 3) (3 3) 6)
(setf (symbol-function 'twice) #'(lambda (x) (list x x)))
=> #<FUNCTION anonymous>
(list (twice 3)
      (funcall (function twice) 3)
      (funcall (symbol-function 'twice) 3))
=> ((3 3) (3 3) (3 3))
(fboundp 'defun) => true
(symbol-function 'defun)
=> implementation-dependent
(functionp (symbol-function 'defun))
=> implementation-dependent
(defun symbol-function-or-nil (symbol)
  (if (and (fboundp symbol)
            (not (macro-function symbol))
            (not (special-operator-p symbol)))
      (symbol-function symbol)
      nil)) => SYMBOL-FUNCTION-OR-NIL
(symbol-function-or-nil 'car) => #<FUNCTION CAR>
(symbol-function-or-nil 'defun) => NIL
```

**Side Effects:** None.

**Affected By:**

**defun**

**Exceptional Situations:**

Should signal an error of **type-type-error** if *symbol* is not a *symbol*.

Should signal **undefined-function** if *symbol* is not *fbound* and an attempt is made to *read* its definition. (No such error is signaled on an attempt to *write* its definition.)

**See Also:**

**fboundp, fmakunbound, macro-function, special-operator-p**

**Notes:**

**symbol-function** cannot *access* the value of a lexical function name produced by **flet** or **labels**; it can *access* only the global function value.

**setf** may be used with **symbol-function** to replace a global function definition when the *symbol*'s function definition does not represent a *special operator*.

## CLHS: Declaration DYNAMIC-EXTENT

```
(symbol-function symbol) == (fdefinition symbol)
```

However, **fdefinition** accepts arguments other than just symbols.

### **Function SYMBOL-NAME**

**Syntax:**

```
symbol-name symbol => name
```

**Arguments and Values:**

*symbol*---a symbol.

*name*---a string.

**Description:**

**symbol-name** returns the name of *symbol*. The consequences are undefined if *name* is ever modified.

**Examples:**

```
(symbol-name 'temp) => "TEMP"  
(symbol-name :start) => "START"  
(symbol-name (gensym)) => "G1234" ;for example
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:**

Should signal an error of type type-error if *symbol* is not a symbol.

**See Also:** None.

**Notes:** None.

### **Function SYMBOL-PACKAGE**

**Syntax:**

```
symbol-package symbol => contents
```

**Arguments and Values:**

*symbol*---a symbol.

*contents*---a package object or nil.

**Description:**

## CLHS: Declaration DYNAMIC-EXTENT

Returns the home package of *symbol*.

### Examples:

```
(in-package "CL-USER") => #<PACKAGE "COMMON-LISP-USER">
(symbol-package 'car) => #<PACKAGE "COMMON-LISP">
(symbol-package 'bus) => #<PACKAGE "COMMON-LISP-USER">
(symbol-package :optional) => #<PACKAGE "KEYWORD">
;; Gensyms are uninterned, so have no home package.
(symbol-package (gensym)) => NIL
(make-package 'pk1) => #<PACKAGE "PK1">
(intern "SAMPLE1" "PK1") => PK1::SAMPLE1, NIL
(export (find-symbol "SAMPLE1" "PK1") "PK1") => T
(make-package 'pk2 :use '(pk1)) => #<PACKAGE "PK2">
(find-symbol "SAMPLE1" "PK2") => PK1:SAMPLE1, :INHERITED
(symbol-package 'pk1::sample1) => #<PACKAGE "PK1">
(symbol-package 'pk2::sample1) => #<PACKAGE "PK1">
(symbol-package 'pk1::sample2) => #<PACKAGE "PK1">
(symbol-package 'pk2::sample2) => #<PACKAGE "PK2">
;; The next several forms create a scenario in which a symbol
;; is not really uninterned, but is "apparently uninterned",
;; and so SYMBOL-PACKAGE still returns NIL.
(setq s3 'pk1::sample3) => PK1::SAMPLE3
(import s3 'pk2) => T
(unintern s3 'pk1) => T
(symbol-package s3) => NIL
(eq s3 'pk2::sample3) => T
```

**Side Effects:** None.

### Affected By:

import, intern, unintern

### Exceptional Situations:

Should signal an error of type type-error if *symbol* is not a symbol.

### See Also:

intern

**Notes:** None.

## Accessor SYMBOL-PLIST

### Syntax:

**symbol-plist** *symbol* => *plist*

```
(setf (symbol-plist symbol) new-plist)
```

### Arguments and Values:

*symbol*---a symbol.

*plist, new-plist*---a property list.

#### Description:

Accesses the property list of *symbol*.

#### Examples:

```
(setq sym (gensym)) => #:G9723
(symbol-plist sym) => ()
(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) (list 'prop3 'val3)) => (PROP3 VAL3)
(symbol-plist sym) => (PROP3 VAL3)
```

**Side Effects:** None.

**Affected By:** None.

#### Exceptional Situations:

Should signal an error of type type-error if *symbol* is not a symbol.

#### See Also:

get, remprop

#### Notes:

The use of setf should be avoided, since a symbol's property list is a global resource that can contain information established and depended upon by unrelated programs in the same Lisp image.

## Accessor SYMBOL-VALUE

#### Syntax:

**symbol-value** *symbol* => *value*

```
(setf (symbol-value symbol) new-value)
```

#### Arguments and Values:

*symbol*---a symbol that must have a value.

*value, new-value*---an object.

#### Description:

Accesses the symbol's value cell.

### Examples:

```
(setf (symbol-value 'a) 1) => 1
(symbol-value 'a) => 1
;; SYMBOL-VALUE cannot see lexical variables.
(let ((a 2)) (symbol-value 'a)) => 1
(let ((a 2)) (setq a 3) (symbol-value 'a)) => 1
;; SYMBOL-VALUE can see dynamic variables.
(let ((a 2))
  (declare (special a))
  (symbol-value 'a)) => 2
(let ((a 2))
  (declare (special a))
  (setq a 3)
  (symbol-value 'a)) => 3
(let ((a 2))
  (setf (symbol-value 'a) 3)
  a) => 2
a => 3
(symbol-value 'a) => 3
(let ((a 4))
  (declare (special a))
  (let ((b (symbol-value 'a)))
    (setf (symbol-value 'a) 5)
    (values a b))) => 5, 4
a => 3
(symbol-value :any-keyword) => :ANY-KEYWORD
(symbol-value 'nil) => NIL
(symbol-value '()) => NIL
;; The precision of this next one is implementation-dependent.
(symbol-value 'pi) => 3.141592653589793d0
```

**Side Effects:** None.

### Affected By:

makunbound, set, setq

### Exceptional Situations:

Should signal an error of type type-error if symbol is not a symbol.

Should signal unbound-variable if symbol is unbound and an attempt is made to read its value. (No such error is signaled on an attempt to write its value.)

### See Also:

boundp, makunbound, set, setq

### Notes:

symbol-value can be used to get the value of a constant variable. symbol-value cannot access the value of a lexical variable.

***Function SYMBOLP*****Syntax:*****symbolp object => generalized-boolean*****Arguments and Values:***object*—an object.*generalized-boolean*—a generalized boolean.**Description:**Returns true if *object* is of type symbol; otherwise, returns false.**Examples:**

```
(symbolp 'elephant) => true
(symbolp 12) => false
(symbolp nil) => true
(symbolp '()) => true
(symbolp :test) => true
(symbolp "hello") => false
```

**Side Effects:** None.**Affected By:** None.**Exceptional Situations:** None.**See Also:****keywordp, symbol, typep****Notes:**(symbolp *object*) == (typep *object* 'symbol)***Function SYNONYM-STREAM-SYMBOL*****Syntax:*****synonym-stream-symbol synonym-stream => symbol*****Arguments and Values:***synonym-stream*—a synonym stream.*symbol*—a symbol.

**Description:**

Returns the *symbol* whose **symbol-value** the *synonym-stream* is using.

**Examples:** None.**Side Effects:** None.**Affected By:** None.**Exceptional Situations:** None.**See Also:****make-synonym-stream****Notes:** None.**Function TERPRI, FRESH-LINE****Syntax:**

**terpri** &optional *output-stream* => *nil*

**fresh-line** &optional *output-stream* => *generalized-boolean*

**Arguments and Values:**

*output-stream* -- an *output stream designator*. The default is *standard output*.

*generalized-boolean* -- a *generalized boolean*.

**Description:**

**terpri** outputs a *newline* to *output-stream*.

**fresh-line** is similar to **terpri** but outputs a *newline* only if the *output-stream* is not already at the start of a line. If for some reason this cannot be determined, then a *newline* is output anyway. **fresh-line** returns *true* if it outputs a *newline*; otherwise it returns *false*.

**Examples:**

```
(with-output-to-string (s)
  (write-string "some text" s)
  (terpri s)
  (terpri s)
  (write-string "more text" s))
=> "some text

more text"
(with-output-to-string (s)
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(write-string "some text" s)
(fresh-line s)
(fresh-line s)
(write-string "more text" s)
=> "some text
more text"
```

### Side Effects:

The *output-stream* is modified.

### Affected By:

\*standard-output\*, \*terminal-io\*.

### Exceptional Situations:

None.

**See Also:** None.

### Notes:

terpri is identical in effect to

```
(write-char #\Newline output-stream)
```

## Function TRUENAME

### Syntax:

**truename** *filespec* => *truename*

### Arguments and Values:

*filespec*—a pathname designator.

*truename*—a physical pathname.

### Description:

**truename** tries to find the file indicated by *filespec* and returns its truename. If the *filespec designator* is an open stream, its associated file is used. If *filespec* is a stream, **truename** can be used whether the stream is open or closed. It is permissible for **truename** to return more specific information after the stream is closed than when the stream was open. If *filespec* is a pathname it represents the name used to open the file. This may be, but is not required to be, the actual name of the file.

### Examples:

```
; ; An example involving version numbers. Note that the precise nature of
; ; the truename is implementation-dependent while the file is still open.
(with-open-file (stream ">vistor>test.text.newest")
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(values (pathname stream)
       (truename stream)))
=> #P"S:>vistor>test.text.newest", #P"S:>vistor>test.text.1"
OR=> #P"S:>vistor>test.text.newest", #P"S:>vistor>test.text.newest"
OR=> #P"S:>vistor>test.text.newest", #P"S:>vistor>_temp_.temp_.1"

;; In this case, the file is closed when the truename is tried, so the
;; truename information is reliable.
(with-open-file (stream ">vistor>test.text.newest")
  (close stream)
  (values (pathname stream)
          (truename stream)))
=> #P"S:>vistor>test.text.newest", #P"S:>vistor>test.text.1"

;; An example involving TOP-20's implementation-dependent concept
;; of logical devices -- in this case, "DOC:" is shorthand for
;; "PS:<DOCUMENTATION>" ...
(with-open-file (stream "CMUC::DOC:DUMPER.HLP")
  (values (pathname stream)
          (truename stream)))
=> #P"CMUC::DOC:DUMPER.HLP", #P"CMUC::PS:<DOCUMENTATION>DUMPER.HLP.13"
```

**Affected By:** None.

### Exceptional Situations:

An error of **type file-error** is signaled if an appropriate *file* cannot be located within the *file system* for the given *filespec*, or if the *file system* cannot perform the requested operation.

An error of **type file-error** is signaled if *pathname* is *wild*.

### See Also:

[pathname, logical-pathname, Section 20.1 \(File System Concepts\), Section 19.1.2 \(Pathnames as Filenames\)](#)

### Notes:

**truename** may be used to account for any *filename* translations performed by the *file system*.

## Function TYPE-ERROR-DATUM, TYPE-ERROR-EXPECTED-TYPE

### Syntax:

**type-error-datum** *condition => datum*

**type-error-expected-type** *condition => expected-type*

### Arguments and Values:

*condition*—a *condition* of type **type-error**.

*datum*—an *object*.

*expected-type*---a type specifier.

**Description:**

**type-error-datum** returns the offending datum in the situation represented by the *condition*.

**type-error-expected-type** returns the expected type of the offending datum in the situation represented by the *condition*.

**Examples:**

```
(defun fix-digits (condition)
  (check-type condition type-error)
  (let* ((digits '(zero one two three four
                  five six seven eight nine))
         (val (position (type-error-datum condition) digits)))
    (if (and val (subtypep 'fixnum (type-error-expected-type condition)))
        (store-value 7)))

(defun foo (x)
  (handler-bind ((type-error #'fix-digits))
    (check-type x number)
    (+ x 3)))

(foo 'seven)
=> 10
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**type-error, Section 9 (Conditions)**

**Notes:** None.

## **Function TYPE-OF**

**Syntax:**

**type-of object => typespec**

**Arguments and Values:**

*object*---an object.

*typespec*---a type specifier.

**Description:**

## CLHS: Declaration DYNAMIC-EXTENT

Returns a type specifier, typespec, for a type that has the object as an element. The typespec satisfies the following:

1. For any object that is an element of some built-in type:

a. the type returned is a recognizable subtype of that built-in type.

b. the type returned does not involve and, eq1, member, not, or, satisfies, or values.

2. For all objects, (typep object (type-of object)) returns true. Implicit in this is that type specifiers which are not valid for use with typep, such as the list form of the function type specifier, are never returned by type-of.

3. The type returned by type-of is always a recognizable subtype of the class returned by class-of. That is,  
(subtypep (type-of object) (class-of object)) => true, true

4. For objects of metaclass structure-class or standard-class, and for conditions, type-of returns the proper name of the class returned by class-of if it has a proper name, and otherwise returns the class itself. In particular, for objects created by the constructor function of a structure defined with defstruct without a :type option, type-of returns the structure name; and for objects created by make-condition, the typespec is the name of the condition type.

5. For each of the types short-float, single-float, double-float, or long-float of which the object is an element, the typespec is a recognizable subtype of that type.

### Examples:

```
(type-of 'a) => SYMBOL
(type-of '(1 . 2))
=> CONS
OR=> (CONS FIXNUM FIXNUM)
(type-of #c(0 1))
=> COMPLEX
OR=> (COMPLEX INTEGER)
(defstruct temp-struct x y z) => TEMP-STRUCT
(type-of (make-temp-struct)) => TEMP-STRUCT
(type-of "abc")
=> STRING
OR=> (STRING 3)
(subtypep (type-of "abc") 'string) => true, true
(type-of (expt 2 40))
=> BIGNUM
OR=> INTEGER
OR=> (INTEGER 1099511627776 1099511627776)
OR=> SYSTEM::TWO-WORD-BIGNUM
OR=> FIXNUM
(subtypep (type-of 112312) 'integer) => true, true
(defvar *foo* (make-array 5 :element-type t)) => *FOO*
(class-name (class-of *foo*)) => VECTOR
(type-of *foo*)
=> VECTOR
OR=> (VECTOR T 5)
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

array-element-type, class-of, defstruct, typecase, typep, Section 4.2 (Types)

#### Notes:

Implementors are encouraged to arrange for type-of to return a portable value.

## Function TRANSLATE-LOGICAL-PATHNAME

#### Syntax:

**translate-logical-pathname** *pathname &key => physical-pathname*

#### Arguments and Values:

*pathname*—a pathname designator, or a logical pathname namestring.

*physical-pathname*—a physical pathname.

#### Description:

Translates *pathname* to a physical pathname, which it returns.

If *pathname* is a stream, the stream can be either open or closed. **translate-logical-pathname** returns the same physical pathname after a file is closed as it did when the file was open. It is an error if *pathname* is a stream that is created with make-two-way-stream, make-echo-stream, make-broadcast-stream, make-concatenated-stream, make-string-input-stream, make-string-output-stream.

If *pathname* is a logical pathname namestring, the host portion of the logical pathname namestring and its following colon are required.

*Pathname* is first coerced to a pathname. If the coerced *pathname* is a physical pathname, it is returned. If the coerced *pathname* is a logical pathname, the first matching translation (according to pathname-match-p) of the logical pathname host is applied, as if by calling **translate-pathname**. If the result is a logical pathname, this process is repeated. When the result is finally a physical pathname, it is returned. If no translation matches, an error is signaled.

**translate-logical-pathname** might perform additional translations, typically to provide translation of file types to local naming conventions, to accomodate physical file systems with limited length names, or to deal with special character requirements such as translating hyphens to underscores or uppercase letters to lowercase. Any such additional translations are implementation-defined. Some implementations do no additional translations.

There are no specified keyword arguments for **translate-logical-pathname**, but implementations are permitted to extend it by adding keyword arguments.

#### Examples:

See logical-pathname-translations.

**Affected By:** None.

**Exceptional Situations:**

If *pathname* is incorrectly supplied, an error of *type type-error* is signaled.

If no translation matches, an error of *type file-error* is signaled.

**See Also:**

**logical-pathname**, **logical-pathname-translations**, **logical-pathname**, Section 20.1 (File System Concepts), Section 19.1.2 (Pathnames as Filenames)

**Notes:** None.

**Function TRANSLATE-PATHNAME****Syntax:**

**translate-pathname** *source from-wildcard to-wildcard &key*

=> *translated-pathname*

**Arguments and Values:**

*source*—a pathname designator.

*from-wildcard*—a pathname designator.

*to-wildcard*—a pathname designator.

*translated-pathname*—a pathname.

**Description:**

**translate-pathname** translates *source* (that matches *from-wildcard*) into a corresponding pathname that matches *to-wildcard*, and returns the corresponding pathname.

The resulting pathname is *to-wildcard* with each wildcard or missing field replaced by a portion of *source*. A "wildcard field" is a pathname component with a value of :wild, a :wild element of a list-valued directory component, or an implementation-defined portion of a component, such as the "\*" in the complex wildcard string "foo\*bar" that some implementations support. An implementation that adds other wildcard features, such as regular expressions, must define how **translate-pathname** extends to those features. A "missing field" is a pathname component with a value of nil.

The portion of *source* that is copied into the resulting pathname is implementation-defined. Typically it is determined by the user interface conventions of the file systems involved. Usually it is the portion of *source* that matches a wildcard field of *from-wildcard* that is in the same position as the wildcard or missing field of *to-wildcard*. If there is no wildcard field in *from-wildcard* at that position, then usually it is the entire corresponding pathname component of *source*, or in the case of a list-valued directory component, the entire corresponding list element.

## CLHS: Declaration DYNAMIC-EXTENT

During the copying of a portion of *source* into the resulting *pathname*, additional *implementation-defined* translations of *case* or file naming conventions might occur, especially when *from-wildcard* and *to-wildcard* are for different hosts.

It is valid for *source* to be a wild *pathname*; in general this will produce a wild result. It is valid for *from-wildcard* and/or *to-wildcard* to be non-wild *pathnames*.

There are no specified keyword arguments for **translate-pathname**, but implementations are permitted to extend it by adding keyword arguments.

**translate-pathname** maps customary case in *source* into customary case in the output *pathname*.

### Examples:

```
;; The results of the following five forms are all implementation-dependent.
;; The second item in particular is shown with multiple results just to
;; emphasize one of many particular variations which commonly occurs.
(pathname-name (translate-pathname "foobar" "foo*" "*baz")) => "barbaz"
(pathname-name (translate-pathname "foobar" "foo*" ""))
=> "foobar"
OR=> "bar"
(pathname-name (translate-pathname "foobar" "*" "foo*")) => "foofoobar"
(pathname-name (translate-pathname "bar" "*" "foo*")) => "foobar"
(pathname-name (translate-pathname "foobar" "foo*" "baz*")) => "bazbar"

(defun translate-logical-pathname-1 (pathname rules)
  (let ((rule (assoc pathname rules :test #'pathname-match-p)))
    (unless rule (error "No translation rule for ~A" pathname))
    (translate-pathname pathname (first rule) (second rule))))
(translate-logical-pathname-1 "FOO:CODE;BASIC.LISP"
                            '(("FOO:DOCUMENTATION;" "MY-UNIX:/doc/foo/")
                              ("FOO:CODE;" "MY-UNIX:/lib/foo/")
                              ("FOO:PATCHES;*;" "MY-UNIX:/lib/foo/patch/*/")))
=> #P"MY-UNIX:/lib/foo/basic.l"

;;This example assumes one particular set of wildcard conventions
;;Not all file systems will run this example exactly as written
(defun rename-files (from to)
  (dolist (file (directory from))
    (rename-file file (translate-pathname file from to))))
(rename-files "/usr/me/*.lisp" "/dev/her/*.l")
;Renames /usr/me/init.lisp to /dev/her/init.l
(rename-files "/usr/me/pcl*/*" "/sys/pcl*/*")
;Renames /usr/me/pcl-5-may/low.lisp to /sys/pcl/pcl-5-may/low.lisp
;In some file systems the result might be /sys/pcl/5-may/low.lisp
(rename-files "/usr/me/pcl*/*" "/sys/library/*/")
;Renames /usr/me/pcl-5-may/low.lisp to /sys/library/pcl-5-may/low.lisp
;In some file systems the result might be /sys/library/5-may/low.lisp
(rename-files "/usr/me/foo.bar" "/usr/me2/")
;Renames /usr/me/foo.bar to /usr/me2/foo.bar
(rename-files "/usr/joe/*-recipes.text" "/usr/jim/cookbook/joe's-*-rec.text")
;Renames /usr/joe/lamb-recipes.text to /usr/jim/cookbook/joe's-lamb-rec.text
;Renames /usr/joe/pork-recipes.text to /usr/jim/cookbook/joe's-pork-rec.text
;Renames /usr/joe/veg-recipes.text to /usr/jim/cookbook/joe's-veg-rec.text
```

**Affected By:** None.

**Exceptional Situations:**

If any of *source*, *from-wildcard*, or *to-wildcard* is not a *pathname*, a *string*, or a *stream associated with a file* an error of *type type-error* is signaled.

(*pathname-match-p source from-wildcard*) must be true or an error of *type error* is signaled.

**See Also:**

[namestring](#), [pathname-host](#), [pathname](#), [logical-pathname](#), [Section 20.1 \(File System Concepts\)](#), [Section 19.1.2 \(Pathnames as Filenames\)](#)

**Notes:**

The exact behavior of *translate-pathname* cannot be dictated by the Common Lisp language and must be allowed to vary, depending on the user interface conventions of the file systems involved.

The following is an implementation guideline. One file system performs this operation by examining each piece of the three *pathnames* in turn, where a piece is a *pathname* component or a *list* element of a structured component such as a hierarchical directory. Hierarchical directory elements in *from-wildcard* and *to-wildcard* are matched by whether they are wildcards, not by depth in the directory hierarchy. If the piece in *to-wildcard* is present and not wild, it is copied into the result. If the piece in *to-wildcard* is :wild or nil, the piece in *source* is copied into the result. Otherwise, the piece in *to-wildcard* might be a complex wildcard such as "foo\*bar" and the piece in *from-wildcard* should be wild; the portion of the piece in *source* that matches the wildcard portion of the piece in *from-wildcard* replaces the wildcard portion of the piece in *to-wildcard* and the value produced is used in the result.

**Function TREE-EQUAL****Syntax:**

**tree-equal** *tree-1 tree-2 &key test test-not => generalized-boolean*

**Arguments and Values:**

*tree-1*—a *tree*.

*tree-2*—a *tree*.

*test*—a *designator* for a *function* of two *arguments* that returns a *generalized boolean*.

*test-not*—a *designator* for a *function* of two *arguments* that returns a *generalized boolean*.

*generalized-boolean*—a *generalized boolean*.

**Description:**

**tree-equal** tests whether two trees are of the same shape and have the same leaves. **tree-equal** returns *true* if *tree-1* and *tree-2* are both *atoms* and *satisfy the test*, or if they are both *conses* and the *car* of *tree-1* is **tree-equal** to the *car* of *tree-2* and the *cdr* of *tree-1* is **tree-equal** to the *cdr* of *tree-2*. Otherwise, **tree-equal** returns *false*.

## CLHS: Declaration DYNAMIC-EXTENT

**tree-equal** recursively compares conses but not any other objects that have components.

The first argument to the :test or :test-not function is *tree-1* or a car or cdr of *tree-1*; the second argument is *tree-2* or a car or cdr of *tree-2*.

### Examples:

```
(setq tree1 '(1 (1 2))
      tree2 '(1 (1 2))) => (1 (1 2))
(tree-equal tree1 tree2) => true
(eql tree1 tree2) => false
(setq tree1 ('a ('b 'c))
      tree2 ('(a ('b 'c)))) => ('a ('b 'c))
=> ((QUOTE A) ((QUOTE B) (QUOTE C)))
(tree-equal tree1 tree2 :test 'eq) => true
```

**Side Effects:** None.

**Affected By:** None.

### Exceptional Situations:

The consequences are undefined if both *tree-1* and *tree-2* are circular.

### See Also:

[equal](#), [Section 3.6 \(Traversal Rules and Side Effects\)](#)

### Notes:

The :test-not parameter is deprecated.

## **Function TWO-WAY-STREAM-INPUT-STREAM, TWO-WAY-STREAM-OUTPUT-STREAM**

### Syntax:

**two-way-stream-input-stream** *two-way-stream* => *input-stream*

**two-way-stream-output-stream** *two-way-stream* => *output-stream*

### Arguments and Values:

*two-way-stream* --- a two-way stream.

*input-stream* --- an input stream.

*output-stream* --- an output stream.

### Description:

**two-way-stream-input-stream** returns the *stream* from which *two-way-stream* receives input.

**two-way-stream-output-stream** returns the *stream* to which *two-way-stream* sends output.

**Examples:** None.

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:** None.

**Notes:** None.

## Function TYPEP

**Syntax:**

**typep** *object type-specifier* &optional *environment* => *generalized-boolean*

**Arguments and Values:**

*object*—an *object*.

*type-specifier*—any *type specifier* except *values*, or a *type specifier* list whose first element is either *function* or *values*.

*environment*—an *environment object*. The default is *nil*, denoting the *null lexical environment* and the current *global environment*.

*generalized-boolean*—a *generalized boolean*.

**Description:**

Returns *true* if *object* is of the *type* specified by *type-specifier*; otherwise, returns *false*.

A *type-specifier* of the form (*satisfies fn*) is handled by applying the function *fn* to *object*.

(**typep** *object* ' (*array type-specifier*)), where *type-specifier* is not \*, returns *true* if and only if *object* is an *array* that could be the result of supplying *type-specifier* as the :element-type argument to **make-array**. (*array* \*) refers to all *arrays* regardless of element type, while (*array type-specifier*) refers only to those *arrays* that can result from giving *type-specifier* as the :element-type argument to **make-array**. A similar interpretation applies to (*simple-array type-specifier*) and (*vector type-specifier*). See [Section 15.1.2.1 \(Array Upgrading\)](#).

(**typep** *object* ' (*complex type-specifier*)) returns *true* for all *complex* numbers that can result from giving *numbers* of type *type-specifier* to the **function complex**, plus all other *complex* numbers of the same specialized representation. Both the real and the imaginary parts of any such *complex* number must satisfy:

## CLHS: Declaration DYNAMIC-EXTENT

```
(typep realpart 'type-specifier)
(typep imagpart 'type-specifier)
```

See the [function upgraded-complex-part-type](#).

### Examples:

```
(typep 12 'integer) => true
(typep (1+ most-positive-fixnum) 'fixnum) => false
(typep nil t) => true
(typep nil nil) => false
(typep 1 '(mod 2)) => true
(typep #c(1 1) '(complex (eql 1))) => true
;; To understand this next example, you might need to refer to
;; Section 12.1.5.3 \(Rule of Canonical Representation for Complex Rationals\).
(typep #c(0 0) '(complex (eql 0))) => false
```

Let Ax and Ay be two [type specifiers](#) that denote different [types](#), but for which

```
(upgraded-array-element-type 'Ax)
```

and

```
(upgraded-array-element-type 'Ay)
```

denote the same [type](#). Notice that

```
(typep (make-array 0 :element-type 'Ax) '(array Ax)) => true
(typep (make-array 0 :element-type 'Ay) '(array Ay)) => true
(typep (make-array 0 :element-type 'Ax) '(array Ay)) => true
(typep (make-array 0 :element-type 'Ay) '(array Ax)) => true
```

**Affected By:** None.

### Exceptional Situations:

An error of [type error](#) is signaled if [type-specifier](#) is [values](#), or a [type specifier](#) list whose first element is either [function](#) or [values](#).

The consequences are undefined if the [type-specifier](#) is not a [type specifier](#).

### See Also:

[type-of](#), [upgraded-array-element-type](#), [upgraded-complex-part-type](#), [Section 4.2.3 \(Type Specifiers\)](#)

### Notes:

[Implementations](#) are encouraged to recognize and optimize the case of `(typep x (the class y))`, since it does not involve any need for expansion of [deftype](#) information at runtime.

***Function UNBOUND-SLOT-INSTANCE*****Syntax:**

**unbound-slot-instance** *condition => instance*

**Arguments and Values:**

*condition*—a condition of type **unbound-slot**.

*instance*—an object.

**Description:**

Returns the instance which had the unbound slot in the situation represented by the *condition*.

**Examples:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**cell-error-name**, **unbound-slot**, Section 9.1 (Condition System Concepts)

**Notes:** None.

***Function UNEXPORT*****Syntax:**

**unexport** *symbols &optional package => t*

**Arguments and Values:**

*symbols*—a designator for a list of symbols.

*package*—a package designator. The default is the current package.

**Description:**

**unexport** reverts external *symbols* in *package* to internal status; it undoes the effect of **export**.

**unexport** works only on *symbols present* in *package*, switching them back to internal status. If **unexport** is given a symbol that is already accessible as an internal symbol in *package*, it does nothing.

**Examples:**

```
(in-package "COMMON-LISP-USER") => #<PACKAGE "COMMON-LISP-USER">
(export (intern "CONTRABAND" (make-package 'temp)) 'temp) => T
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(find-symbol "CONTRABAND") => NIL, NIL
(use-package 'temp) => T
(find-symbol "CONTRABAND") => CONTRABAND, :INHERITED
(unexport 'contraband 'temp) => T
(find-symbol "CONTRABAND") => NIL, NIL
```

### Side Effects:

Package system is modified.

### Affected By:

Current state of the package system.

### Exceptional Situations:

If **unexport** is given a symbol not accessible in package at all, an error of type package-error is signaled.

The consequences are undefined if package is the KEYWORD package or the COMMON-LISP package.

### See Also:

**export**, Section 11.1 (Package Concepts)

**Notes:** None.

## Function UNINTERN

### Syntax:

**unintern** *symbol* &*optional package* => *generalized-boolean*

### Arguments and Values:

*symbol*—a symbol.

*package*—a package designator. The default is the current package.

*generalized-boolean*—a generalized boolean.

### Description:

**unintern** removes *symbol* from *package*. If *symbol* is present in *package*, it is removed from *package* and also from *package's shadowing symbols list* if it is present there. If *package* is the home package for *symbol*, *symbol* is made to have no home package. *Symbol* may continue to be accessible in *package* by inheritance.

Use of **unintern** can result in a symbol that has no recorded home package, but that in fact is accessible in some package. Common Lisp does not check for this pathological case, and such symbols are always printed preceded by #::.

**unintern** returns true if it removes *symbol*, and nil otherwise.

**Examples:**

```
(in-package "COMMON-LISP-USER") => #<PACKAGE "COMMON-LISP-USER">
(setq temps-unpack (intern "UNPACK" (make-package 'temp))) => TEMP::UNPACK
(unintern temps-unpack 'temp) => T
(find-symbol "UNPACK" 'temp) => NIL, NIL
temps-unpack => #:UNPACK
```

**Side Effects:**

**unintern** changes the state of the package system in such a way that the consistency rules do not hold across the change.

**Affected By:**

Current state of the package system.

**Exceptional Situations:**

Giving a shadowing symbol to **unintern** can uncover a name conflict that had previously been resolved by the shadowing. If package A uses packages B and C, A contains a shadowing symbol x, and B and C each contain external symbols named x, then removing the shadowing symbol x from A will reveal a name conflict between b:x and c:x if those two *symbols* are distinct. In this case **unintern** will signal an error.

**See Also:**

**Notes:** None.

**Function UNION, NUNION****Syntax:**

**union** *list-1* *list-2* &**key** *key test test-not* => *result-list*

**nunion** *list-1* *list-2* &**key** *key test test-not* => *result-list*

**Arguments and Values:**

*list-1*—[a proper list](#).

*list-2*—[a proper list](#).

*test*—[a designator for a function of two arguments that returns a generalized boolean](#).

*test-not*—[a designator for a function of two arguments that returns a generalized boolean](#).

*key*—[a designator for a function of one argument, or nil](#).

*result-list*—[a list](#).

**Description:**

**union** and **nunion** return a *list* that contains every element that occurs in either *list-1* or *list-2*.

For all possible ordered pairs consisting of one element from *list-1* and one element from *list-2*, **:test** or **:test-not** is used to determine whether they *satisfy the test*. The first argument to the **:test** or **:test-not** function is the part of the element of *list-1* extracted by the **:key** function (if supplied); the second argument is the part of the element of *list-2* extracted by the **:key** function (if supplied).

The argument to the **:key** function is an element of *list-1* or *list-2*; the return value is part of the supplied element. If **:key** is not supplied or **nil**, the element of *list-1* or *list-2* itself is supplied to the **:test** or **:test-not** function.

For every matching pair, one of the two elements of the pair will be in the result. Any element from either *list-1* or *list-2* that matches no element of the other will appear in the result.

If there is a duplication between *list-1* and *list-2*, only one of the duplicate instances will be in the result. If either *list-1* or *list-2* has duplicate entries within it, the redundant entries might or might not appear in the result.

The order of elements in the result do not have to reflect the ordering of *list-1* or *list-2* in any way. The result *list* may be **eq** to either *list-1* or *list-2* if appropriate.

**Examples:**

```
(union '(a b c) '(f a d))
=> (A B C F D)
OR=> (B C F A D)
OR=> (D F A B C)
(union '((x 5) (y 6)) '((z 2) (x 4)) :key #'car)
=> ((X 5) (Y 6) (Z 2))
OR=> ((X 4) (Y 6) (Z 2))

(setq lst1 (list 1 2 '(1 2) "a" "b")
      lst2 (list 2 3 '(2 3) "B" "C"))
=> (2 3 (2 3) "B" "C")
(nunion lst1 lst2)
=> (1 (1 2) "a" "b" 2 3 (2 3) "B" "C")
OR=> (1 2 (1 2) "a" "b" "C" "B" (2 3) 3)
```

**Side Effects:**

**nunion** is permitted to modify any part, *car* or *cdr*, of the *list structure* of *list-1* or *list-2*.

**Affected By:** None.

**Exceptional Situations:**

Should be prepared to signal an error of **type type-error** if *list-1* and *list-2* are not *proper lists*.

**See Also:**

**intersection**, [Section 3.2.1 \(Compiler Terminology\)](#), [Section 3.6 \(Traversal Rules and Side Effects\)](#)

**Notes:**

The `:test-not` parameter is deprecated.

Since the **nunion** side effect is not required, it should not be used in for-effect-only positions in portable code.

## **Function UNREAD-CHAR**

**Syntax:**

**unread-char** *character* &*optional input-stream => nil*

**Arguments and Values:**

*character*—a *character*; must be the last *character* that was read from *input-stream*.

*input-stream*—an *input stream designator*. The default is *standard input*.

**Description:**

**unread-char** places *character* back onto the front of *input-stream* so that it will again be the next character in *input-stream*.

When *input-stream* is an *echo stream*, no attempt is made to undo any echoing of the character that might already have been done on *input-stream*. However, characters placed on *input-stream* by **unread-char** are marked in such a way as to inhibit later re-echo by **read-char**.

It is an error to invoke **unread-char** twice consecutively on the same *stream* without an intervening call to **read-char** (or some other input operation which implicitly reads characters) on that *stream*.

Invoking **peek-char** or **read-char** commits all previous characters. The consequences of invoking **unread-char** on any character preceding that which is returned by **peek-char** (including those passed over by **peek-char** that has a *non-nil peek-type*) are unspecified. In particular, the consequences of invoking **unread-char** after **peek-char** are unspecified.

**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
```

**Affected By:**

\*standard-input\*, \*terminal-io\*.

**Exceptional Situations:** None.

**See Also:**

[peek-char](#), [read-char](#), [Section 21.1 \(Stream Concepts\)](#)

**Notes:**

**unread-char** is intended to be an efficient mechanism for allowing the *Lisp reader* and other parsers to perform one-character lookahead in *input-stream*.

**Function UNUSE-PACKAGE****Syntax:**

**unuse-package** *packages-to-unuse* &optional *package => t*

**Arguments and Values:**

*packages-to-unuse*—a designator for a list of package designators.

*package*—a package designator. The default is the current package.

**Description:**

**unuse-package** causes *package* to cease inheriting all the external symbols of *packages-to-unuse*; **unuse-package** undoes the effects of **use-package**. The *packages-to-unuse* are removed from the use list of *package*.

Any symbols that have been *imported* into *package* continue to be present in *package*.

**Examples:**

```
(in-package "COMMON-LISP-USER") => #<PACKAGE "COMMON-LISP-USER">
(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 'common-lisp-user)) => #<PACKAGE "TEMP">
(unuse-package 'temp) => T
(find-symbol "SHOES") => NIL, NIL
```

**Side Effects:**

The use list of *package* is modified.

**Affected By:**

Current state of the package system.

**Exceptional Situations:** None.

**See Also:**

**use-package, package-use-list**

**Notes:** None.

**Standard Generic Function UPDATE-INSTANCE-FOR-REDEFINED-CLASS****Syntax:**

**update-instance-for-redefined-class** *instance added-slots discarded-slots property-list &rest initargs &key &allow-other-keys*

=> *result\**

**Method Signatures:**

**update-instance-for-redefined-class** (*instance standard-object*) *added-slots discarded-slots property-list &rest initargs*

**Arguments and Values:**

*instance*—an object.

*added-slots*—a list.

*discarded-slots*—a list.

*property-list*—a list.

*initargs*—an initialization argument list.

*result*—an object.

**Description:**

The generic function update-instance-for-redefined-class is not intended to be called by programmers. Programmers may write methods for it. The generic function update-instance-for-redefined-class is called by the mechanism activated by make-instances-obsolete.

The system-supplied primary method on update-instance-for-redefined-class checks the validity of *initargs* and signals an error if an *initarg* is supplied that is not declared as valid. This method then initializes slots with values according to the *initargs*, and initializes the newly *added-slots* with values according to their :*initform* forms. It does this by calling the generic function shared-initialize with the following arguments: the *instance*, a list of names of the newly *added-slots* to *instance*, and the *initargs* it received. Newly *added-slots* are those local slots for which no slot of the same name exists in the old version of the class.

When make-instances-obsolete is invoked or when a class has been redefined and an instance is being updated, a property-list is created that captures the slot names and values of all the discarded-slots with values in the original *instance*. The structure of the *instance* is transformed so that it conforms to the current class definition. The arguments to update-instance-for-redefined-class are this transformed *instance*, a list of *added-slots* to the *instance*, a list *discarded-slots* from the *instance*, and the property-list containing the

## CLHS: Declaration DYNAMIC-EXTENT

slot names and values for *slots* that were discarded and had values. Included in this list of discarded *slots* are *slots* that were local in the old *class* and are shared in the new *class*.

The value returned by **update-instance-for-redefined-class** is ignored.

### Examples:

```
(defclass position () ())

(defclass x-y-position (position)
  ((x :initform 0 :accessor position-x)
   (y :initform 0 :accessor position-y)))

;; It turns out polar coordinates are used more than Cartesian
;; coordinates, so the representation is altered and some new
;; accessor methods are added.

(defmethod update-instance-for-redefined-class :before
  ((pos x-y-position) added deleted plist &key)
  ; Transform the x-y coordinates to polar coordinates
  ; and store into the new slots.
  (let ((x (getf plist 'x))
        (y (getf plist 'y)))
    (setf (position-rho pos) (sqrt (+ (* x x) (* y y))))
      (position-theta pos) (atan y x)))

(defclass x-y-position (position)
  ((rho :initform 0 :accessor position-rho)
   (theta :initform 0 :accessor position-theta)))

;; All instances of the old x-y-position class will be updated
;; automatically.

;; The new representation is given the look and feel of the old one.

(defmethod position-x ((pos x-y-position))
  (with-slots (rho theta) pos (* rho (cos theta)))))

(defmethod (setf position-x) (new-x (pos x-y-position))
  (with-slots (rho theta) pos
    (let ((y (position-y pos)))
      (setq rho (sqrt (+ (* new-x new-x) (* y y))))
        theta (atan y new-x))
      new-x)))

(defmethod position-y ((pos x-y-position))
  (with-slots (rho theta) pos (* rho (sin theta)))))

(defmethod (setf position-y) (new-y (pos x-y-position))
  (with-slots (rho theta) pos
    (let ((x (position-x pos)))
      (setq rho (sqrt (+ (* x x) (* new-y new-y))))
        theta (atan new-y x))
      new-y)))
```

**Affected By:** None.

**Exceptional Situations:**

The system-supplied primary method on update-instance-for-redefined-class signals an error if an initarg is supplied that is not declared as valid.

**See Also:**

make-instances-obsolete, shared-initialize, Section 4.3.6 (Redefining Classes), Section 7.1.4 (Rules for Initialization Arguments), Section 7.1.2 (Declaring the Validity of Initialization Arguments)

**Notes:**

Initargs are declared as valid by using the `:initarg` option to defclass, or by defining methods for update-instance-for-redefined-class or shared-initialize. The keyword name of each keyword parameter specifier in the lambda list of any method defined on update-instance-for-redefined-class or shared-initialize is declared as a valid initarg name for all classes for which that method is applicable.

## Standard Generic Function UPDATE-INSTANCE-FOR-DIFFERENT-CLASS

**Syntax:**

update-instance-for-different-class *previous current &rest initargs &key &allow-other-keys => implementation-dependent*

**Method Signatures:**

update-instance-for-different-class (*previous standard-object*) (*current standard-object*) *&rest initargs*

**Arguments and Values:**

*previous*—a copy of the original instance.

*current*—the original instance (altered).

*initargs*—an initialization argument list.

**Description:**

The generic function update-instance-for-different-class is not intended to be called by programmers. Programmers may write methods for it. The function update-instance-for-different-class is called only by the function change-class.

The system-supplied primary method on update-instance-for-different-class checks the validity of initargs and signals an error if an initarg is supplied that is not declared as valid. This method then initializes slots with values according to the initargs, and initializes the newly added slots with values according to their `:initform` forms. It does this by calling the generic function shared-initialize with the following arguments: the instance (*current*), a list of names of the newly added slots, and the initargs it received. Newly added slots are those local slots for which no slot of the same name exists in the *previous* class.

Methods for update-instance-for-different-class can be defined to specify actions to be taken when an instance is updated. If only after methods for update-instance-for-different-class are defined, they will be run after the system-supplied primary method for initialization and therefore will not interfere with the default behavior of update-instance-for-different-class.

Methods on update-instance-for-different-class can be defined to initialize slots differently from change-class. The default behavior of change-class is described in Section 7.2 (Changing the Class of an Instance).

The arguments to update-instance-for-different-class are computed by change-class. When change-class is invoked on an instance, a copy of that instance is made; change-class then destructively alters the original instance. The first argument to update-instance-for-different-class, previous, is that copy; it holds the old slot values temporarily. This argument has dynamic extent within change-class; if it is referenced in any way once update-instance-for-different-class returns, the results are undefined. The second argument to update-instance-for-different-class, current, is the altered original instance. The intended use of previous is to extract old slot values by using slot-value or with-slots or by invoking a reader generic function, or to run other methods that were applicable to instances of the original class.

#### Examples:

See the example for the function change-class.

**Affected By:** None.

#### Exceptional Situations:

The system-supplied primary method on update-instance-for-different-class signals an error if an initialization argument is supplied that is not declared as valid.

#### See Also:

change-class, shared-initialize, Section 7.2 (Changing the Class of an Instance), Section 7.1.4 (Rules for Initialization Arguments), Section 7.1.2 (Declaring the Validity of Initialization Arguments)

#### Notes:

Initargs are declared as valid by using the :initarg option to defclass, or by defining methods for update-instance-for-different-class or shared-initialize. The keyword name of each keyword parameter specifier in the lambda list of any method defined on update-instance-for-different-class or shared-initialize is declared as a valid initarg name for all classes for which that method is applicable.

The value returned by update-instance-for-different-class is ignored by change-class.

## Function UPGRADED-ARRAY-ELEMENT-TYPE

#### Syntax:

**upgraded-array-element-type** *typespec* &*optional environment* => *upgraded-typespec*

#### Arguments and Values:

*typespec*—a type specifier.

*environment*—an environment object. The default is nil, denoting the null lexical environment and the current global environment.

*upgraded-typespec*—a type specifier.

#### Description:

Returns the element type of the most specialized array representation capable of holding items of the type denoted by *typespec*.

The *typespec* is a subtype of (and possibly type equivalent to) the *upgraded-typespec*.

If *typespec* is bit, the result is type equivalent to bit. If *typespec* is base-char, the result is type equivalent to base-char. If *typespec* is character, the result is type equivalent to character.

The purpose of upgraded-array-element-type is to reveal how an implementation does its *upgrading*.

The *environment* is used to expand any derived type specifiers that are mentioned in the *typespec*.

**Examples:** None.

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

array-element-type, make-array

**Notes:**

Except for storage allocation consequences and dealing correctly with the optional environment argument, upgraded-array-element-type could be defined as:

```
(defun upgraded-array-element-type (type &optional environment)
  (array-element-type (make-array 0 :element-type type)))
```

## Function UPGRADED-COMPLEX-PART-TYPE

#### Syntax:

**upgraded-complex-part-type** *typespec* &*optional environment* => *upgraded-typespec*

#### Arguments and Values:

*typespec*—a type specifier.

## CLHS: Declaration DYNAMIC-EXTENT

*environment*—an *environment object*. The default is **nil**, denoting the *null lexical environment* and the and current *global environment*.

*upgraded-typespec*—a *type specifier*.

### Description:

**upgraded-complex-part-type** returns the part type of the most specialized *complex* number representation that can hold parts of *type typespec*.

The *typespec* is a *subtype* of (and possibly *type equivalent* to) the *upgraded-typespec*.

The purpose of **upgraded-complex-part-type** is to reveal how an implementation does its *upgrading*.

**Examples:** None.

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

### See Also:

**complex** (*function* and *type*)

### Notes:

## Function UPPER-CASE-P, LOWER-CASE-P, BOTH-CASE-P

### Syntax:

**upper-case-p** *character* => *generalized-boolean*

**lower-case-p** *character* => *generalized-boolean*

**both-case-p** *character* => *generalized-boolean*

### Arguments and Values:

*character*—a *character*.

*generalized-boolean*—a *generalized boolean*.

### Description:

These functions test the case of a given *character*.

**upper-case-p** returns *true* if *character* is an *uppercase character*; otherwise, returns *false*.

**lower-case-p** returns *true* if *character* is a *lowercase character*; otherwise, returns *false*.

**both-case-p** returns *true* if *character* is a *character* with *case*; otherwise, returns *false*.

### Examples:

```
(upper-case-p #\A) => true
(upper-case-p #\a) => false
(both-case-p #\a) => true
(both-case-p #\5) => false
(lower-case-p #\5) => false
(upper-case-p #\5) => false
;; This next example presupposes an implementation
;; in which #\Bell is an implementation-defined character.
(lower-case-p #\Bell) => false
```

**Side Effects:** None.

**Affected By:** None.

### Exceptional Situations:

Should signal an error of *type type-error* if *character* is not a *character*.

### See Also:

**char-upcase**, **char-downcase**, [Section 13.1.4.3 \(Characters With Case\)](#), [Section 13.1.10 \(Documentation of Implementation-Defined Scripts\)](#)

**Notes:** None.

## Function USE-PACKAGE

### Syntax:

**use-package** *packages-to-use* &*optional package* => *t*

### Arguments and Values:

*packages-to-use*—a *designator* for a *list of package designators*. The KEYWORD package may not be supplied.

*package*—a *package designator*. The default is the *current package*. The *package* cannot be the KEYWORD package.

### Description:

**use-package** causes *package* to inherit all the *external symbols* of *packages-to-use*. The inherited *symbols* become *accessible* as *internal symbols* of *package*.

*Packages-to-use* are added to the *use list* of *package* if they are not there already. All *external symbols* in *packages-to-use* become *accessible* in *package* as *internal symbols*. **use-package** does not cause any new

## CLHS: Declaration DYNAMIC-EXTENT

symbols to be present in *package* but only makes them accessible by inheritance.

**use-package** checks for name conflicts between the newly imported symbols and those already accessible in *package*. A name conflict in **use-package** between two external symbols inherited by *package* from *packages-to-use* may be resolved in favor of either symbol by *importing* one of them into *package* and making it a shadowing symbol.

### Examples:

```
(export (intern "LAND-FILL" (make-package 'trash)) 'trash) => T
(find-symbol "LAND-FILL" (make-package 'temp)) => NIL, NIL
(package-use-list 'temp) => (#<PACKAGE "TEMP">)
(use-package 'trash 'temp) => T
(package-use-list 'temp) => (#<PACKAGE "TEMP"> #<PACKAGE "TRASH">)
(find-symbol "LAND-FILL" 'temp) => TRASH:LAND-FILL, :INHERITED
```

### Side Effects:

The use list of *package* may be modified.

**Affected By:** None.

**Exceptional Situations:** None.

### See Also:

**unuse-package**, **package-use-list**, [Section 11.1 \(Package Concepts\)](#)

### Notes:

It is permissible for a *package* P1 to use a *package* P2 even if P2 already uses P1. The using of packages is not transitive, so no problem results from the apparent circularity.

## Function USER-HOMEDIR-PATHNAME

### Syntax:

**user-homedir-pathname** &*optional host* => *pathname*

### Arguments and Values:

*host*—a string, a list of strings, or :unspecific.

*pathname*—a pathname, or nil.

### Description:

**user-homedir-pathname** determines the pathname that corresponds to the user's home directory on *host*. If *host* is not supplied, its value is implementation-dependent. For a description of :unspecific, see [Section 19.2.1 \(Pathname Components\)](#).

## CLHS: Declaration DYNAMIC-EXTENT

The definition of home directory is *implementation-dependent*, but defined in Common Lisp to mean the directory where the user keeps personal files such as initialization files and mail.

**user-homedir-pathname** returns a *pathname* without any name, type, or version component (those components are all **nil**) for the user's home directory on *host*.

If it is impossible to determine the user's home directory on *host*, then **nil** is returned.  
**user-homedir-pathname** never returns **nil** if *host* is not supplied.

### Examples:

```
(pathnamep (user-homedir-pathname)) => true
```

**Side Effects:** None.

### Affected By:

The host computer's file system, and the *implementation*.

**Exceptional Situations:** None.

**See Also:** None.

**Notes:** None.

## Function VALUES-LIST

### Syntax:

**values-list** *list* => *element*\*

### Arguments and Values:

*list*—a *list*.

*elements*—the *elements* of the *list*.

### Description:

Returns the *elements* of the *list* as *multiple values*[2].

### Examples:

```
(values-list nil) => <no values>
(values-list '(1)) => 1
(values-list '(1 2)) => 1, 2
(values-list '(1 2 3)) => 1, 2, 3
```

**Affected By:** None.

**Exceptional Situations:**

Should signal **type-error** if its argument is not a *proper list*.

#### See Also:

**multiple-value-bind**, **multiple-value-list**, **multiple-values-limit**, **values**

#### Notes:

```
(values-list list) == (apply #'values list)
```

(equal x (multiple-value-list (values-list x))) returns *true* for all *lists* *x*.

## Accessor VALUES

#### Syntax:

**values** &*rest object* => *object\**

```
(setf (values &rest place) new-values)
```

#### Arguments and Values:

*object*---an *object*.

*place*---a *place*.

*new-value*---an *object*.

#### Description:

**values** returns the *objects* as *multiple values*[2].

**setf** of **values** is used to store the *multiple values*[2] *new-values* into the *places*. See [Section 5.1.2.3 \(VALUES Forms as Places\)](#).

#### Examples:

```
(values) => <no values>
(values 1) => 1
(values 1 2) => 1, 2
(values 1 2 3) => 1, 2, 3
(values (values 1 2 3) 4 5) => 1, 2, 3, 4, 5
(defun polar (x y)
  (values (sqrt (+ (* x x) (* y y))) (atan y x))) => POLAR
(multiple-value-bind (r theta) (polar 3.0 4.0)
  (vector r theta))
=> #(5.0 0.927295)
```

Sometimes it is desirable to indicate explicitly that a function returns exactly one value. For example, the function

```
(defun foo (x y)
  (floor (+ x y)) => FOO
```

## CLHS: Declaration DYNAMIC-EXTENT

returns two values because **floor** returns two values. It may be that the second value makes no sense, or that for efficiency reasons it is desired not to compute the second value. **values** is the standard idiom for indicating that only one value is to be returned:

```
(defun foo (x y)
  (values (floor (+ x y) y))) => FOO
```

This works because **values** returns exactly one value for each of *args*; as for any function call, if any of *args* produces more than one value, all but the first are discarded.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[\*\*values-list\*\*](#), [\*\*multiple-value-bind\*\*](#), [\*\*multiple-values-limit\*\*](#), [Section 3.1 \(Evaluation\)](#)

**Notes:**

Since **values** is a *function*, not a *macro* or *special form*, it receives as *arguments* only the *primary values* of its *argument forms*.

## Function VECTOR-POP

**Syntax:**

**vector-pop** *vector* => *element*

**Arguments and Values:**

*vector*—a *vector* with a *fill pointer*.

*element*—an *object*.

**Description:**

Decreases the *fill pointer* of *vector* by one, and retrieves the *element* of *vector* that is designated by the new *fill pointer*.

**Examples:**

```
(vector-push (setq fable (list 'fable))
             (setq fa (make-array 8
                                   :fill-pointer 2
                                   :initial-element 'sisyphus))) => 2
(fill-pointer fa) => 3
(eq (vector-pop fa) fable) => true
(vector-pop fa) => SISYPHUS
(fill-pointer fa) => 1
```

**Side Effects:**

The *fill pointer* is decreased by one.

#### Affected By:

The value of the *fill pointer*.

#### Exceptional Situations:

An error of *type type-error* is signaled if *vector* does not have a *fill pointer*.

If the *fill pointer* is zero, **vector-pop** signals an error of *type error*.

#### See Also:

**vector-push**, **vector-push-extend**, **fill-pointer**

**Notes:** None.

## Function VECTOR-PUSH, VECTOR-PUSH-EXTEND

#### Syntax:

**vector-push** *new-element* *vector* => *new-index-p*

**vector-push-extend** *new-element* *vector* &*optional extension* => *new-index*

#### Arguments and Values:

*new-element*—*an object*.

*vector*—*a vector* with a *fill pointer*.

*extension*—*a positive integer*. The default is *implementation-dependent*.

*new-index-p*—*a valid array index* for *vector*, or **nil**.

*new-index*—*a valid array index* for *vector*.

#### Description:

**vector-push** and **vector-push-extend** store *new-element* in *vector*. **vector-push** attempts to store *new-element* in the element of *vector* designated by the *fill pointer*, and to increase the *fill pointer* by one. If the ( $\geq$  (fill-pointer *vector*) (array-dimension *vector* 0)), neither *vector* nor its *fill pointer* are affected. Otherwise, the store and increment take place and **vector-push** returns the former value of the *fill pointer* which is one less than the one it leaves in *vector*.

**vector-push-extend** is just like **vector-push** except that if the *fill pointer* gets too large, *vector* is extended using **adjust-array** so that it can contain more elements. *Extension* is the minimum number of elements to be added to *vector* if it must be extended.

## CLHS: Declaration DYNAMIC-EXTENT

**vector-push** and **vector-push-extend** return the index of *new-element* in *vector*. If ( $>=$  (fill-pointer *vector*) (array-dimension *vector* 0)), **vector-push** returns **nil**.

### Examples:

```
(vector-push (setq fable (list 'fable))
             (setq fa (make-array 8
                                   :fill-pointer 2
                                   :initial-element 'first-one))) => 2
(fill-pointer fa) => 3
(eq (aref fa 2) fable) => true
(vector-push-extend #\X
                    (setq aa
                          (make-array 5
                                     :element-type 'character
                                     :adjustable t
                                     :fill-pointer 3))) => 3
(fill-pointer aa) => 4
(vector-push-extend #\Y aa 4) => 4
(array-total-size aa) => at least 5
(vector-push-extend #\Z aa 4) => 5
(array-total-size aa) => 9 ;(or more)
```

### Affected By:

The value of the *fill pointer*.

How *vector* was created.

### Exceptional Situations:

An error of **type error** is signaled by **vector-push-extend** if it tries to extend *vector* and *vector* is not *actually adjustable*.

An error of **type error** is signaled if *vector* does not have a *fill pointer*.

### See Also:

**adjustable-array-p**, **fill-pointer**, **vector-pop**

**Notes:** None.

## Function VECTОР

### Syntax:

**vectorp** *object*  $\Rightarrow$  *generalized-boolean*

### Arguments and Values:

*object*—an *object*.

*generalized-boolean*—a *generalized boolean*.

**Description:**

Returns true if *object* is of type vector; otherwise, returns false.

**Examples:**

```
(vectorp "aaaaaaaa") => true
(vectorp (make-array 6 :fill-pointer t)) => true
(vectorp (make-array '(2 3 4))) => false
(vectorp #*11) => true
(vectorp #b11) => false
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:** None.

**Notes:**

```
(vectorp object) == (typep object 'vector)
```

**Function VECTOR****Syntax:**

**vector** &*rest objects* => *vector*

**Arguments and Values:**

*object*---an object.

*vector*---a vector of type (vector t \*).

**Description:**

Creates a fresh simple general vector whose size corresponds to the number of *objects*.

The vector is initialized to contain the *objects*.

**Examples:**

```
(arrayp (setq v (vector 1 2 'sirens))) => true
(vectorp v) => true
(simple-vector-p v) => true
(length v) => 3
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

### make-array

**Notes:**

vector is analogous to list.

```
(vector a1 a2 ... an)
== (make-array (list n) :element-type t
              :initial-contents
              (list a1 a2 ... an))
```

## **Function** **WARN**

**Syntax:**

**warn** *datum &rest arguments => nil*

**Arguments and Values:**

*datum, arguments*—*designators* for a *condition* of default type simple-warning.

**Description:**

*Signals* a *condition* of type warning. If the *condition* is not *handled*, reports the *condition* to *error output*.

The precise mechanism for warning is as follows:

### *The warning condition is signaled*

While the warning condition is being signaled, the muffle-warning restart is established for use by a handler. If invoked, this restart bypasses further action by warn, which in turn causes warn to immediately return nil.

### *If no handler for the warning condition is found*

If no handlers for the warning condition are found, or if all such handlers decline, then the *condition* is reported to *error output* by warn in an implementation-dependent format.

### *nil is returned*

The value returned by warn if it returns is nil.

**Examples:**

```
(defun foo (x)
  (let ((result (* x 2)))
    (if (not (typep result 'fixnum))
        (warn "You're using very big numbers."))
    result))
=> FOO

(foo 3)
=> 6
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(foo most-positive-fixnum)
>> Warning: You're using very big numbers.
=> 4294967294

(setq *break-on-signals* t)
=> T

(foo most-positive-fixnum)
>> Break: Caveat emptor.
>> To continue, type :CONTINUE followed by an option number.
>> 1: Return from Break.
>> 2: Abort to Lisp Toplevel.
>> Debug> :continue 1
>> Warning: You're using very big numbers.
=> 4294967294
```

### Side Effects:

A warning is issued. The debugger might be entered.

### Affected By:

Existing handler bindings.

\*break-on-signals\*, \*error-output\*.

### Exceptional Situations:

If *datum* is a condition and if the condition is not of type warning, or *arguments* is non-nil, an error of type type-error is signaled.

If *datum* is a condition type, the result of (apply #'make-condition datum arguments) must be of type warning or an error of type type-error is signaled.

### See Also:

\*break-on-signals\*, muffle-warning, signal

**Notes:** None.

## Function WILD-PATHNAME-P

### Syntax:

**wild-pathname-p** *pathname* &*optional field-key* => *generalized-boolean*

### Arguments and Values:

*pathname*—a pathname designator.

*Field-key*—one of :host, :device :directory, :name, :type, :version, or nil.

*generalized-boolean*—a generalized boolean.

**Description:**

**wild-pathname-p** tests *pathname* for the presence of wildcard components.

If *pathname* is a pathname (as returned by **pathname**) it represents the name used to open the file. This may be, but is not required to be, the actual name of the file.

If *field-key* is not supplied or nil, **wild-pathname-p** returns true if *pathname* has any wildcard components, nil if *pathname* has none. If *field-key* is non-nil, **wild-pathname-p** returns true if the indicated component of *pathname* is a wildcard, nil if the component is not a wildcard.

**Examples:**

```
;;;The following examples are not portable. They are written to run
;;;with particular file systems and particular wildcard conventions.
;;;Other implementations will behave differently. These examples are
;;;intended to be illustrative, not to be prescriptive.
```

```
(wild-pathname-p (make-pathname :name :wild)) => true
(wild-pathname-p (make-pathname :name :wild) :name) => true
(wild-pathname-p (make-pathname :name :wild) :type) => false
(wild-pathname-p (pathname "s:>foo>**>")) => true ;LispM
(wild-pathname-p (pathname :name "F*O")) => true ;Most places
```

**Affected By:** None.

**Exceptional Situations:**

If *pathname* is not a pathname, a string, or a stream associated with a file an error of type **type-error** is signaled.

**See Also:**

pathname, logical-pathname, Section 20.1 (File System Concepts), Section 19.1.2 (Pathnames as Filenames)

**Notes:**

Not all implementations support wildcards in all fields. See Section 19.2.2.2 (:WILD as a Component Value) and Section 19.2.2.3 (Restrictions on Wildcard Pathnames).

## Function WRITE-BYTE

**Syntax:**

**write-byte** *byte stream* => *byte*

**Arguments and Values:**

*byte*—an integer of the stream element type of *stream*.

*stream*—a binary output stream.

**Description:**

**write-byte** writes one byte, *byte*, to *stream*.

**Examples:**

```
(with-open-file (s "temp-bytes"
                    :direction :output
                    :element-type 'unsigned-byte)
  (write-byte 101 s)) => 101
```

**Side Effects:**

*stream* is modified.

**Affected By:**

The *element type* of the *stream*.

**Exceptional Situations:**

Should signal an error of *type type-error* if *stream* is not a *stream*. Should signal an error of *type error* if *stream* is not a *binary output stream*.

Might signal an error of *type type-error* if *byte* is not an *integer* of the *stream element type* of *stream*.

**See Also:**

**read-byte, write-char, write-sequence**

**Notes:** None.

**Function WRITE-CHAR****Syntax:**

**write-char** *character* &*optional output-stream* => *character*

**Arguments and Values:**

*character*—a *character*.

*output-stream* — an *output stream designator*. The default is *standard output*.

**Description:**

**write-char** outputs *character* to *output-stream*.

**Examples:**

```
(write-char #\a)
>> a
```

```
=> #\a
  (with-output-to-string (s)
    (write-char #\a s)
    (write-char #\Space s)
    (write-char #\b s))
=> "a b"
```

**Side Effects:**

The *output-stream* is modified.

**Affected By:**

\*standard-output\*, \*terminal-io\*.

**Exceptional Situations:** None.

**See Also:**

read-char, write-byte, write-sequence

**Notes:** None.

## **Function WRITE, PRIN1, PRINT, PPRINT, PRINC**

**Syntax:**

**write** *object &key array base case circle escape gensym length level lines miser-width pprint-dispatch pretty radix readably right-margin stream*

=> *object*

**prin1** *object &optional output-stream => object*

**princ** *object &optional output-stream => object*

**print** *object &optional output-stream => object*

**pprint** *object &optional output-stream => <no values>*

**Arguments and Values:**

*object*—an object.

*output-stream*—an output stream designator. The default is standard output.

*array*—a generalized boolean.

*base*---a radix.

*case*---a symbol of type (member :upcase :downcase :capitalize).

*circle*---a generalized boolean.

*escape*---a generalized boolean.

*gensym*---a generalized boolean.

*length*---a non-negative integer, or nil.

*level*---a non-negative integer, or nil.

*lines*---a non-negative integer, or nil.

*miser-width*---a non-negative integer, or nil.

*pprint-dispatch*---a pprint dispatch table.

*pretty*---a generalized boolean.

*radix*---a generalized boolean.

*readably*---a generalized boolean.

*right-margin*---a non-negative integer, or nil.

*stream*---an output stream designator. The default is standard output.

#### Description:

**write**, **prin1**, **princ**, **print**, and **pprint** write the printed representation of *object* to *output-stream*.

**write** is the general entry point to the Lisp printer. For each explicitly supplied keyword parameter named in the next figure, the corresponding printer control variable is dynamically bound to its value while printing goes on; for each keyword parameter in the next figure that is not explicitly supplied, the value of the corresponding printer control variable is the same as it was at the time **write** was invoked. Once the appropriate bindings are *established*, the *object* is output by the Lisp printer.

Parameter	Corresponding Dynamic Variable
array	<u>*print-array*</u>
base	<u>*print-base*</u>
case	<u>*print-case*</u>
circle	<u>*print-circle*</u>
escape	<u>*print-escape*</u>
gensym	<u>*print-gensym*</u>
length	<u>*print-length*</u>
level	<u>*print-level*</u>
lines	<u>*print-lines*</u>
miser-width	<u>*print-miser-width*</u>
pprint-dispatch	<u>*print-pprint-dispatch*</u>
pretty	<u>*print-pretty*</u>

radix	<u>*print-radix*</u>
readably	<u>*print-readably*</u>
right-margin	<u>*print-right-margin*</u>

**Figure 22–7.** Argument correspondences for the WRITE function.

**prin1**, **princ**, **print**, and **pprint** implicitly *bind* certain print parameters to particular values. The remaining parameter values are taken from \*print-array\*, \*print-base\*, \*print-case\*, \*print-circle\*, \*print-escape\*, \*print-gensym\*, \*print-length\*, \*print-level\*, \*print-lines\*, \*print-miser-width\*, \*print-pprint-dispatch\*, \*print-pretty\*, \*print-radix\*, and \*print-right-margin\*.

**prin1** produces output suitable for input to **read**. It binds \*print-escape\* to *true*.

**princ** is just like **prin1** except that the output has no *escape characters*. It binds \*print-escape\* to *false* and \*print-readably\* to *false*. The general rule is that output from **princ** is intended to look good to people, while output from **prin1** is intended to be acceptable to **read**.

**print** is just like **prin1** except that the printed representation of *object* is preceded by a newline and followed by a space.

**pprint** is just like **print** except that the trailing space is omitted and *object* is printed with the \*print-pretty\* flag *non-nil* to produce pretty output.

*Output-stream* specifies the *stream* to which output is to be sent.

#### Affected By:

\*standard-output\*, \*terminal-io\*, \*print-escape\*, \*print-radix\*, \*print-base\*, \*print-circle\*, \*print-pretty\*, \*print-level\*, \*print-length\*, \*print-case\*, \*print-gensym\*, \*print-array\*, \*read-default-float-format\*.

**Exceptional Situations:** None.

#### See Also:

**readable-case**, Section 22.3.4 (FORMAT Printer Operations)

#### Notes:

The *functions* **prin1** and **print** do not bind \*print-readably\*.

```
(prin1 object output-stream)
== (write object :stream output-stream :escape t)

(princ object output-stream)
== (write object stream output-stream :escape nil :readably nil)

(print object output-stream)
== (progn (terpri output-stream)
          (write object :stream output-stream
                :escape t)
          (write-char #\space output-stream))
```

```
(pprint object output-stream)
== (write object :stream output-stream :escape t :pretty t)
```

## Function WRITE-SEQUENCE

### Syntax:

**write-sequence** *sequence stream &key start end => sequence*

*sequence*—a sequence.

*stream*—an output stream.

*start, end*—bounding index designators of *sequence*. The defaults for *start* and *end* are 0 and nil, respectively.

### Description:

**write-sequence** writes the elements of the subsequence of *sequence* bounded by *start* and *end* to *stream*.

### Examples:

```
(write-sequence "bookworms" *standard-output* :end 4)
>> book
=> "bookworms"
```

### Side Effects:

Modifies *stream*.

**Affected By:** None.

### Exceptional Situations:

Should be prepared to signal an error of type type-error if *sequence* is not a proper sequence. Should signal an error of type type-error if *start* is not a non-negative integer. Should signal an error of type type-error if *end* is not a non-negative integer or nil.

Might signal an error of type type-error if an element of the bounded sequence is not a member of the stream element type of the *stream*.

### See Also:

### Notes:

**write-sequence** is identical in effect to iterating over the indicated subsequence and writing one element at a time to *stream*, but may be more efficient than the equivalent loop. An efficient implementation is more likely to exist for the case where the *sequence* is a vector with the same element type as the *stream*.

**Function WRITE-STRING, WRITE-LINE****Syntax:**

**write-string** *string &optional output-stream &key start end => string*

**write-line** *string &optional output-stream &key start end => string*

**Arguments and Values:**

*string*—a string.

*output-stream*—an output stream designator. The default is standard output.

*start, end*—bounding index designators of *string*. The defaults for *start* and *end* are 0 and nil, respectively.

**Description:**

**write-string** writes the characters of the subsequence of *string bounded* by *start* and *end* to *output-stream*. **write-line** does the same thing, but then outputs a newline afterwards.

**Examples:**

```
(prog1 (write-string "books" nil :end 4) (write-string "worms"))
>> bookworms
=> "books"
(progn (write-char #\*)
       (write-line "test12" *standard-output* :end 5)
       (write-line "*test2")
       (write-char #\*)
       nil)
>> *test1
>> *test2
>> *
=> NIL
```

**Side Effects:** None.

**Affected By:**

\*standard-output\*, \*terminal-io\*.

**Exceptional Situations:** None.

**See Also:**

read-line, write-char

**Notes:**

**write-line** and **write-string** return *string*, not the substring *bounded* by *start* and *end*.

(write-string string)

```

==  (dotimes (i (length string)
    (write-char (char string i)))

  (write-line string)
==  (progl (write-string string) (terpri))

```

## **Function WRITE-TO-STRING, PRIN1-TO-STRING, PRINC-TO-STRING**

### Syntax:

**write-to-string** *object &key array base case circle escape gensym length level lines miser-width pprint-dispatch pretty radix readably right-margin*

=> *string*

**prin1-to-string** *object => string*

**princ-to-string** *object => string*

### Arguments and Values:

*object*---an object.

*array*---a generalized boolean.

*base*---a radix.

*case*---a symbol of type (member :upcase :downcase :capitalize).

*circle*---a generalized boolean.

*escape*---a generalized boolean.

*gensym*---a generalized boolean.

*length*---a non-negative integer, or **nil**.

*level*---a non-negative integer, or **nil**.

*lines*---a non-negative integer, or **nil**.

*miser-width*---a non-negative integer, or **nil**.

*pprint-dispatch*---a pprint dispatch table.

*pretty*---a generalized boolean.

*radix*---a generalized boolean.

*readably*—a generalized boolean.

*right-margin*—a non-negative integer, or nil.

*string*—a string.

#### Description:

**write-to-string**, **prin1-to-string**, and **princ-to-string** are used to create a string consisting of the printed representation of *object*. *Object* is effectively printed as if by **write**, **prin1**, or **princ**, respectively, and the characters that would be output are made into a string.

**write-to-string** is the general output function. It has the ability to specify all the parameters applicable to the printing of *object*.

**prin1-to-string** acts like **write-to-string** with :escape t, that is, escape characters are written where appropriate.

**princ-to-string** acts like **write-to-string** with :escape nil :readably nil. Thus no escape characters are written.

All other keywords that would be specified to **write-to-string** are default values when **prin1-to-string** or **princ-to-string** is invoked.

The meanings and defaults for the keyword arguments to **write-to-string** are the same as those for **write**.

#### Examples:

```
(prin1-to-string "abc") => "\\\"abc\\\""
(princ-to-string "abc") => "abc"
```

**Side Effects:** None.

#### Affected By:

\*print-escape\*, \*print-radix\*, \*print-base\*, \*print-circle\*, \*print-pretty\*, \*print-level\*,  
\*print-length\*, \*print-case\*, \*print-gensym\*, \*print-array\*, \*read-default-float-format\*.

**Exceptional Situations:** None.

#### See Also:

**write**

#### Notes:

```
(write-to-string object {key argument}*)
== (with-output-to-string (#1=#:string-stream)
  (write object :stream #1# {key argument}*))  
  

(princ-to-string object)
== (with-output-to-string (string-stream)
```

```
(princ object string-stream)
(prin1-to-string object)
== (with-output-to-string (string-stream)
  (prin1 object string-stream))
```

## Function Y-OR-N-P, YES-OR-NO-P

### Syntax:

**y-or-n-p** &optional control &rest arguments => generalized-boolean

**yes-or-no-p** &optional control &rest arguments => generalized-boolean

### Arguments and Values:

*control*---a format control.

*arguments*---format arguments for *control*.

*generalized-boolean*---a generalized boolean.

### Description:

These functions ask a question and parse a response from the user. They return true if the answer is affirmative, or false if the answer is negative.

**y-or-n-p** is for asking the user a question whose answer is either "yes" or "no." It is intended that the reply require the user to answer a yes-or-no question with a single character. **yes-or-no-p** is also for asking the user a question whose answer is either "Yes" or "No." It is intended that the reply require the user to take more action than just a single keystroke, such as typing the full word yes or no followed by a newline.

**y-or-n-p** types out a message (if supplied), reads an answer in some implementation-dependent manner (intended to be short and simple, such as reading a single character such as Y or N). **yes-or-no-p** types out a message (if supplied), attracts the user's attention (for example, by ringing the terminal's bell), and reads an answer in some implementation-dependent manner (intended to be multiple characters, such as YES or NO).

If *format-control* is supplied and not nil, then a **fresh-line** operation is performed; then a message is printed as if *format-control* and *arguments* were given to **format**. In any case, **yes-or-no-p** and **y-or-n-p** will provide a prompt such as "(Y or N)" or "(Yes or No)" if appropriate.

All input and output are performed using query I/O.

### Examples:

```
(y-or-n-p "(t or nil) given by")
>> (t or nil) given by (Y or N) Y
=> true
(yes-or-no-p "a ~S message" 'frightening)
>> a FRIGHTENING message (Yes or No) no
=> false
(y-or-n-p "Produce listing file?")
```

## CLHS: Declaration DYNAMIC-EXTENT

```
>> Produce listing file?  
>> Please respond with Y or N. n  
=> false
```

### Side Effects:

Output to and input from query I/O will occur.

### Affected By:

\*query-io\*.

**Exceptional Situations:** None.

### See Also:

format

### Notes:

yes-or-no-p and yes-or-no-p do not add question marks to the end of the prompt string, so any desired question mark or other punctuation should be explicitly included in the text query.

## Function ZEROP

### Syntax:

**zerop** *number* => *generalized-boolean*

### Pronunciation:

[zee(,)roh(,)pee]

### Arguments and Values:

*number*—a *number*.

*generalized-boolean*—a *generalized boolean*.

### Description:

Returns true if *number* is zero (integer, float, or complex); otherwise, returns false.

Regardless of whether an implementation provides distinct representations for positive and negative floating-point zeros, (`(zerop -0.0)`) always returns true.

### Examples:

```
(zerop 0) => true  
(zerop 1) => false  
(zerop -0.0) => true  
(zerop 0/100) => true
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(zerop #c(0 0.0)) => true
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:**

Should signal an error of type type-error if *number* is not a number.

**See Also:** None.

**Notes:**

```
(zerop number) == (= number 0)
```

## Macro AND

**Syntax:**

**and** *form\** => *result\**

**Arguments and Values:**

*form*—a form.

*results*—the values resulting from the evaluation of the last *form*, or the symbols nil or t.

**Description:**

The macro **and** evaluates each *form* one at a time from left to right. As soon as any *form* evaluates to nil, **and** returns nil without evaluating the remaining *forms*. If all *forms* but the last evaluate to true values, **and** returns the results produced by evaluating the last *form*.

If no *forms* are supplied, (**and**) returns t.

**and** passes back multiple values from the last subform but not from subforms other than the last.

**Examples:**

```
(if (and (>= n 0)
          (< n (length a-simple-vector))
          (eq (elt a-simple-vector n) 'foo))
    (princ "Foo!"))
```

The above expression prints **Foo!** if element *n* of *a-simple-vector* is the symbol **foo**, provided also that *n* is indeed a valid index for *a-simple-vector*. Because **and** guarantees left-to-right testing of its parts, elt is not called if *n* is out of range.

```
(setq temp1 1 temp2 1 temp3 1) => 1
(and (incf temp1) (incf temp2) (incf temp3)) => 2
(and (eql 2 temp1) (eql 2 temp2) (eql 2 temp3)) => true
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(decf temp3) => 1
(and (decf temp1) (decf temp2) (eq temp3 'nil) (decf temp3)) => NIL
(and (eql temp1 temp2) (eql temp2 temp3)) => true
(and) => T
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[cond](#), [every](#), [if](#), [or](#), [when](#)

**Notes:**

```
(and form) == (let () form)
(and form1 form2 ...) == (when form1 (and form2 ...))
```

## Macro ASSERT

**Syntax:**

**assert** *test-form* [*(place\*)* [*datum-form argument-form\**]]

=> nil

**Arguments and Values:**

*test-form*—a form; always evaluated.

*place*—a place; evaluated if an error is signaled.

*datum-form*—a form that evaluates to a *datum*. Evaluated each time an error is to be signaled, or not at all if no error is to be signaled.

*argument-form*—a form that evaluates to an *argument*. Evaluated each time an error is to be signaled, or not at all if no error is to be signaled.

*datum, arguments*—designators for a condition of default type error. (These designators are the result of evaluating *datum-form* and each of the *argument-forms*.)

**Description:**

**assert** assures that *test-form* evaluates to true. If *test-form* evaluates to false, **assert** signals a correctable error (denoted by *datum* and *arguments*). Continuing from this error using the continue restart makes it possible for the user to alter the values of the *places* before **assert** evaluates *test-form* again. If the value of *test-form* is non-nil, **assert** returns nil.

The *places* are generalized references to data upon which *test-form* depends, whose values can be changed by the user in attempting to correct the error. Subforms of each *place* are only evaluated if an error is signaled, and might be re-evaluated if the error is re-signaled (after continuing without actually fixing the problem). The order of evaluation of the *places* is not specified; see Section 5.1.1.1 (Evaluation of Subforms to Places).

## CLHS: Declaration DYNAMIC-EXTENT

If a *place form* is supplied that produces more values than there are store variables, the extra values are ignored. If the supplied *form* produces fewer values than there are store variables, the missing values are set to **nil**.

### Examples:

```
(setq x (make-array '(3 5) :initial-element 3))
=> #2A((3 3 3 3 3) (3 3 3 3 3) (3 3 3 3 3))
(setq y (make-array '(3 5) :initial-element 7))
=> #2A((7 7 7 7 7) (7 7 7 7 7) (7 7 7 7 7))
(defun matrix-multiply (a b)
  (let ((*print-array* nil))
    (assert (and (= (array-rank a) (array-rank b) 2)
                 (= (array-dimension a 1) (array-dimension b 0)))
            (a b)
            "Cannot multiply ~S by ~S." a b)
    (really-matrix-multiply a b))) => MATRIX-MULTIPLY
(matrix-multiply x y)
>> Correctable error in MATRIX-MULTIPLY:
>> Cannot multiply #<ARRAY ...> by #<ARRAY ...>.
>> Restart options:
>> 1: You will be prompted for one or more new values.
>> 2: Top level.
>> Debug> :continue 1
>> Value for A: x
>> Value for B: (make-array '(5 3) :initial-element 6)
=> #2A((54 54 54 54 54)
        (54 54 54 54 54)
        (54 54 54 54 54)
        (54 54 54 54 54))

(defun double-safely (x) (assert (numberp x) (x)) (+ x x))
(double-safely 4)
=> 8

(double-safely t)
>> Correctable error in DOUBLE-SAFELY: The value of (NUMBERP X) must be non-NIL.
>> Restart options:
>> 1: You will be prompted for one or more new values.
>> 2: Top level.
>> Debug> :continue 1
>> Value for X: 7
=> 14
```

### Affected By:

#### **\*break-on-signals\***

The set of active *condition handlers*.

**Exceptional Situations:** None.

### See Also:

[check-type, error, Section 5.1 \(Generalized Reference\)](#)

**Notes:**

## Local Macro CALL-METHOD, MAKE-METHOD

**Syntax:**

**call-method** *method* &optional *next-method-list* => *result\**

**make-method** *form* => *method-object*

**Arguments and Values:**

*method*—a method object, or a list (see below); not evaluated.

*method-object*—a method object.

*next-method-list*—a list of method objects; not evaluated.

*results*—the values returned by the method invocation.

**Description:**

The macro **call-method** is used in method combination. It hides the implementation-dependent details of how methods are called. The macro **call-method** has lexical scope and can only be used within an effective method form.

Whether or not **call-method** is fbound in the global environment is implementation-dependent; however, the restrictions on redefinition and shadowing of **call-method** are the same as for symbols in the COMMON-LISP package which are fbound in the global environment. The consequences of attempting to use **call-method** outside of an effective method form are undefined.

The macro **call-method** invokes the specified method, supplying it with arguments and with definitions for **call-next-method** and for **next-method-p**. If the invocation of **call-method** is lexically inside of a **make-method**, the arguments are those that were supplied to that method. Otherwise the arguments are those that were supplied to the generic function. The definitions of **call-next-method** and **next-method-p** rely on the specified *next-method-list*.

If *method* is a list, the first element of the list must be the symbol **make-method** and the second element must be a form. Such a list specifies a method object whose method function has a body that is the given form.

*Next-method-list* can contain method objects or lists, the first element of which must be the symbol **make-method** and the second element of which must be a form.

Those are the only two places where **make-method** can be used. The form used with **make-method** is evaluated in the null lexical environment augmented with a local macro definition for **call-method** and with bindings named by symbols not accessible from the COMMON-LISP-USER package.

The **call-next-method** function available to *method* will call the first method in *next-method-list*. The **call-next-method** function available in that method, in turn, will call the second method in *next-method-list*, and so on, until the list of next-methods is exhausted.

## CLHS: Declaration DYNAMIC-EXTENT

If *next-method-list* is not supplied, the **call-next-method** function available to *method* signals an error of **type control-error** and the **next-method-p** function available to *method* returns **nil**.

**Examples:**

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**call-next-method, define-method-combination, next-method-p**

**Notes:** None.

## Macro CASE, CCASE, ECASE

**Syntax:**

**case** *keyform* {*normal-clause*}\* [*otherwise-clause*] => *result*\*

**ccase** *keyplace* {*normal-clause*}\* => *result*\*

**ecase** *keyform* {*normal-clause*}\* => *result*\*

*normal-clause*::= (*keys* *form*\*)

*otherwise-clause*::= ({*otherwise* | *t*} *form*\*)

*clause*::= *normal-clause* | *otherwise-clause*

**Arguments and Values:**

*keyform*---a **form**; evaluated to produce a *test-key*.

*keyplace*---a **form**; evaluated initially to produce a *test-key*. Possibly also used later as a **place** if no *keys* match.

*test-key*---an object produced by evaluating *keyform* or *keyplace*.

*keys*---a **designator** for a **list** of **objects**. In the case of **case**, the **symbols t** and **otherwise** may not be used as the **keys designator**. To refer to these **symbols** by themselves as *keys*, the designators (*t*) and (*otherwise*), respectively, must be used instead.

*forms*---an **implicit progn**.

*results*---the **values** returned by the *forms* in the matching *clause*.

**Description:**

These macros allow the conditional execution of a body of *forms* in a *clause* that is selected by matching the *test-key* on the basis of its identity.

The *keyform* or *keyplace* is *evaluated* to produce the *test-key*.

Each of the *normal-clauses* is then considered in turn. If the *test-key* is the same as any *key* for that *clause*, the *forms* in that *clause* are *evaluated* as an implicit progn, and the values it returns are returned as the value of the case, ccase, or ecase form.

These macros differ only in their *behavior* when no *normal-clause* matches; specifically:

case

If no *normal-clause* matches, and there is an *otherwise-clause*, then that *otherwise-clause* automatically matches; the *forms* in that *clause* are *evaluated* as an implicit progn, and the values it returns are returned as the value of the case.

If there is no *otherwise-clause*, case returns nil.

ccase

If no *normal-clause* matches, a correctable error of type type-type-error is signaled. The offending datum is the *test-key* and the expected type is type equivalent to (member *key1 key2 ...*). The store-value restart can be used to correct the error.

If the store-value restart is invoked, its argument becomes the new *test-key*, and is stored in *keyplace* as if by (setf *keyplace test-key*). Then ccase starts over, considering each *clause* anew.

The subforms of *keyplace* might be evaluated again if none of the cases holds.

ecase

If no *normal-clause* matches, a non-correctable error of type type-type-error is signaled. The offending datum is the *test-key* and the expected type is type equivalent to (member *key1 key2 ...*).

Note that in contrast with ccase, the caller of ecase may rely on the fact that ecase does not return if a *normal-clause* does not match.

**Examples:**

```
(dolist (k '(1 2 3 :four #\v () t 'other))
  (format t "~S "
    (case k ((1 2) 'clause1)
      (3 'clause2)
      (nil 'no-keys-so-never-seen)
      ((nil) 'nilslot)
      ((:four #\v) 'clause4)
      ((t) 'tslot)
      (otherwise 'others))))
>> CLAUSE1 CLAUSE1 CLAUSE2 CLAUSE4 CLAUSE4 NILSLOT TSLOT OTHERS
=> NIL
(defun add-em (x) (apply #'+ (mapcar #'decode x)))
=> ADD-EM
(defun decode (x)
  (ccase x
    ((i uno) 1)
    ((ii dos) 2))
```

```
((iii tres) 3)
((iv cuatro) 4)))
=> DECODE
(add-em '(uno iii)) => 4
(add-em '(uno iiiii))
>> Error: The value of X, IIII, is not I, UNO, II, DOS, III,
>>      TRES, IV, or CUATRO.
>> 1: Supply a value to use instead.
>> 2: Return to Lisp Toplevel.
>> Debug> :CONTINUE 1
>> Value to evaluate and use for X: 'IV
=> 5
```

**Side Effects:**

The debugger might be entered. If the [store-value restart](#) is invoked, the [value](#) of *keyplace* might be changed.

**Affected By:**

[ccase](#) and [ecase](#), since they might signal an error, are potentially affected by existing *handlers* and [\\*debug-io\\*](#).

**Exceptional Situations:**

[ccase](#) and [ecase](#) signal an error of [type type-error](#) if no *normal-clause* matches.

**See Also:**

[cond](#), [typecase](#), [setf](#), Section 5.1 (Generalized Reference)

**Notes:**

```
(case test-key
  {((key*) form*)}*)
===
(let ((#1=#:g0001 test-key))
  (cond {((member #1# '(key*)) form*)}))
```

The specific error message used by [ecase](#) and [ccase](#) can vary between implementations. In situations where control of the specific wording of the error message is important, it is better to use [case](#) with an *otherwise-clause* that explicitly signals an error with an appropriate message.

**Macro CHECK-TYPE****Syntax:**

[check-type](#) *place typespec [string]* => [nil](#)

**Arguments and Values:**

*place*—a *place*.

*typespec*—a *type specifier*.

*string*—a string; evaluated.

### Description:

**check-type** signals a *correctable error* of type **type-error** if the contents of *place* are not of the type *typespec*.

**check-type** can return only if the store-value restart is invoked, either explicitly from a handler or implicitly as one of the options offered by the debugger. If the store-value restart is invoked, **check-type** stores the new value that is the argument to the restart invocation (or that is prompted for interactively by the debugger) in *place* and starts over, checking the type of the new value and signaling another error if it is still not of the desired *type*.

The first time *place* is *evaluated*, it is *evaluated* by normal evaluation rules. It is later *evaluated* as a place if the type check fails and the store-value restart is used; see [Section 5.1.1.1 \(Evaluation of Subforms to Places\)](#).

*string* should be an English description of the type, starting with an indefinite article ("a" or "an"). If *string* is not supplied, it is computed automatically from *typespec*. The automatically generated message mentions *place*, its contents, and the desired type. An implementation may choose to generate a somewhat differently worded error message if it recognizes that *place* is of a particular form, such as one of the arguments to the function that called **check-type**. *string* is allowed because some applications of **check-type** may require a more specific description of what is wanted than can be generated automatically from *typespec*.

### Examples:

```
(setq aardvarks '(sam harry fred))
=> (SAM HARRY FRED)
  (check-type aardvarks (array * (3)))
>> Error: The value of AARDVARKS, (SAM HARRY FRED),
>>       is not a 3-long array.
>> To continue, type :CONTINUE followed by an option number:
>> 1: Specify a value to use instead.
>> 2: Return to Lisp Toplevel.
>> Debug> :CONTINUE 1
>> Use Value: #(SAM FRED HARRY)
=> NIL
  aardvarks
=> #<ARRAY-T-3 13571>
  (map 'list #'identity aardvarks)
=> (SAM FRED HARRY)
  (setq aardvark-count 'foo)
=> FOO
  (check-type aardvark-count (integer 0 *) "A positive integer")
>> Error: The value of AARDVARK-COUNT, FOO, is not a positive integer.
>> To continue, type :CONTINUE followed by an option number:
>> 1: Specify a value to use instead.
>> 2: Top level.
>> Debug> :CONTINUE 2

(defmacro define-adder (name amount)
  (check-type name (and symbol (not null)) "a name for an adder function")
  (check-type amount integer)
  ` (defun ,name (x) (+ x ,amount)))

(macroexpand '(define-adder add3 3))
```

## CLHS: Declaration DYNAMIC-EXTENT

```
=> (defun add3 (x) (+ x 3))

(macroexpand '(define-adder 7 7))
>> Error: The value of NAME, 7, is not a name for an adder function.
>> To continue, type :CONTINUE followed by an option number:
>> 1: Specify a value to use instead.
>> 2: Top level.
>> Debug> :Continue 1
>> Specify a value to use instead.
>> Type a form to be evaluated and used instead: 'ADD7
=> (defun add7 (x) (+ x 7))

(macroexpand '(define-adder add5 something))
>> Error: The value of AMOUNT, SOMETHING, is not an integer.
>> To continue, type :CONTINUE followed by an option number:
>> 1: Specify a value to use instead.
>> 2: Top level.
>> Debug> :Continue 1
>> Type a form to be evaluated and used instead: 5
=> (defun add5 (x) (+ x 5))
```

Control is transferred to a handler.

### Side Effects:

The debugger might be entered.

### Affected By:

#### **\*break-on-signals\***

The implementation.

**Exceptional Situations:** None.

### See Also:

### Notes:

```
(check-type place typespec)
== (assert (typep place 'typespec) (place)
           'type-error :datum place :expected-type 'typespec)
```

## Macro COND

### Syntax:

**cond {clause}\* => result\***

clause ::= (test-form form\*)

### Arguments and Values:

*test-form*—a form.

*forms*—an implicit progn.

*results*—the values of the *forms* in the first *clause* whose *test-form* yields true, or the primary value of the *test-form* if there are no *forms* in that *clause*, or else nil if no *test-form* yields true.

#### Description:

**cond** allows the execution of *forms* to be dependent on *test-form*.

*Test-forms* are evaluated one at a time in the order in which they are given in the argument list until a *test-form* is found that evaluates to true.

If there are no *forms* in that clause, the primary value of the *test-form* is returned by the **cond form**. Otherwise, the *forms* associated with this *test-form* are evaluated in order, left to right, as an implicit progn, and the values returned by the last *form* are returned by the **cond form**.

Once one *test-form* has yielded true, no additional *test-forms* are *evaluated*. If no *test-form* yields true, **nil** is returned.

#### Examples:

```
(defun select-options ()
  (cond ((= a 1) (setq a 2))
        ((= a 2) (setq a 3))
        ((and (= a 3) (floor a 2)))
        (t (floor a 3))) =>  SELECT-OPTIONS
  (setq a 1) => 1
  (select-options) => 2
  a => 2
  (select-options) => 3
  a => 3
  (select-options) => 1
  (setq a 5) => 5
  (select-options) => 1, 2
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

#### See Also:

if, case.

**Notes:** None.

## Macro DECLAIM

#### Syntax:

**declare declaration-specifier\* => implementation-dependent**

**Arguments and Values:**

*declaration-specifier*—a declaration specifier; not evaluated.

**Description:**

Establishes the declarations specified by the *declaration-specifiers*.

If a use of this macro appears as a top level form in a file being processed by the file compiler, the proclamations are also made at compile-time. As with other defining macros, it is unspecified whether or not the compile-time side-effects of a claim persist after the file has been *compiled*.

**Examples:**

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

declare, proclaim

**Notes:** None.

**Macro DEFCLASS****Syntax:**

**defclass** *class-name* (*{superclass-name}\**) (*{slot-specifier}\**) [*[class-option]*]

=> *new-class*

```

slot-specifier ::= slot-name | (slot-name [[slot-option]])
slot-name ::= symbol
slot-option ::= { :reader reader-function-name}* |
               { :writer writer-function-name}* |
               { :accessor reader-function-name}* |
               { :allocation allocation-type} |
               { :initarg initarg-name}* |
               { :initform form} |
               { :type type-specifier} |
               { :documentation string}
function-name ::= symbol | (setf symbol)
class-option ::= (:default-initargs . initarg-list) |
                (:documentation string) |
                (:metaclass class-name)

```

**Arguments and Values:**

*Class-name*—a non-nil symbol.

*Superclass-name*—a non-nil symbol.

*Slot-name*—a symbol. The *slot-name* argument is a symbol that is syntactically valid for use as a variable name.

*Reader-function-name*—a non-nil symbol. :reader can be supplied more than once for a given slot.

*Writer-function-name*—a generic function name. :writer can be supplied more than once for a given slot.

*Reader-function-name*—a non-nil symbol. :accessor can be supplied more than once for a given slot.

*Allocation-type*—(member :instance :class). :allocation can be supplied once at most for a given slot.

*Initarg-name*—a symbol. :initarg can be supplied more than once for a given slot.

*Form*—a form. :init-form can be supplied once at most for a given slot.

*Type-specifier*—a type specifier. :type can be supplied once at most for a given slot.

*Class-option*—refers to the class as a whole or to all class slots.

*Initarg-list*—a list of alternating initialization argument names and default initial value forms. :default-initargs can be supplied at most once.

*Class-name*—a non-nil symbol. :metaclass can be supplied once at most.

*new-class*—the new class object.

### Description:

The macro **defclass** defines a new named class. It returns the new class object as its result.

The syntax of **defclass** provides options for specifying initialization arguments for slots, for specifying default initialization values for slots, and for requesting that methods on specified generic functions be automatically generated for reading and writing the values of slots. No reader or writer functions are defined by default; their generation must be explicitly requested. However, slots can always be accessed using slot-value.

Defining a new class also causes a type of the same name to be defined. The predicate (*typep object class-name*) returns true if the class of the given object is the class named by *class-name* itself or a subclass of the class *class-name*. A class object can be used as a type specifier. Thus (*typep object class*) returns *true* if the class of the object is class itself or a subclass of class.

The *class-name* argument specifies the proper name of the new class. If a class with the same proper name already exists and that class is an instance of standard-class, and if the **defclass** form for the definition of the new class specifies a class of class standard-class, the existing class is redefined, and instances of it (and its subclasses) are updated to the new definition at the time that they are next accessed. For details, see Section 4.3.6 (Redefining Classes).

## CLHS: Declaration DYNAMIC-EXTENT

Each *superclass-name* argument specifies a direct *superclass* of the new *class*. If the *superclass* list is empty, then the *superclass* defaults depending on the *metaclass*, with **standard-object** being the default for **standard-class**.

The new *class* will inherit *slots* and *methods* from each of its direct *superclasses*, from their direct *superclasses*, and so on. For a discussion of how *slots* and *methods* are inherited, see [Section 4.3.4 \(Inheritance\)](#).

The following slot options are available:

- The :*reader* slot option specifies that an *unqualified method* is to be defined on the *generic function* named *reader-function-name* to read the value of the given *slot*.
- The :*writer* slot option specifies that an *unqualified method* is to be defined on the *generic function* named *writer-function-name* to write the value of the *slot*.
- The :*accessor* slot option specifies that an *unqualified method* is to be defined on the generic function named *reader-function-name* to read the value of the given *slot* and that an *unqualified method* is to be defined on the *generic function* named (*setf reader-function-name*) to be used with *setf* to modify the value of the *slot*.
- The :*allocation* slot option is used to specify where storage is to be allocated for the given *slot*. Storage for a *slot* can be located in each instance or in the *class object* itself. The value of the *allocation-type* argument can be either the keyword :*instance* or the keyword :*class*. If the :*allocation* slot option is not specified, the effect is the same as specifying :*allocation* :*instance*.
  - ◆ If *allocation-type* is :*instance*, a *local slot* of the name *slot-name* is allocated in each instance of the *class*.
  - ◆ If *allocation-type* is :*class*, a shared *slot* of the given name is allocated in the *class object* created by this **defclass** form. The value of the *slot* is shared by all *instances* of the *class*. If a class C1 defines such a *shared slot*, any subclass C2 of C1 will share this single *slot* unless the **defclass** form for C2 specifies a *slot* of the same *name* or there is a superclass of C2 that precedes C1 in the class precedence list of C2 and that defines a *slot* of the same *name*.
- The :*initform* slot option is used to provide a default initial value form to be used in the initialization of the *slot*. This *form* is evaluated every time it is used to initialize the *slot*. The lexical environment in which this *form* is evaluated is the lexical environment in which the **defclass** form was evaluated. Note that the lexical environment refers both to variables and to functions. For *local slots*, the dynamic environment is the dynamic environment in which **make-instance** is called; for shared *slots*, the dynamic environment is the dynamic environment in which the **defclass** form was evaluated. See [Section 7.1 \(Object Creation and Initialization\)](#).

No implementation is permitted to extend the syntax of **defclass** to allow (*slot-name form*) as an abbreviation for (*slot-name* :*initform form*).

- The :*initarg* slot option declares an initialization argument named *initarg-name* and specifies that this initialization argument initializes the given *slot*. If the initialization argument has a value in the call to **initialize-instance**, the value will be stored into the given *slot*, and the slot's :*initform* slot option, if any, is not evaluated. If none of the initialization arguments specified for a given *slot* has a value, the *slot* is initialized according to the :*initform* slot option, if specified.
- The :*type* slot option specifies that the contents of the *slot* will always be of the specified data type. It effectively declares the result type of the reader generic function when applied to an *object* of this *class*. The consequences of attempting to store in a *slot* a value that does not satisfy the type of the *slot* are undefined. The :*type* slot option is further discussed in [Section 7.5.3 \(Inheritance of Slots and Slot Options\)](#).

## CLHS: Declaration DYNAMIC-EXTENT

- The :documentation slot option provides a *documentation string* for the *slot*. :documentation can be supplied once at most for a given *slot*.

Each class option is an option that refers to the *class* as a whole. The following class options are available:

- The :default-initargs class option is followed by a list of alternating initialization argument *names* and default initial value forms. If any of these initialization arguments does not appear in the initialization argument list supplied to make-instance, the corresponding default initial value form is evaluated, and the initialization argument *name* and the *form*'s value are added to the end of the initialization argument list before the instance is created; see Section 7.1 (Object Creation and Initialization). The default initial value form is evaluated each time it is used. The lexical environment in which this *form* is evaluated is the lexical environment in which the **defclass** form was evaluated. The dynamic environment is the dynamic environment in which make-instance was called. If an initialization argument *name* appears more than once in a :default-initargs class option, an error is signaled.
- The :documentation class option causes a *documentation string* to be attached with the *class object*, and attached with kind *type* to the *class-name*. :documentation can be supplied once at most.
- The :metaclass class option is used to specify that instances of the *class* being defined are to have a different metaclass than the default provided by the system (the *class standard-class*).

Note the following rules of **defclass** for *standard classes*:

- It is not required that the *superclasses* of a *class* be defined before the **defclass** form for that *class* is evaluated.
- All the *superclasses* of a *class* must be defined before an *instance* of the *class* can be made.
- A *class* must be defined before it can be used as a parameter specializer in a **defmethod** form.

The object system can be extended to cover situations where these rules are not obeyed.

Some slot options are inherited by a *class* from its *superclasses*, and some can be shadowed or altered by providing a local slot description. No class options except :default-initargs are inherited. For a detailed description of how *slots* and slot options are inherited, see Section 7.5.3 (Inheritance of Slots and Slot Options).

The options to **defclass** can be extended. It is required that all implementations signal an error if they observe a class option or a slot option that is not implemented locally.

It is valid to specify more than one reader, writer, accessor, or initialization argument for a *slot*. No other slot option can appear more than once in a single slot description, or an error is signaled.

If no reader, writer, or accessor is specified for a *slot*, the *slot* can only be accessed by the *function slot-value*.

If a **defclass form** appears as a *top level form*, the *compiler* must make the *class name* be recognized as a valid *type name* in subsequent declarations (as for **deftype**) and be recognized as a valid *class name* for **defmethod parameter specializers** and for use as the :metaclass option of a subsequent **defclass**. The *compiler* must make the *class* definition available to be returned by **find-class** when its *environment argument* is a value received as the *environment parameter* of a *macro*.

**Examples:** None.

**Affected By:** None.

**Exceptional Situations:**

If there are any duplicate slot names, an error of *type program-error* is signaled.

If an initialization argument *name* appears more than once in :default-initargs class option, an error of *type program-error* is signaled.

If any of the following slot options appears more than once in a single slot description, an error of *type program-error* is signaled: :allocation, :initform, :type, :documentation.

It is required that all implementations signal an error of *type program-error* if they observe a class option or a slot option that is not implemented locally.

**See Also:**

[documentation](#), [initialize-instance](#), [make-instance](#), [slot-value](#), [Section 4.3 \(Classes\)](#), [Section 4.3.4 \(Inheritance\)](#), [Section 4.3.6 \(Redefining Classes\)](#), [Section 4.3.5 \(Determining the Class Precedence List\)](#), [Section 7.1 \(Object Creation and Initialization\)](#)

**Notes:** None.

## Macro DEFCONSTANT

**Syntax:**

**defconstant** *name initial-value [documentation] => name*

**Arguments and Values:**

*name*—*a symbol*; not evaluated.

*initial-value*—*a form*; evaluated.

*documentation*—*a string*; not evaluated.

**Description:**

**defconstant** causes the global variable named by *name* to be given a value that is the result of evaluating *initial-value*.

A constant defined by **defconstant** can be redefined with **defconstant**. However, the consequences are undefined if an attempt is made to assign a *value* to the *symbol* using another operator, or to assign it to a *different value* using a subsequent **defconstant**.

If *documentation* is supplied, it is attached to *name* as a *documentation string* of kind *variable*.

**defconstant** normally appears as a *top level form*, but it is meaningful for it to appear as a *non-top-level form*. However, the compile-time side effects described below only take place when **defconstant** appears as a *top level form*.

## CLHS: Declaration DYNAMIC-EXTENT

The consequences are undefined if there are any *bindings* of the variable named by *name* at the time **defconstant** is executed or if the value is not **eql** to the value of *initial-value*.

The consequences are undefined when constant *symbols* are rebound as either lexical or dynamic variables. In other words, a reference to a *symbol* declared with **defconstant** always refers to its global value.

The side effects of the execution of **defconstant** must be equivalent to at least the side effects of the execution of the following code:

```
(setf (symbol-value 'name) initial-value)
      (setf (documentation 'name 'variable) 'documentation)
```

If a **defconstant** form appears as a *top level form*, the *compiler* must recognize that *name* names a *constant variable*. An implementation may choose to evaluate the value-form at compile time, load time, or both. Therefore, users must ensure that the *initial-value* can be *evaluated* at compile time (regardless of whether or not references to *name* appear in the file) and that it always *evaluates* to the same value.

### 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) => true
```

**Affected By:** None.

**Exceptional Situations:** None.

### See Also:

[\*\*declare\*\*](#), [\*\*defparameter\*\*](#), [\*\*defvar\*\*](#), [\*\*documentation\*\*](#), [\*\*proclaim\*\*](#), [Section 3.1.2.1.1.3 \(Constant Variables\)](#), [Section 3.2 \(Compilation\)](#)

**Notes:** None.

## Macro DEFGENERIC

### Syntax:

**defgeneric** *function-name* *gf-lambda-list* [[*option* / {*method-description*}\*]]

=> *new-generic*

```
option ::= (:argument-precedence-order parameter-name+) |
          (declare gf-declaration+) |
          (:documentation gf-documentation) |
          (:method-combination method-combination method-combination-argument*) |
          (:generic-function-class generic-function-class) |
          (:method-class method-class)
```

```
method-description ::= (:method method-qualifier* specialized-lambda-list [[declaration* | docum
```

### Arguments and Values:

*function-name*—a function name.

*generic-function-class*—a non-nil symbol naming a class.

*gf-declaration*—an optimize declaration specifier; other declaration specifiers are not permitted.

*gf-documentation*—a string; not evaluated.

*gf-lambda-list*—a generic function lambda list.

*method-class*—a non-nil symbol naming a class.

*method-combination-argument*—an object.

*method-combination-name*—a symbol naming a method combination type.

*method-qualifiers, specialized-lambda-list, declarations, documentation, forms*—as per defmethod.

*new-generic*—the generic function object.

*parameter-name*—a symbol that names a required parameter in the lambda-list. (If the :argument-precedence-order option is specified, each required parameter in the lambda-list must be used exactly once as a parameter-name.)

### Description:

The macro defgeneric is used to define a generic function or to specify options and declarations that pertain to a generic function as a whole.

If *function-name* is a list it must be of the form (setf symbol). If (fboundp function-name) is false, a new generic function is created. If (fdefinition function-name) is a generic function, that generic function is modified. If *function-name* names an ordinary function, a macro, or a special operator, an error is signaled.

The effect of the defgeneric macro is as if the following three steps were performed: first, methods defined by previous defgeneric forms are removed; second, ensure-generic-function is called; and finally, methods specified by the current defgeneric form are added to the generic function.

Each *method-description* defines a method on the generic function. The lambda list of each method must be congruent with the lambda list specified by the gf-lambda-list option. If no method descriptions are specified and a generic function of the same name does not already exist, a generic function with no methods is created.

The gf-lambda-list argument of defgeneric specifies the shape of lambda lists for the methods on this generic function. All methods on the resulting generic function must have lambda lists that are congruent with this shape. If a defgeneric form is evaluated and some methods for that generic function have lambda lists that are not congruent with that given in the defgeneric form, an error is signaled. For further details on method congruence, see Section 7.6.4 (Congruent Lambda-lists for all Methods of a Generic Function).

The generic function passes to the method all the argument values passed to it, and only those; default values are not supported. Note that optional and keyword arguments in method definitions, however, can have default initial value forms and can use supplied-p parameters.

## CLHS: Declaration DYNAMIC-EXTENT

The following options are provided. Except as otherwise noted, a given option may occur only once.

- The :argument-precedence-order option is used to specify the order in which the required arguments in a call to the *generic function* are tested for specificity when selecting a particular *method*. Each required argument, as specified in the *gf-lambda-list* argument, must be included exactly once as a *parameter-name* so that the full and unambiguous precedence order is supplied. If this condition is not met, an error is signaled.
- The **declare** option is used to specify declarations that pertain to the *generic function*.

An *optimize declaration specifier* is allowed. It specifies whether method selection should be optimized for speed or space, but it has no effect on *methods*. To control how a *method* is optimized, an **optimize** declaration must be placed directly in the **defmethod form** or method description. The optimization qualities **speed** and **space** are the only qualities this standard requires, but an implementation can extend the object system to recognize other qualities. A simple implementation that has only one method selection technique and ignores *optimize declaration specifiers* is valid.

The **special**, **ftype**, **function**, **inline**, **notinline**, and **declaration** declarations are not permitted. Individual implementations can extend the **declare** option to support additional declarations. If an implementation notices a *declaration specifier* that it does not support and that has not been proclaimed as a non-standard *declaration identifier* name in a *declaration proclamation*, it should issue a warning.

The **declare** option may be specified more than once. The effect is the same as if the lists of *declaration specifiers* had been appended together into a single list and specified as a single **declare** option.

- The :documentation argument is a *documentation string* to be attached to the *generic function object*, and to be attached with kind **function** to the *function-name*.
- The :generic-function-class option may be used to specify that the *generic function* is to have a different *class* than the default provided by the system (the *class standard-generic-function*). The *class-name* argument is the name of a *class* that can be the *class* of a *generic function*. If *function-name* specifies an existing *generic function* that has a different value for the :generic-function-class argument and the new generic function *class* is compatible with the old, **change-class** is called to change the *class* of the *generic function*; otherwise an error is signaled.
- The :method-class option is used to specify that all *methods* on this *generic function* are to have a different *class* from the default provided by the system (the *class standard-method*). The *class-name* argument is the name of a *class* that is capable of being the *class* of a *method*.
- The :method-combination option is followed by a symbol that names a type of method combination. The arguments (if any) that follow that symbol depend on the type of method combination. Note that the standard method combination type does not support any arguments. However, all types of method combination defined by the short form of **define-method-combination** accept an optional argument named *order*, defaulting to :most-specific-first, where a value of :most-specific-last reverses the order of the primary *methods* without affecting the order of the auxiliary *methods*.

The *method-description* arguments define *methods* that will be associated with the *generic function*. The *method-qualifier* and *specialized-lambda-list* arguments in a method description are the same as for **defmethod**.

The *form* arguments specify the method body. The body of the *method* is enclosed in an *implicit block*. If *function-name* is a *symbol*, this block bears the same name as the *generic function*. If *function-name* is a *list*

of the form (`(setf symbol)`), the name of the block is *symbol*.

Implementations can extend **defgeneric** to include other options. It is required that an implementation signal an error if it observes an option that is not implemented locally.

**defgeneric** is not required to perform any compile-time side effects. In particular, the *methods* are not installed for invocation during compilation. An *implementation* may choose to store information about the *generic function* for the purposes of compile-time error-checking (such as checking the number of arguments on calls, or noting that a definition for the function name has been seen).

### Examples:

**Affected By:** None.

### Exceptional Situations:

If *function-name* names an *ordinary function*, a *macro*, or a *special operator*, an error of *type program-error* is signaled.

Each required argument, as specified in the *gf-lambda-list* argument, must be included exactly once as a *parameter-name*, or an error of *type program-error* is signaled.

The *lambda list* of each *method* specified by a *method-description* must be congruent with the *lambda list* specified by the *gf-lambda-list* option, or an error of *type error* is signaled.

If a **defgeneric** form is evaluated and some *methods* for that *generic function* have *lambda lists* that are not congruent with that given in the **defgeneric** form, an error of *type error* is signaled.

A given *option* may occur only once, or an error of *type program-error* is signaled.

If *function-name* specifies an existing *generic function* that has a different value for the *:generic-function-class* argument and the new generic function *class* is compatible with the old, **change-class** is called to change the *class* of the *generic function*; otherwise an error of *type error* is signaled.

Implementations can extend **defgeneric** to include other options. It is required that an implementation signal an error of *type program-error* if it observes an option that is not implemented locally.

### See Also:

**defmethod**, **documentation**, **ensure-generic-function**, **generic-function**, [Section 7.6.4 \(Congruent Lambda-lists for all Methods of a Generic Function\)](#)

**Notes:** None.

## Macro DEFINE-SYMBOL-MACRO

### Syntax:

**define-symbol-macro** *symbol expansion*

=> *symbol*

### Arguments and Values:

*symbol*—a symbol.

*expansion*—a form.

### Description:

Provides a mechanism for globally affecting the macro expansion of the indicated *symbol*.

Globally establishes an expansion function for the symbol macro named by *symbol*. The only guaranteed property of an expansion function for a symbol macro is that when it is applied to the form and the environment it returns the correct expansion. (In particular, it is implementation-dependent whether the expansion is conceptually stored in the expansion function, the environment, or both.)

Each global reference to *symbol* (i.e., not shadowed[2] by a binding for a variable or symbol macro named by the same symbol) is expanded by the normal macro expansion process; see [Section 3.1.2.1.1 \(Symbols as Forms\)](#). The expansion of a symbol macro is subject to further macro expansion in the same lexical environment as the symbol macro reference, exactly analogous to normal macros.

The consequences are unspecified if a special declaration is made for *symbol* while in the scope of this definition (i.e., when it is not shadowed[2] by a binding for a variable or symbol macro named by the same symbol).

Any use of setq to set the value of the *symbol* while in the scope of this definition is treated as if it were a setf. psetq of *symbol* is treated as if it were a psetf, and multiple-value-setq is treated as if it were a setf of values.

A binding for a symbol macro can be shadowed[2] by let or symbol-macrolet.

### Examples:

```
(defvar *things* (list 'alpha 'beta 'gamma)) => *THINGS*
(define-symbol-macro thing1 (first *things*)) => THING1
(define-symbol-macro thing2 (second *things*)) => THING2
(define-symbol-macro thing3 (third *things*)) => THING3

thing1 => ALPHA
(setq thing1 'ONE) => ONE
*things* => (ONE BETA GAMMA)
(multiple-value-setq (thing2 thing3) (values 'two 'three)) => TWO
thing3 => THREE
*things* => (ONE TWO THREE)

(list thing2 (let ((thing2 2)) thing2)) => (TWO 2)
```

**Affected By:** None.

### Exceptional Situations:

If *symbol* is already defined as a *global variable*, an error of *type program-error* is signaled.

**See Also:**

**symbol-macrolet, macroexpand**

**Notes:** None.

## Macro DEFINE-MODIFY-MACRO

**Syntax:**

**define-modify-macro** *name lambda-list function [documentation] => name*

**Arguments and Values:**

*name*—a *symbol*.

*lambda-list*—a *define-modify-macro lambda list*

*function*—a *symbol*.

*documentation*—a *string*; not evaluated.

**Description:**

**define-modify-macro** defines a *macro* named *name* to *read* and *write* a *place*.

The arguments to the new *macro* are a *place*, followed by the arguments that are supplied in *lambda-list*. *Macros* defined with **define-modify-macro** correctly pass the *environment parameter* to **get-setf-expansion**.

When the *macro* is invoked, *function* is applied to the old contents of the *place* and the *lambda-list* arguments to obtain the new value, and the *place* is updated to contain the result.

Except for the issue of avoiding multiple evaluation (see below), the expansion of a **define-modify-macro** is equivalent to the following:

```
(defmacro name (reference . lambda-list)
  documentation
  `(setf ,reference
        (function ,reference ,arg1 ,arg2 ...)))
```

where *arg1*, *arg2*, ..., are the parameters appearing in *lambda-list*; appropriate provision is made for a *rest parameter*.

The *subforms* of the macro calls defined by **define-modify-macro** are evaluated as specified in [Section 5.1.1.1 \(Evaluation of Subforms to Places\)](#).

*Documentation* is attached as a *documentation string* to *name* (as kind **function**) and to the *macro function*.

## CLHS: Declaration DYNAMIC-EXTENT

If a **define-modify-macro** form appears as a *top level form*, the *compiler* must store the *macro* definition at compile time, so that occurrences of the macro later on in the file can be expanded correctly.

### 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)
(define-modify-macro new-incf (&optional (delta 1)) +)
(define-modify-macro unionf (other-set &rest keywords) union)
```

### Side Effects:

A macro definition is assigned to *name*.

**Affected By:** None.

**Exceptional Situations:** None.

### See Also:

**defsetf, define-setf-expander, documentation, Section 3.4.11 (Syntactic Interaction of Documentation Strings and Declarations)**

**Notes:** None.

## Macro DEFINE-SETF-EXPANDER

### Syntax:

**define-setf-expander** *access-fn lambda-list [[declaration\* / documentation]] form\**

=> *access-fn*

### Arguments and Values:

*access-fn*—a *symbol* that *names* a *function* or *macro*.

*lambda-list*—*macro lambda list*.

*declaration*—a **declare expression**; not evaluated.

*documentation*—a *string*; not evaluated.

*forms*—an *implicit progn*.

### Description:

**define-setf-expander** specifies the means by which **setf** updates a *place* that is referenced by *access-fn*.

## CLHS: Declaration DYNAMIC-EXTENT

When setf is given a place that is specified in terms of access-fn and a new value for the place, it is expanded into a form that performs the appropriate update.

The lambda-list supports destructuring. See [Section 3.4.4 \(Macro Lambda Lists\)](#).

Documentation is attached to access-fn as a documentation string of kind setf.

Forms constitute the body of the setf expander definition and must compute the setf expansion for a call on setf that references the place by means of the given access-fn. The setf expander function is defined in the same lexical environment in which the define-setf-expander form appears. While forms are being executed, the variables in lambda-list are bound to parts of the place form. The body forms (but not the lambda-list) in a define-setf-expander form are implicitly enclosed in a block whose name is access-fn.

The evaluation of forms must result in the five values described in [Section 5.1.1.2 \(Setf Expansions\)](#).

If a define-setf-expander form appears as a top level form, the compiler must make the setf expander available so that it may be used to expand calls to setf later on in the file. Programmers must ensure that the forms can be evaluated at compile time if the access-fn is used in a place later in the same file. The compiler must make these setf expanders available to compile-time calls to get-setf-expansion when its environment argument is a value received as the environment parameter of a macro.

### Examples:

```
(defun lastguy (x) (car (last x))) => LASTGUY
(define-setf-expander lastguy (x &environment env)
  "Set the last element in a list to the given value."
  (multiple-value-bind (dummies vals newval setter getter)
      (get-setf-expansion x env)
    (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)) 'lastguy-symbol) => LASTGUY-SYMBOL
a => (A B C 3)
b => (7)
c => (1 2 3 (4 5 LASTGUY-SYMBOL))

;; Setf expander for the form (LDB bytespec int).
;; Recall that the int form must itself be suitable for SETF.
(define-setf-expander ldb (bytespec int &environment env)
  (multiple-value-bind (temps vals stores
                               store-form access-form)
      (get-setf-expansion int env);Get setf expansion for int.
    (let ((btemp (gensym))      ;Temp var for byte specifier.
          (store (gensym))     ;Temp var for byte to store.
          (stemp (first stores))) ;Temp var for int to store.
      (if (cdr stores) (error "Can't expand this."))
    ;; Return the setf expansion for LDB as five values.
    (values (cons btemp temps)      ;Temporary variables.
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(cons bytespec vals)      ;Value forms.  
(list store)            ;Store variables.  
`(let ((,stemp (dpb ,store ,btemp ,access-form)))  
  ,store-form  
  ,store)                ;Storing form.  
`(ldb ,btemp ,access-form) ;Accessing form.  
))))
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[setf](#), [defsetf](#), [documentation](#), [get-setf-expansion](#), [Section 3.4.11 \(Syntactic Interaction of Documentation Strings and Declarations\)](#)

**Notes:**

[define-setf-expander](#) differs from the long form of [defsetf](#) in that while the body is being executed the [variables](#) in [lambda-list](#) are bound to parts of the [place form](#), not to temporary variables that will be bound to the values of such parts. In addition, [define-setf-expander](#) does not have [defsetf](#)'s restriction that [access-fn](#) must be a [function](#) or a function-like [macro](#); an arbitrary [defmacro](#) destructuring pattern is permitted in [lambda-list](#).

## Macro DEFINE-METHOD-COMBINATION

**Syntax:**

**define-method-combination** *name* [[*short-form-option*]]

=> *name*

**define-method-combination** *name* *lambda-list* (*method-group-specifier*\* ) [(:*arguments-args-lambda-list*) [(:*generic-function generic-function-symbol*) ] [*declaration*\* / *documentation*]] *form*\*

=> *name*

```
short-form-option ::= :documentation documentation |  
                   :identity-with-one-argument identity-with-one-argument |  
                   :operator operator  
  
method-group-specifier ::= (name {qualifier-pattern+} | predicate) [[long-form-option]]  
  
long-form-option ::= :description description |  
                   :order order |  
                   :required required-p
```

**Arguments and Values:**

*args-lambda-list*—a [define-method-combination arguments lambda list](#).

*declaration*—a declare expression; not evaluated.

*description*—a format control.

*documentation*—a string; not evaluated.

*forms*—an implicit progn that must compute and return the form that specifies how the methods are combined, that is, the effective method.

*generic-function-symbol*—a symbol.

*identity-with-one-argument*—a generalized boolean.

*lambda-list*—ordinary lambda list.

*name*—a symbol. Non-keyword, non-nil symbols are usually used.

*operator*—an operator. *Name* and *operator* are often the same symbol. This is the default, but it is not required.

*order*—:most-specific-first or :most-specific-last; evaluated.

*predicate*—a symbol that names a function of one argument that returns a generalized boolean.

*qualifier-pattern*—a list, or the symbol \*.

*required-p*—a generalized boolean.

## Description:

The macro define-method-combination is used to define new types of method combination.

There are two forms of define-method-combination. The short form is a simple facility for the cases that are expected to be most commonly needed. The long form is more powerful but more verbose. It resembles defmacro in that the body is an expression, usually using backquote, that computes a form. Thus arbitrary control structures can be implemented. The long form also allows arbitrary processing of method qualifiers.

### Short Form

The short form syntax of define-method-combination is recognized when the second subform is a non-nil symbol or is not present. When the short form is used, *name* is defined as a type of method combination that produces a Lisp form (*operator method-call method-call ...*). The *operator* is a symbol that can be the name of a function, macro, or special operator. The *operator* can be supplied by a keyword option; it defaults to *name*.

Keyword options for the short form are the following:

- ◊ The :documentation option is used to document the method-combination type; see description of long form below.
- ◊ The :identity-with-one-argument option enables an optimization when its value is true (the default is false). If there is exactly one applicable method and it is a primary method, that method serves as the effective method and *operator* is not called. This optimization

## CLHS: Declaration DYNAMIC-EXTENT

avoids the need to create a new effective method and avoids the overhead of a *function* call. This option is designed to be used with operators such as *progn*, *and*, *+*, and *max*. ◊ The *:operator* option specifies the *name* of the operator. The *operator* argument is a *symbol* that can be the *name* of a *function*, *macro*, or *special form*.

These types of method combination require exactly one *qualifier* per method. An error is signaled if there are applicable methods with no *qualifiers* or with *qualifiers* that are not supported by the method combination type.

A method combination procedure defined in this way recognizes two roles for methods. A method whose one *qualifier* is the symbol naming this type of method combination is defined to be a primary method. At least one primary method must be applicable or an error is signaled. A method with *:around* as its one *qualifier* is an auxiliary method that behaves the same as an *around method* in standard method combination. The *function call-next-method* can only be used in *around methods*; it cannot be used in primary methods defined by the short form of the *define-method-combination* macro.

A method combination procedure defined in this way accepts an optional argument named *order*, which defaults to *:most-specific-first*. A value of *:most-specific-last* reverses the order of the primary methods without affecting the order of the auxiliary methods.

The short form automatically includes error checking and support for *around methods*.

For a discussion of built-in method combination types, see [Section 7.6.6.4 \(Built-in Method Combination Types\)](#).

### Long Form

The long form syntax of *define-method-combination* is recognized when the second *subform* is a list.

The *lambda-list* receives any arguments provided after the *name* of the method combination type in the *:method-combination* option to *defgeneric*.

A list of method group specifiers follows. Each specifier selects a subset of the applicable methods to play a particular role, either by matching their *qualifiers* against some patterns or by testing their *qualifiers* with a *predicate*. These method group specifiers define all method *qualifiers* that can be used with this type of method combination.

The *car* of each *method-group-specifier* is a *symbol* which *names* a *variable*. During the execution of the *forms* in the body of *define-method-combination*, this *variable* is bound to a list of the *methods* in the method group. The *methods* in this list occur in the order specified by the *:order* option.

If *qualifier-pattern* is a *symbol* it must be *\**. A method matches a *qualifier-pattern* if the method's list of *qualifiers* is *equal* to the *qualifier-pattern* (except that the symbol *\** in a *qualifier-pattern* matches anything). Thus a *qualifier-pattern* can be one of the following: the *empty list*, which matches *unqualified methods*; the symbol *\**, which matches all methods; a true list, which matches methods with the same number of *qualifiers* as the length of the list when each *qualifier* matches the corresponding list element; or a dotted list that ends in the symbol *\** (the *\** matches any number of additional *qualifiers*).

## CLHS: Declaration DYNAMIC-EXTENT

Each applicable method is tested against the *qualifier-patterns* and *predicates* in left-to-right order. As soon as a *qualifier-pattern* matches or a *predicate* returns true, the method becomes a member of the corresponding method group and no further tests are made. Thus if a method could be a member of more than one method group, it joins only the first such group. If a method group has more than one *qualifier-pattern*, a method need only satisfy one of the *qualifier-patterns* to be a member of the group.

The *name* of a *predicate* function can appear instead of *qualifier-patterns* in a method group specifier. The *predicate* is called for each method that has not been assigned to an earlier method group; it is called with one argument, the method's *qualifier list*. The *predicate* should return true if the method is to be a member of the method group. A *predicate* can be distinguished from a *qualifier-pattern* because it is a *symbol* other than *nil* or *\**.

If there is an applicable method that does not fall into any method group, the *function invalid-method-error* is called.

Method group specifiers can have keyword options following the *qualifier* patterns or predicate. Keyword options can be distinguished from additional *qualifier* patterns because they are neither lists nor the symbol *\**. The keyword options are as follows:

- ◊ The *:description* option is used to provide a description of the role of methods in the method group. Programming environment tools use (*apply #'format stream format-control (method-qualifiers method)*) to print this description, which is expected to be concise. This keyword option allows the description of a method *qualifier* to be defined in the same module that defines the meaning of the method *qualifier*. In most cases, *format-control* will not contain any *format* directives, but they are available for generality. If *:description* is not supplied, a default description is generated based on the variable name and the *qualifier* patterns and on whether this method group includes the *unqualified methods*.
- ◊ The *:order* option specifies the order of methods. The *order* argument is a *form* that evaluates to *:most-specific-first* or *:most-specific-last*. If it evaluates to any other value, an error is signaled. If *:order* is not supplied, it defaults to *:most-specific-first*.
- ◊ The *:required* option specifies whether at least one method in this method group is required. If its value is *true* and the method group is empty (that is, no applicable methods match the *qualifier* patterns or satisfy the predicate), an error is signaled. If *:required* is not supplied, it defaults to *nil*.

The use of method group specifiers provides a convenient syntax to select methods, to divide them among the possible roles, and to perform the necessary error checking. It is possible to perform further filtering of methods in the body *forms* by using normal list-processing operations and the functions *method-qualifiers* and *invalid-method-error*. It is permissible to use *setq* on the variables named in the method group specifiers and to bind additional variables. It is also possible to bypass the method group specifier mechanism and do everything in the body *forms*. This is accomplished by writing a single method group with *\** as its only *qualifier-pattern*; the variable is then bound to a *list* of all of the *applicable methods*, in most-specific-first order.

The body *forms* compute and return the *form* that specifies how the methods are combined, that is, the effective method. The effective method is evaluated in the *null lexical environment* augmented with a local macro definition for *call-method* and with bindings named by symbols not *accessible* from the COMMON-LISP-USER package. Given a method object in one of the *lists* produced by the method

## CLHS: Declaration DYNAMIC-EXTENT

group specifiers and a *list* of next methods, **call-method** will invoke the method such that **call-next-method** has available the next methods.

When an effective method has no effect other than to call a single method, some implementations employ an optimization that uses the single method directly as the effective method, thus avoiding the need to create a new effective method. This optimization is active when the effective method form consists entirely of an invocation of the **call-method** macro whose first *subform* is a method object and whose second *subform* is **nil** or unsupplied. Each **define-method-combination** body is responsible for stripping off redundant invocations of **progn**, **and**, **multiple-value-prog**, and the like, if this optimization is desired.

The list (`:arguments . lambda-list`) can appear before any declarations or *documentation string*. This form is useful when the method combination type performs some specific behavior as part of the combined method and that behavior needs access to the arguments to the *generic function*. Each parameter variable defined by *lambda-list* is bound to a *form* that can be inserted into the effective method. When this *form* is evaluated during execution of the effective method, its value is the corresponding argument to the *generic function*; the consequences of using such a *form* as the *place* in a *setf form* are undefined. Argument correspondence is computed by dividing the `:arguments lambda-list` and the *generic function lambda-list* into three sections: the *required parameters*, the *optional parameters*, and the *keyword* and *rest parameters*. The *arguments* supplied to the *generic function* for a particular *call* are also divided into three sections; the required *arguments* section contains as many *arguments* as the *generic function* has *required parameters*, the optional *arguments* section contains as many arguments as the *generic function* has *optional parameters*, and the keyword/rest *arguments* section contains the remaining arguments. Each *parameter* in the required and optional sections of the `:arguments lambda-list` accesses the argument at the same position in the corresponding section of the *arguments*. If the section of the `:arguments lambda-list` is shorter, extra *arguments* are ignored. If the section of the `:arguments lambda-list` is longer, excess *required parameters* are bound to forms that evaluate to **nil** and excess *optional parameters* are *bound* to their initforms. The *keyword parameters* and *rest parameters* in the `:arguments lambda-list` access the keyword/rest section of the *arguments*. If the `:arguments lambda-list` contains `&key`, it behaves as if it also contained `&allow-other-keys`.

In addition, `&whole var` can be placed first in the `:arguments lambda-list`. It causes *var* to be *bound* to a *form* that *evaluates* to a *list* of all of the *arguments* supplied to the *generic function*. This is different from `&rest` because it accesses all of the arguments, not just the keyword/rest *arguments*.

Erroneous conditions detected by the body should be reported with **method-combination-error** or **invalid-method-error**; these *functions* add any necessary contextual information to the error message and will signal the appropriate error.

The body *forms* are evaluated inside of the *bindings* created by the *lambda list* and method group specifiers. Declarations at the head of the body are positioned directly inside of *bindings* created by the *lambda list* and outside of the *bindings* of the method group variables. Thus method group variables cannot be declared in this way. **locally** may be used around the body, however.

Within the body *forms*, *generic-function-symbol* is bound to the *generic function object*.

*Documentation* is attached as a *documentation string* to *name* (as kind **method-combination**) and to the *method combination object*.

## CLHS: Declaration DYNAMIC-EXTENT

Note that two methods with identical specializers, but with different *qualifiers*, are not ordered by the algorithm described in Step 2 of the method selection and combination process described in [Section 7.6.6 \(Method Selection and Combination\)](#). Normally the two methods play different roles in the effective method because they have different *qualifiers*, and no matter how they are ordered in the result of Step 2, the effective method is the same. If the two methods play the same role and their order matters, an error is signaled. This happens as part of the *qualifier* pattern matching in [define-method-combination](#).

If a [define-method-combination](#) form appears as a *top level form*, the *compiler* must make the *method combination name* be recognized as a valid *method combination name* in subsequent [defgeneric forms](#). However, the *method combination* is executed no earlier than when the [define-method-combination](#) form is executed, and possibly as late as the time that *generic functions* that use the *method combination* are executed.

### Examples:

Most examples of the long form of [define-method-combination](#) also illustrate the use of the related *functions* that are provided as part of the declarative method combination facility.

```
;; Examples of the short form of define-method-combination

(define-method-combination and :identity-with-one-argument t)

(defmethod func and ((x class1) y) ...)

;; The equivalent of this example in the long form is:

(define-method-combination and
  (&optional (order :most-specific-first))
  ((around (:around))
   (primary (and) :order order :required t))
  (let ((form (if (rest primary)
                  `(^and ,@(mapcar #'(lambda (method)
                                         `(^call-method ,method))
                                  primary))
                  `(^call-method ,(first primary))))))
    (if around
        `(^call-method ,(first around)
                     ,@(rest around)
                     (make-method ,form)))
        form)))

;; Examples of the long form of define-method-combination

;The default method-combination technique
(define-method-combination standard ()
  ((around (:around))
   (before (:before))
   (primary () :required t)
   (after (:after)))
  (flet ((call-methods (methods)
           (mapcar #'(lambda (method)
                       `(^call-method ,method))
                   methods)))
    (let ((form (if (or before after (rest primary))
                    `(^multiple-value-prog1
                      (progn ,@(call-methods before)
                             (call-method ,(first primary)
                                         ,(rest primary)))))))
```

## CLHS: Declaration DYNAMIC-EXTENT

```

        ,@(call-methods (reverse after)))
        `(call-method ,(first primary)))))

(if around
    `(call-method ,(first around)
        (,@(rest around)
            (make-method ,form)))
    form))

;A simple way to try several methods until one returns non-nil
(define-method-combination or ()
    ((methods (or)))
    `(or ,@(mapcar #'(lambda (method)
        `(call-method ,method))
        methods)))))

;A more complete version of the preceding
(define-method-combination or
    (&optional (order ':most-specific-first))
    ((around (:around))
     (primary (or)))
    ;; Process the order argument
    (case order
        (:most-specific-first)
        (:most-specific-last (setq primary (reverse primary)))
        (otherwise (method-combination-error "~S is an invalid order.~@
:most-specific-first and :most-specific-last are the possible values."
                                             order))))
    ;; Must have a primary method
    (unless primary
        (method-combination-error "A primary method is required."))
    ;; Construct the form that calls the primary methods
    (let ((form (if (rest primary)
        `(or ,@(mapcar #'(lambda (method)
            `(call-method ,method))
            primary))
        `(call-method ,(first primary))))))
    ;; Wrap the around methods around that form
    (if around
        `(call-method ,(first around)
            (,@(rest around)
                (make-method ,form)))
        form)))

;The same thing, using the :order and :required keyword options
(define-method-combination or
    (&optional (order ':most-specific-first))
    ((around (:around))
     (primary (or) :order order :required t)))
    (let ((form (if (rest primary)
        `(or ,@(mapcar #'(lambda (method)
            `(call-method ,method))
            primary))
        `(call-method ,(first primary))))))
    (if around
        `(call-method ,(first around)
            (,@(rest around)
                (make-method ,form)))
        form)))

;This short-form call is behaviorally identical to the preceding
(define-method-combination or :identity-with-one-argument t)

```

## CLHS: Declaration DYNAMIC-EXTENT

```
;Order methods by positive integer qualifiers
;:around methods are disallowed to keep the example small
(define-method-combination example-method-combination ()
  ((methods positive-integer-qualifier-p))
  `(progn ,@(mapcar #'(lambda (method)
    `(call-method ,method))
  (stable-sort methods #'<
    :key #'(lambda (method)
      (first (method-qualifiers method)))))))

(defun positive-integer-qualifier-p (method-qualifiers)
  (and (= (length method-qualifiers) 1)
    (typep (first method-qualifiers) '(integer 0 *)))))

;; Example of the use of :arguments
(define-method-combination progn-with-lock ()
  ((methods ()))
  (:arguments object)
  `(unwind-protect
    (progn (lock (object-lock ,object))
      ,@(mapcar #'(lambda (method)
        `(call-method ,method))
      methods))
    (unlock (object-lock ,object))))
```

**Affected By:** None.

**Side Effects:**

The *compiler* is not required to perform any compile-time side-effects.

**Exceptional Situations:**

Method combination types defined with the short form require exactly one *qualifier* per method. An error of *type error* is signaled if there are applicable methods with no *qualifiers* or with *qualifiers* that are not supported by the method combination type. At least one primary method must be applicable or an error of *type error* is signaled.

If an applicable method does not fall into any method group, the system signals an error of *type error* indicating that the method is invalid for the kind of method combination in use.

If the value of the :*required* option is *true* and the method group is empty (that is, no applicable methods match the *qualifier* patterns or satisfy the predicate), an error of *type error* is signaled.

If the :*order* option evaluates to a value other than :most-specific-first or :most-specific-last, an error of *type error* is signaled.

**See Also:**

[call-method](#), [call-next-method](#), [documentation](#), [method-qualifiers](#), [method-combination-error](#), [invalid-method-error](#), [defgeneric](#), [Section 7.6.6 \(Method Selection and Combination\)](#), [Section 7.6.6.4 \(Built-in Method Combination Types\)](#), [Section 3.4.11 \(Syntactic Interaction of Documentation Strings and Declarations\)](#)

**Notes:**

The :method-combination option of **defgeneric** is used to specify that a *generic function* should use a particular method combination type. The first argument to the :method-combination option is the *name* of a method combination type and the remaining arguments are options for that type.

***Macro DEFINE-CONDITION*****Syntax:**

**define-condition** *name* (*parent-type*\*) ({*slot-spec*}\*) *option*\*

=> *name*

*slot-spec*::= *slot-name* | (*slot-name* *slot-option*)

*slot-option*::= [{:reader symbol}\* |  
                  {:writer function-name}\* |  
                  {:accessor symbol}\* |  
                  {:allocation allocation-type} |  
                  {:initarg symbol}\* |  
                  {:initform form} |  
                  {:type type-specifier} ]]

*option*::= [[(:default-initargs . *initarg-list*) |  
                  (:documentation string) |  
                  (:report report-name) ]]

*function-name*::= {symbol | (setf symbol})

*allocation-type*::= :instance | :class

*report-name*::= string | symbol | lambda expression

**Arguments and Values:**

*name*---a symbol.

*parent-type*---a symbol naming a *condition type*. If no *parent-types* are supplied, the *parent-types* default to (condition).

*default-initargs*---a list of keyword/value pairs.

*Slot-spec* -- the *name* of a *slot* or a *list* consisting of the *slot-name* followed by zero or more *slot-options*.

*Slot-name* --- a slot name (a symbol), the *list* of a slot name, or the *list* of slot name/slot form pairs.

*Option* -- Any of the following:

:*reader*

:*reader* can be supplied more than once for a given *slot* and cannot be nil.

:*writer*

:*writer* can be supplied more than once for a given *slot* and must name a *generic function*.

```

:accessor
  :accessor can be supplied more than once for a given slot and cannot be nil.
:allocation
  :allocation can be supplied once at most for a given slot. The default if :allocation is not
  supplied is :instance.
:initarg
  :initarg can be supplied more than once for a given slot.
:initform
  :initform can be supplied once at most for a given slot.
:type
  :type can be supplied once at most for a given slot.
:documentation
  :documentation can be supplied once at most for a given slot.
:report
  :report can be supplied once at most.

```

**Description:**

**define-condition** defines a new condition type called *name*, which is a *subtype* of the *type* or *types* named by *parent-type*. Each *parent-type* argument specifies a direct *supertype* of the new *condition*. The new *condition* inherits *slots* and *methods* from each of its direct *supertypes*, and so on.

If a slot name/slot form pair is supplied, the slot form is a *form* that can be evaluated by **make-condition** to produce a default value when an explicit value is not provided. If no slot form is supplied, the contents of the *slot* is initialized in an *implementation-dependent* way.

If the *type* being defined and some other *type* from which it inherits have a slot by the same name, only one slot is allocated in the *condition*, but the supplied slot form overrides any slot form that might otherwise have been inherited from a *parent-type*. If no slot form is supplied, the inherited slot form (if any) is still visible.

Accessors are created according to the same rules as used by **defclass**.

A description of *slot-options* follows:

```

:reader
  The :reader slot option specifies that an unqualified method is to be defined on the generic
  function named by the argument to :reader to read the value of the given slot.
* The :initform slot option is used to provide a default initial value form to be used in the initialization of
  the slot. This form is evaluated every time it is used to initialize the slot. The lexical environment in which this
  form is evaluated is the lexical environment in which the define-condition form was evaluated. Note that the
  lexical environment refers both to variables and to functions. For local slots, the dynamic environment is the
  dynamic environment in which make-condition was called; for shared slots, the dynamic environment is the
  dynamic environment in which the define-condition form was evaluated.
  No implementation is permitted to extend the syntax of define-condition to allow (slot-name form)
  as an abbreviation for (slot-name :initform form).
:initarg
  The :initarg slot option declares an initialization argument named by its symbol argument and
  specifies that this initialization argument initializes the given slot. If the initialization argument has a
  value in the call to initialize-instance, the value is stored into the given slot, and the slot's
  :initform slot option, if any, is not evaluated. If none of the initialization arguments specified for
  a given slot has a value, the slot is initialized according to the :initform slot option, if specified.

```

## CLHS: Declaration DYNAMIC-EXTENT

### :type

The :type slot option specifies that the contents of the *slot* is always of the specified *type*. It effectively declares the result type of the reader generic function when applied to an *object* of this *condition* type. The consequences of attempting to store in a *slot* a value that does not satisfy the type of the *slot* is undefined.

### :default-initargs

This option is treated the same as it would be **defclass**.

### :documentation

The :documentation slot option provides a *documentation string* for the *slot*.

### :report

*Condition* reporting is mediated through the **print-object** method for the *condition* type in question, with \***print-escape**\* always being **nil**. Specifying (:report *report-name*) in the definition of a condition type C is equivalent to:

```
(defmethod print-object ((x c) stream)
  (if *print-escape* (call-next-method) (report-name x stream)))
```

If the value supplied by the argument to :report (*report-name*) is a *symbol* or a *lambda expression*, it must be acceptable to **function**. (*function report-name*) is evaluated in the current *lexical environment*. It should return a *function* of two arguments, a *condition* and a *stream*, that prints on the *stream* a description of the *condition*. This *function* is called whenever the *condition* is printed while \***print-escape**\* is **nil**.

If *report-name* is a *string*, it is a shorthand for

```
(lambda (condition stream)
  (declare (ignore condition))
  (write-string report-name stream))
```

This option is processed after the new *condition* type has been defined, so use of the *slot* accessors within the :report function is permitted. If this option is not supplied, information about how to report this type of *condition* is inherited from the *parent-type*.

The consequences are unspecified if an attempt is made to **read** a *slot* that has not been explicitly initialized and that has not been given a default value.

The consequences are unspecified if an attempt is made to assign the *slots* by using **setf**.

If a **define-condition form** appears as a *top level form*, the *compiler* must make *name* recognizable as a valid *type* name, and it must be possible to reference the *condition type* as the *parent-type* of another *condition type* in a subsequent **define-condition form** in the *file* being compiled.

### Examples:

The following form defines a condition of *type* peg/hole-mismatch which inherits from a condition type called blocks-world-error:

```
(define-condition peg/hole-mismatch
  (blocks-world-error)
  ((peg-shape :initarg :peg-shape
              :reader peg/hole-mismatch-peg-shape)
   (hole-shape :initarg :hole-shape
              :reader peg/hole-mismatch-hole-shape))
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(:report (lambda (condition stream)
    (format stream "A ~A peg cannot go in a ~A hole."
        (peg/hole-mismatch-peg-shape condition)
        (peg/hole-mismatch-hole-shape condition))))
```

The new type has slots `peg-shape` and `hole-shape`, so `make-condition` accepts `:peg-shape` and `:hole-shape` keywords. The `readers` `peg/hole-mismatch-peg-shape` and `peg/hole-mismatch-hole-shape` apply to objects of this type, as illustrated in the `:report` information.

The following form defines a `condition type` named `machine-error` which inherits from `error`:

```
(define-condition machine-error
  (error)
  ((machine-name :initarg :machine-name
                 :reader machine-error-machine-name)))
(:report (lambda (condition stream)
    (format stream "There is a problem with ~A."
        (machine-error-machine-name condition))))
```

Building on this definition, a new error condition can be defined which is a subtype of `machine-error` for use when machines are not available:

```
(define-condition machine-not-available-error (machine-error) ())
(:report (lambda (condition stream)
    (format stream "The machine ~A is not available."
        (machine-error-machine-name condition))))
```

This defines a still more specific condition, built upon `machine-not-available-error`, which provides a slot initialization form for `machine-name` but which does not provide any new slots or report information. It just gives the `machine-name` slot a default initialization:

```
(define-condition my-favorite-machine-not-available-error
  (machine-not-available-error)
  ((machine-name :initform "mc.lcs.mit.edu")))
```

Note that since no `:report` clause was given, the information inherited from `machine-not-available-error` is used to report this type of condition.

```
(define-condition ate-too-much (error)
  ((person :initarg :person :reader ate-too-much-person)
   (weight :initarg :weight :reader ate-too-much-weight)
   (kind-of-food :initarg :kind-of-food
                 :reader :ate-too-much-kind-of-food)))
=> ATE-TOO-MUCH
(define-condition ate-too-much-ice-cream (ate-too-much)
  ((kind-of-food :initform 'ice-cream)
   (flavor :initarg :flavor
           :reader ate-too-much-ice-cream-flavor
           :initform 'vanilla)))
(:report (lambda (condition stream)
    (format stream "~A ate too much ~A ice-cream"
        (ate-too-much-person condition)
        (ate-too-much-ice-cream-flavor condition))))
=> ATE-TOO-MUCH-ICE-CREAM
(make-condition 'ate-too-much-ice-cream
  :person 'fred)
```

## CLHS: Declaration DYNAMIC-EXTENT

```
:weight 300
:flavor 'chocolate)
=> #<ATE-TOO-MUCH-ICE-CREAM 32236101>
(format t "~A" *)
>> FRED ate too much CHOCOLATE ice-cream
=> NIL
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[make-condition](#), [defclass](#), [Section 9.1 \(Condition System Concepts\)](#)

**Notes:** None.

## **Macro DEFINE-COMPILER-MACRO**

**Syntax:**

**define-compiler-macro** *name lambda-list [[declaration\* / documentation]] form\**

=> *name*

**Arguments and Values:**

*name*—a function name.

*lambda-list*—a macro lambda list.

*declaration*—a declare expression; not evaluated.

*documentation*—a string; not evaluated.

*form*—a form.

**Description:**

This is the normal mechanism for defining a compiler macro function. Its manner of definition is the same as for **defmacro**; the only differences are:

\* The name can be a function name naming any function or macro.

\* The expander function is installed as a compiler macro function for the name, rather than as a macro function.

\* The &whole argument is bound to the form argument that is passed to the compiler macro function. The remaining lambda-list parameters are specified as if this form contained the function name in the car and the actual arguments in the cdr, but if the car of the actual form is the symbol funcall, then the destructuring of the arguments is actually performed using its cddr instead.

\* Documentation is attached as a documentation string to *name* (as kind **compiler-macro**) and to the compiler macro function.

## CLHS: Declaration DYNAMIC-EXTENT

*\* Unlike an ordinary macro, a compiler macro can decline to provide an expansion merely by returning a form that is the same as the original (which can be obtained by using &whole).*

### Examples:

```
(defun square (x) (expt x 2)) =>  SQUARE
(define-compiler-macro square (&whole form arg)
  (if (atom arg)
      `(expt ,arg 2)
      (case (car arg)
        (square (if (= (length arg) 2)
                   `',(expt ,(nth 1 arg) 4)
                   form))
        (expt   (if (= (length arg) 3)
                   (if (numberp (nth 2 arg))
                       `',(expt ,(nth 1 arg) ,(* 2 (nth 2 arg)))
                       `',(expt ,(nth 1 arg) (* 2 ,(nth 2 arg)))))
                   form))
        (otherwise `',(expt ,arg 2)))) =>  SQUARE
(square (square 3)) =>  81
(macroexpand '(square x)) =>  (SQUARE X), false
(funcall (compiler-macro-function 'square) '(square x) nil)
=>  (EXPT X 2)
(funcall (compiler-macro-function 'square) '(square (square x)) nil)
=>  (EXPT X 4)
(funcall (compiler-macro-function 'square) '(funcall #'square x) nil)
=>  (EXPT X 2)

(defun distance-positional (x1 y1 x2 y2)
  (sqrt (+ (expt (- x2 x1) 2) (expt (- y2 y1) 2))))
=>  DISTANCE-POSITIONAL
(defun distance (&key (x1 0) (y1 0) (x2 x1) (y2 y1))
  (distance-positional x1 y1 x2 y2))
=>  DISTANCE
(define-compiler-macro distance (&whole form
                                         &rest key-value-pairs
                                         &key (x1 0 x1-p)
                                               (y1 0 y1-p)
                                               (x2 x1 x2-p)
                                               (y2 y1 y2-p)
                                         &allow-other-keys
                                         &environment env)
  (flet ((key (n) (nth (* n 2) key-value-pairs))
         (arg (n) (nth (1+ (* n 2)) key-value-pairs))
         (simplep (x)
           (let ((expanded-x (macroexpand x env)))
             (or (constantp expanded-x env)
                 (symbolp expanded-x))))
         (let ((n (/ (length key-value-pairs) 2)))
           (multiple-value-bind (x1s y1s x2s y2s others)
               (loop for (key) on key-value-pairs by #'cddr
                     count (eq key ':x1) into x1s
                     count (eq key ':y1) into y1s
                     count (eq key ':x2) into x2s
                     count (eq key ':y2) into y2s
                     count (not (member key '(:x1 :x2 :y1 :y2)))
                           into others
                     finally (return (values x1s y1s x2s y2s others)))
             (cond ((and (= n 4)
                         (eq (key 0) :x1)
                         (eq (key 1) :y1)
                         (eq (key 2) :x2)
                         (eq (key 3) :y2))
                     (multiple-value-bind (x1 y1 x2 y2)
                         (loop for (key) on key-value-pairs by #'cddr
                               count (eq key ':x1) into x1
                               count (eq key ':y1) into y1
                               count (eq key ':x2) into x2
                               count (eq key ':y2) into y2
                               finally (return (values x1 y1 x2 y2)))
                     (values x1 y1 x2 y2)))
                   (t (values x1s y1s x2s y2s others)))))))
```

## CLHS: Declaration DYNAMIC-EXTENT

```

(eq (key 1) :y1)
(eq (key 2) :x2)
(eq (key 3) :y2))
`(distance-positional ,x1 ,y1 ,x2 ,y2))
((and (if x1-p (and (= x1s 1) (simplep x1)) t)
        (if y1-p (and (= y1s 1) (simplep y1)) t)
        (if x2-p (and (= x2s 1) (simplep x2)) t)
        (if y2-p (and (= y2s 1) (simplep y2)) t)
        (zerop others))
`(distance-positional ,x1 ,y1 ,x2 ,y2))
((and (< x1s 2) (< y1s 2) (< x2s 2) (< y2s 2)
      (zerop others))
(let ((temp (loop repeat n collect (gensym))))
  ` (let ,(loop for i below n
               collect (list (nth i temps) (arg i)))
       (distance
        ,@(loop for i below n
                append (list (key i) (nth i temps)))))))
  (t form)))))

=> DISTANCE
(dolist (form
  '(((distance :x1 (setq x 7) :x2 (decf x) :y1 (decf x) :y2 (decf x))
     (distance :x1 (setq x 7) :y1 (decf x) :x2 (decf x) :y2 (decf x))
     (distance :x1 (setq x 7) :y1 (incf x))
     (distance :x1 (setq x 7) :y1 (incf x) :x1 (incf x))
     (distance :x1 a1 :y1 b1 :x2 a2 :y2 b2)
     (distance :x1 a1 :x2 a2 :y1 b1 :y2 b2)
     (distance :x1 a1 :y1 b1 :z1 c1 :x2 a2 :y2 b2 :z2 c2)))
  (print (funcall (compiler-macro-function 'distance) form nil)))
>> (LET ((#:G6558 (SETQ X 7)))
>>      (#:G6559 (DECFL X)))
>>      (#:G6560 (DECFL X)))
>>      (#:G6561 (DECFL X)))
>>      (DISTANCE :X1 #:G6558 :X2 #:G6559 :Y1 #:G6560 :Y2 #:G6561))
>>      (DISTANCE-POSITIONAL (SETQ X 7) (DECFL X) (DECFL X) (DECFL X)))
>>      (LET ((#:G6567 (SETQ X 7)))
>>          (#:G6568 (INCF X)))
>>          (DISTANCE :X1 #:G6567 :Y1 #:G6568))
>>          (DISTANCE :X1 (SETQ X 7) :Y1 (INCF X) :X1 (INCF X)))
>>          (DISTANCE-POSITIONAL A1 B1 A2 B2)
>>          (DISTANCE-POSITIONAL A1 B1 A2 B2))
>>          (DISTANCE :X1 A1 :Y1 B1 :Z1 C1 :X2 A2 :Y2 B2 :Z2 C2))
=> NIL

```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[compiler-macro-function](#), [defmacro](#), [documentation](#), [Section 3.4.11 \(Syntactic Interaction of Documentation Strings and Declarations\)](#)

**Notes:**

The consequences of writing a *compiler macro* definition for a function in the COMMON-LISP package are undefined; it is quite possible that in some *implementations* such an attempt would override an equivalent or equally important definition. In general, it is recommended that a programmer only write *compiler macro*

definitions for *functions* he or she personally maintains—writing a *compiler macro* definition for a function maintained elsewhere is normally considered a violation of traditional rules of modularity and data abstraction.

## **Macro DEFMACRO**

### Syntax:

**defmacro** *name lambda-list* [[*declaration\** / *documentation*]] *form\**

=> *name*

### Arguments and Values:

*name*—a *symbol*. *lambda-list*—a *macro lambda list*.

*declaration*—a *declare expression*; not evaluated.

*documentation*—a *string*; not evaluated.

*form*—a *form*.

### Description:

Defines *name* as a *macro* by associating a *macro function* with that *name* in the global environment. The *macro function* is defined in the same *lexical environment* in which the **defmacro** form appears.

The parameter variables in *lambda-list* are bound to destructured portions of the macro call.

The expansion function accepts two arguments, a *form* and an *environment*. The expansion function returns a *form*. The body of the expansion function is specified by *forms*. *Forms* are executed in order. The value of the last *form* executed is returned as the expansion of the *macro*. The body *forms* of the expansion function (but not the *lambda-list*) are implicitly enclosed in a *block* whose name is *name*.

The *lambda-list* conforms to the requirements described in Section 3.4.4 (Macro Lambda Lists).

*Documentation* is attached as a *documentation string* to *name* (as kind **function**) and to the *macro function*.

**defmacro** can be used to redefine a *macro* or to replace a *function* definition with a *macro* definition.

Recursive expansion of the *form* returned must terminate, including the expansion of other *macros* which are *subforms* of other *forms* returned.

The consequences are undefined if the result of fully macroexpanding a *form* contains any *circular list structure* except in *literal objects*.

If a **defmacro** form appears as a *top level form*, the *compiler* must store the *macro* definition at compile time, so that occurrences of the macro later on in the file can be expanded correctly. Users must ensure that the body of the *macro* can be evaluated at compile time if it is referenced within the *file* being *compiled*.

### Examples:

## CLHS: Declaration DYNAMIC-EXTENT

```
(defmacro mac1 (a b) "Mac1 multiplies and adds"
  `(+ ,a (* ,b 3))) => MAC1
(mac1 4 5) => 19
(documentation 'mac1 'function) => "Mac1 multiplies and adds"
(defmacro mac2 (&optional (a 2 b) (c 3 d) &rest x) `'(,a ,b ,c ,d ,x)) => MAC2
(mac2 6) => (6 T 3 NIL NIL)
(mac2 6 3 8) => (6 T 3 T (8))
(defmacro mac3 (&whole r a &optional (b 3) &rest x &key c (d a))
  `'(',r ,a ,b ,c ,d ,x)) => MAC3
(mac3 1 6 :d 8 :c 9 :d 10) => ((MAC3 1 6 :D 8 :C 9 :D 10) 1 6 9 8 (:D 8 :C 9 :D 10))
```

The stipulation that an embedded *destructuring lambda list* is permitted only where *ordinary lambda list* syntax would permit a parameter name but not a *list* is made to prevent ambiguity. For example, the following is not valid:

```
(defmacro loser (x &optional (a b &rest c) &rest z)
  ...)
```

because *ordinary lambda list* syntax does permit a *list* following &optional; the list (a b &rest c) would be interpreted as describing an optional parameter named a whose default value is that of the form b, with a supplied-p parameter named &rest (not valid), and an extraneous symbol c in the list (also not valid). An almost correct way to express this is

```
(defmacro loser (x &optional ((a b &rest c)) &rest z)
  ...)
```

The extra set of parentheses removes the ambiguity. However, the definition is now incorrect because a macro call such as (loser (car pool)) would not provide any argument form for the lambda list (a b &rest c), and so the default value against which to match the *lambda list* would be *nil* because no explicit default value was specified. The consequences of this are unspecified since the empty list, *nil*, does not have *forms* to satisfy the parameters a and b. The fully correct definition would be either

```
(defmacro loser (x &optional ((a b &rest c) '(nil nil)) &rest z)
  ...)
```

or

```
(defmacro loser (x &optional ((&optional a b &rest c)) &rest z)
  ...)
```

These differ slightly: the first requires that if the macro call specifies a explicitly then it must also specify b explicitly, whereas the second does not have this requirement. For example,

```
(loser (car pool) ((+ x 1)))
```

would be a valid call for the second definition but not for the first.

```
(defmacro dm1a (&whole x) `',x)
(macroexpand '(dm1a)) => (QUOTE (DM1A))
(macroexpand '(dm1a a)) is an error.

(defmacro dm1b (&whole x a &optional b) `'(,x ,a ,b))
(macroexpand '(dm1b)) is an error.
(macroexpand '(dm1b q)) => (QUOTE ((DM1B Q) Q NIL))
(macroexpand '(dm1b q r)) => (QUOTE ((DM1B Q R) Q R))
(macroexpand '(dm1b q r s)) is an error.
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(defmacro dm2a (&whole form a b) `'(form ,form a ,a b ,b))
(macroexpand '(dm2a x y)) => (QUOTE (FORM (DM2A X Y) A X B Y))
(dm2a x y) => (FORM (DM2A X Y) A X B Y)

(defmacro dm2b (&whole form a (&whole b (c . d) &optional (e 5))
                         &body f &environment env)
  "(& ',form ,,a ,',b ,',(macroexpand c env) ,',d ,',e ,',f))
;Note that because backquote is involved, implementations may differ
;slightly in the nature (though not the functionality) of the expansion.
(macroexpand '(dm2b x1 (((incf x2) x3 x4)) x5 x6))
=> (LIST* '(DM2B X1 (((INCF X2) X3 X4))
            X5 X6)
           X1
           '((((INCF X2) X3 X4)) (SETQ X2 (+ X2 1)) (X3 X4) 5 (X5 X6))),  
T
(let ((x1 5))
  (macrolet ((segundo (x) `(cadr ,x)))
    (dm2b x1 (((segundo x2) x3 x4)) x5 x6)))
=> ((DM2B X1 (((SEGUNDO X2) X3 X4)) X5 X6)
      5 (((SEGUNDO X2) X3 X4)) (CADR X2) (X3 X4) 5 (X5 X6)))
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[define-compiler-macro](#), [destructuring-bind](#), [documentation](#), [macroexpand](#), [\\*macroexpand-hook\\*](#), [macrolet](#), [macro-function](#), [Section 3.1 \(Evaluation\)](#), [Section 3.2 \(Compilation\)](#), [Section 3.4.11 \(Syntactic Interaction of Documentation Strings and Declarations\)](#)

**Notes:** None.

## Macro DEFMETHOD

**Syntax:**

**defmethod** *function-name* {*method-qualifier*}\* *specialized-lambda-list* [[*declaration*\* / *documentation*]] *form*\*

=> *new-method*

*function-name*::= {symbol | (*setf symbol*)}

*method-qualifier*::=non-list

*specialized-lambda-list*::= ({var | (var parameter-specializer-name)}\*  
 [&optional {var | (var [initform [supplied-p-parameter] ])}\*]  
 [&rest var]  
 [&key{var | ({var | (keywordvar)} [initform [supplied-p-parameter]  
 [&allow-other-keys] ]  
 [&aux {var | (var [initform] )}\*] } ]  
 parameter-specializer-name::= symbol | (eql eql-specializer-form)

**Arguments and Values:**

*declaration*—a declare expression; not evaluated.

*documentation*—a string; not evaluated.

*var*—a variable name.

*eql-specializer-form*—a form.

*Form*—a form.

*Initform*—a form.

*Supplied-p-parameter*—variable name.

*new-method*—the new method object.

### Description:

The macro **defmethod** defines a method on a generic function.

If (`fboundp function-name`) is nil, a generic function is created with default values for the argument precedence order (each argument is more specific than the arguments to its right in the argument list), for the generic function class (the class standard-generic-function), for the method class (the class standard-method), and for the method combination type (the standard method combination type). The lambda list of the generic function is congruent with the lambda list of the method being defined; if the **defmethod** form mentions keyword arguments, the lambda list of the generic function will mention . . . . key (but no keyword arguments). If *function-name* names an ordinary function, a macro, or a special operator, an error is signaled.

If a generic function is currently named by *function-name*, the lambda list of the method must be congruent with the lambda list of the generic function. If this condition does not hold, an error is signaled. For a definition of congruence in this context, see [Section 7.6.4 \(Congruent Lambda-lists for all Methods of a Generic Function\)](#).

Each *method-qualifier* argument is an object that is used by method combination to identify the given method. The method combination type might further restrict what a method qualifier can be. The standard method combination type allows for unqualified methods and methods whose sole qualifier is one of the keywords :before, :after, or :around.

The *specialized-lambda-list* argument is like an ordinary lambda list except that the names of required parameters can be replaced by specialized parameters. A specialized parameter is a list of the form (*var parameter-specializer-name*). Only required parameters can be specialized. If *parameter-specializer-name* is a symbol it names a class; if it is a list, it is of the form (`eq1 eql-specializer-form`). The parameter specializer name (`eq1 eql-specializer-form`) indicates that the corresponding argument must be eq1 to the object that is the value of *eql-specializer-form* for the method to be applicable. The *eql-specializer-form* is evaluated at the time that the expansion of the **defmethod** macro is evaluated. If no parameter specializer name is specified for a given required parameter, the parameter specializer defaults to the class t. For further discussion, see [Section 7.6.2 \(Introduction to Methods\)](#).

The *form* arguments specify the method body. The body of the method is enclosed in an implicit block. If *function-name* is a symbol, this block bears the same name as the generic function. If *function-name* is a list

## CLHS: Declaration DYNAMIC-EXTENT

of the form (`setf symbol`), the name of the block is *symbol*.

The class of the method object that is created is that given by the method class option of the generic function on which the method is defined.

If the generic function already has a method that agrees with the method being defined on parameter specializers and qualifiers, **defmethod** replaces the existing method with the one now being defined. For a definition of agreement in this context. see [Section 7.6.3 \(Agreement on Parameter Specializers and Qualifiers\)](#).

The parameter specializers are derived from the parameter specializer names as described in [Section 7.6.2 \(Introduction to Methods\)](#).

The expansion of the **defmethod** macro "refers to" each specialized parameter (see the description of **ignore** within the description of **declare**). This includes parameters that have an explicit parameter specializer name of `t`. This means that a compiler warning does not occur if the body of the method does not refer to a specialized parameter, while a warning might occur if the body of the method does not refer to an unspecialized parameter. For this reason, a parameter that specializes on `t` is not quite synonymous with an unspecialized parameter in this context.

Declarations at the head of the method body that apply to the method's lambda variables are treated as bound declarations whose scope is the same as the corresponding bindings.

Declarations at the head of the method body that apply to the functional bindings of call-next-method or next-method-p apply to references to those functions within the method body *forms*. Any outer bindings of the function names call-next-method and next-method-p, and declarations associated with such bindings are *shadowed*[2] within the method body *forms*.

The scope of free declarations at the head of the method body is the entire method body, which includes any implicit local function definitions but excludes initialization forms for the lambda variables.

**defmethod** is not required to perform any compile-time side effects. In particular, the methods are not installed for invocation during compilation. An implementation may choose to store information about the generic function for the purposes of compile-time error-checking (such as checking the number of arguments on calls, or noting that a definition for the function name has been seen).

*Documentation* is attached as a documentation string to the method object.

**Examples:** None.

**Affected By:**

The definition of the referenced generic function.

**Exceptional Situations:**

If *function-name* names an ordinary function, a macro, or a special operator, an error of type error is signaled.

If a generic function is currently named by *function-name*, the lambda list of the method must be congruent with the lambda list of the generic function, or an error of type error is signaled.

**See Also:**

[\*\*defgeneric\*\*](#), [\*\*documentation\*\*](#), [Section 7.6.2 \(Introduction to Methods\)](#), [Section 7.6.4 \(Congruent Lambda-lists for all Methods of a Generic Function\)](#), [Section 7.6.3 \(Agreement on Parameter Specializers and Qualifiers\)](#), [Section 3.4.11 \(Syntactic Interaction of Documentation Strings and Declarations\)](#)

**Notes:** None.

## Macro DEFPARAMETER, DEFVAR

**Syntax:**

**defparameter** *name initial-value [documentation]* => *name*

**defvar** *name [initial-value [documentation]]* => *name*

**Arguments and Values:**

*name*—a symbol; not evaluated.

*initial-value*—a form; for **defparameter**, it is always *evaluated*, but for **defvar** it is *evaluated* only if *name* is not already bound.

*documentation*—a string; not evaluated.

**Description:**

**defparameter** and **defvar** establish *name* as a dynamic variable.

**defparameter** unconditionally assigns the *initial-value* to the dynamic variable named *name*. **defvar**, by contrast, assigns *initial-value* (if supplied) to the dynamic variable named *name* only if *name* is not already bound.

If no *initial-value* is supplied, **defvar** leaves the value cell of the dynamic variable named *name* undisturbed; if *name* was previously bound, its old value persists, and if it was previously unbound, it remains unbound.

If *documentation* is supplied, it is attached to *name* as a documentation string of kind **variable**.

**defparameter** and **defvar** normally appear as a top level form, but it is meaningful for them to appear as non-top-level forms. However, the compile-time side effects described below only take place when they appear as top level forms.

**Examples:**

```
(defparameter *p* 1) => *p*
*p* => 1
(constantp '*p*) => false
(setq *p* 2) => 2
(defparameter *p* 3) => *p*
*p* => 3
```

```
(defvar *v* 1) => *V*
*V* => 1
(constantly '*V*) => false
(setq *v* 2) => 2
(defvar *v* 3) => *V*
*V* => 2

(defun foo ()
  (let ((*p* 'p) (*v* 'v))
    (bar))) => FOO
(defun bar () (list *p* *v*)) => BAR
(foo) => (P V)
```

The principal operational distinction between **defparameter** and **defvar** is that **defparameter** makes an unconditional assignment to *name*, while **defvar** makes a conditional one. In practice, this means that **defparameter** is useful in situations where loading or reloading the definition would want to pick up a new value of the variable, while **defvar** is used in situations where the old value would want to be retained if the file were loaded or reloaded. For example, one might create a file which contained:

```
(defvar *the-interesting-numbers* '())
(defmacro define-interesting-number (name n)
  `(progn (defvar ,name ,n)
           (pushnew ,name *the-interesting-numbers*)
           ',name))
(define-interesting-number *my-height* 168) ;cm
(define-interesting-number *my-weight* 13) ;stones
```

Here the initial value, `()`, for the variable `*the-interesting-numbers*` is just a seed that we are never likely to want to reset to something else once something has been grown from it. As such, we have used **defvar** to avoid having the `*interesting-numbers*` information reset if the file is loaded a second time. It is true that the two calls to **define-interesting-number** here would be reprocessed, but if there were additional calls in another file, they would not be and that information would be lost. On the other hand, consider the following code:

```
(defparameter *default-beep-count* 3)
(defun beep (&optional (n *default-beep-count*))
  (dotimes (i n) (si:%beep 1000. 100000.) (sleep 0.1)))
```

Here we could easily imagine editing the code to change the initial value of `*default-beep-count*`, and then reloading the file to pick up the new value. In order to make value updating easy, we have used **defparameter**.

On the other hand, there is potential value to using **defvar** in this situation. For example, suppose that someone had predefined an alternate value for `*default-beep-count*`, or had loaded the file and then manually changed the value. In both cases, if we had used **defvar** instead of **defparameter**, those user preferences would not be overridden by (re)loading the file.

The choice of whether to use **defparameter** or **defvar** has visible consequences to programs, but is nevertheless often made for subjective reasons.

### Side Effects:

If a **defvar** or **defparameter** form appears as a *top level form*, the *compiler* must recognize that the *name* has been proclaimed **special**. However, it must neither *evaluate* the *initial-value form* nor *assign* the *dynamic*

variable named *name* at compile time.

There may be additional (*implementation-defined*) compile-time or run-time side effects, as long as such effects do not interfere with the correct operation of conforming programs.

#### Affected By:

defvar is affected by whether *name* is already bound.

**Exceptional Situations:** None.

#### See Also:

declare, defconstant, documentation, Section 3.2 (Compilation)

#### Notes:

It is customary to name dynamic variables with an asterisk at the beginning and end of the name. e.g., `*foo*` is a good name for a dynamic variable, but not for a lexical variable; `foo` is a good name for a lexical variable, but not for a dynamic variable. This naming convention is observed for all defined names in Common Lisp; however, neither conforming programs nor conforming implementations are obliged to adhere to this convention.

The intent of the permission for additional side effects is to allow implementations to do normal "bookkeeping" that accompanies definitions. For example, the macro expansion of a defvar or defparameter form might include code that arranges to record the name of the source file in which the definition occurs.

defparameter and defvar might be defined as follows:

```
(defmacro defparameter (name initial-value
                        &optional (documentation nil documentation-p))
  `(progn (declare (special ,name))
          (setf (symbol-value ',name) ,initial-value)
          ,(when documentation-p
              `',(setf (documentation ',name 'variable) ',documentation))
          ',name))
(defmacro defvar (name &optional
                  (initial-value nil initial-value-p)
                  (documentation nil documentation-p))
  `(progn (declare (special ,name))
          ,(when initial-value-p
              `(unless (boundp ',name)
                  (setf (symbol-value ',name) ,initial-value)))
          ,(when documentation-p
              `',(setf (documentation ',name 'variable) ',documentation))
          ',name))
```

## Macro DEFPACKAGE

#### Syntax:

**defpackage** *defined-package-name* [[*option*]] => *package*

*option*::= (:nicknames *nickname*\*)\* |

## CLHS: Declaration DYNAMIC-EXTENT

```
(:documentation string) |  
(:use package-name*)* |  
(:shadow {symbol-name})* |  
(:shadowing-import-from package-name {symbol-name}* )* |  
(:import-from package-name {symbol-name}* )* |  
(:export {symbol-name}* )* |  
(:intern {symbol-name}* )* |  
(:size integer)
```

### Arguments and Values:

*defined-package-name*—a string designator.

*package-name*—a package designator.

*nickname*—a string designator.

*symbol-name*—a string designator.

*package*—the package named *package-name*.

### Description:

**defpackage** creates a package as specified and returns the package.

If *defined-package-name* already refers to an existing package, the name-to-package mapping for that name is not changed. If the new definition is at variance with the current state of that package, the consequences are undefined; an implementation might choose to modify the existing package to reflect the new definition. If *defined-package-name* is a symbol, its name is used.

The standard *options* are described below.

#### :nicknames

The arguments to :nicknames set the package's nicknames to the supplied names.

#### :documentation

The argument to :documentation specifies a documentation string; it is attached as a documentation string to the package. At most one :documentation option can appear in a single **defpackage form**.

#### :use

The arguments to :use set the packages that the package named by *package-name* will inherit from. If :use is not supplied, it defaults to the same implementation-dependent value as the :use argument to **make-package**.

#### :shadow

The arguments to :shadow, *symbol-names*, name symbols that are to be created in the package being defined. These symbols are added to the list of shadowing symbols effectively as if by shadow.

#### :shadowing-import-from

The symbols named by the argument *symbol-names* are found (involving a lookup as if by **find-symbol**) in the specified *package-name*. The resulting symbols are imported into the package being defined, and placed on the shadowing symbols list as if by shadowing-import. In no case are symbols created in any package other than the one being defined.

#### :import-from

## CLHS: Declaration DYNAMIC-EXTENT

The *symbols* named by the argument *symbol-names* are found in the *package* named by *package-name* and they are *imported* into the *package* being defined. In no case are *symbols* created in any *package* other than the one being defined.

### :export

The *symbols* named by the argument *symbol-names* are found or created in the *package* being defined and *exported*. The :export option interacts with the :use option, since inherited *symbols* can be used rather than new ones created. The :export option interacts with the :import-from and :shadowing-import-from options, since *imported symbols* can be used rather than new ones created. If an argument to the :export option is *accessible* as an (inherited) *internal symbol* via *use-package*, that the *symbol* named by *symbol-name* is first *imported* into the *package* being defined, and is then *exported* from that *package*.

### :intern

The *symbols* named by the argument *symbol-names* are found or created in the *package* being defined. The :intern option interacts with the :use option, since inherited *symbols* can be used rather than new ones created.

### :size

The argument to the :size option declares the approximate number of *symbols* expected in the *package*. This is an efficiency hint only and might be ignored by an implementation.

The order in which the options appear in a **defpackage** form is irrelevant. The order in which they are executed is as follows:

1. :shadow and :shadowing-import-from.
2. :use.
3. :import-from and :intern.
4. :export.

Shadows are established first, since they might be necessary to block spurious name conflicts when the :use option is processed. The :use option is executed next so that :intern and :export options can refer to normally inherited *symbols*. The :export option is executed last so that it can refer to *symbols* created by any of the other options; in particular, *shadowing symbols* and *imported symbols* can be made external.

If a defpackage *form* appears as a *top level form*, all of the actions normally performed by this *macro* at load time must also be performed at compile time.

### Examples:

```
(defpackage "MY-PACKAGE"
  (:nicknames "MYPKG" "MY-PKG")
  (:use "COMMON-LISP")
  (:shadow "CAR" "CDR")
  (:shadowing-import-from "VENDOR-COMMON-LISP" "CONS")
  (:import-from "VENDOR-COMMON-LISP" "GC")
  (:export "EQ" "CONS" "FROBOLA")
  )

(defpackage my-package
  (:nicknames mypkg :MY-PKG) ; remember Common Lisp conventions for case
  (:use common-lisp) ; conversion on symbols
  (:shadow CAR :cdr #:cons)
  (:export "CONS") ; this is the shadowed one.
)
```

**Affected By:**

Existing *packages*.

**Exceptional Situations:**

If one of the supplied :nicknames already refers to an existing *package*, an error of *type package-error* is signaled.

An error of *type program-error* should be signaled if :size or :documentation appears more than once.

Since *implementations* might allow extended *options* an error of *type program-error* should be signaled if an *option* is present that is not actually supported in the host *implementation*.

The collection of *symbol-name* arguments given to the options :shadow, :intern, :import-from, and :shadowing-import-from must all be disjoint; additionally, the *symbol-name* arguments given to :export and :intern must be disjoint. Disjoint in this context is defined as no two of the *symbol-names* being *string=* with each other. If either condition is violated, an error of *type program-error* should be signaled.

For the :shadowing-import-from and :import-from options, a *correctable error* of *type package-error* is signaled if no *symbol* is *accessible* in the *package* named by *package-name* for one of the argument *symbol-names*.

Name conflict errors are handled by the underlying calls to *make-package*, *use-package*, *import*, and *export*. See [Section 11.1 \(Package Concepts\)](#).

**See Also:**

[documentation](#), [Section 11.1 \(Package Concepts\)](#), [Section 3.2 \(Compilation\)](#)

**Notes:**

The :intern option is useful if an :import-from or a :shadowing-import-from option in a subsequent call to *defpackage* (for some other *package*) expects to find these *symbols accessible* but not necessarily external.

It is recommended that the entire *package* definition is put in a single place, and that all the *package* definitions of a program are in a single file. This file can be *loaded* before *loading* or compiling anything else that depends on those *packages*. Such a file can be read in the COMMON-LISP-USER package, avoiding any initial state issues.

**defpackage** cannot be used to create two "mutually recursive" packages, such as:

```
(defpackage my-package
  (:use common-lisp your-package)      ; requires your-package to exist first
  (:export "MY-FUN"))
(defpackage your-package
  (:use common-lisp)
  (:import-from my-package "MY-FUN") ; requires my-package to exist first
  (:export "MY-FUN"))
```

However, nothing prevents the user from using the package-affecting functions such as use-package, import, and export to establish such links after a more standard use of defpackage.

The macroexpansion of defpackage could usefully canonicalize the names into strings, so that even if a source file has random symbols in the defpackage form, the compiled file would only contain strings.

Frequently additional implementation-dependent options take the form of a keyword standing by itself as an abbreviation for a list (keyword T); this syntax should be properly reported as an unrecognized option in implementations that do not support it.

## **Macro DEFSETF**

### Syntax:

The "short form":

**defsetf** *access-fn update-fn [documentation]*

=> *access-fn*

The "long form":

**defsetf** *access-fn lambda-list (store-variable\*) [[declaration\* / documentation]] form\**

=> *access-fn*

### Arguments and Values:

*access-fn*—a symbol which names a function or a macro.

*update-fn*—a symbol naming a function or macro.

*lambda-list*—a defsetf lambda list.

*store-variable*—a symbol (a variable name).

*declaration*—a declare expression; not evaluated.

*documentation*—a string; not evaluated.

*form*—a form.

### Description:

**defsetf** defines how to setf a place of the form (*access-fn* . . .) for relatively simple cases. (See define-setf-expander for more general access to this facility.) It must be the case that the function or macro named by *access-fn* evaluates all of its arguments.

**defsetf** may take one of two forms, called the "short form" and the "long form," which are distinguished by the type of the second argument.

## CLHS: Declaration DYNAMIC-EXTENT

When the short form is used, *update-fn* must name a *function* (or *macro*) that takes one more argument than *access-fn* takes. When *setf* is given a *place* that is a call on *access-fn*, it expands into a call on *update-fn* that is given all the arguments to *access-fn* and also, as its last argument, the new value (which must be returned by *update-fn* as its value).

The long form **defsetf** resembles **defmacro**. The *lambda-list* describes the arguments of *access-fn*. The *store-variables* describe the value or values to be stored into the *place*. The *body* must compute the expansion of a *setf* of a call on *access-fn*. The expansion function is defined in the same *lexical environment* in which the **defsetf form** appears.

During the evaluation of the *forms*, the variables in the *lambda-list* and the *store-variables* are bound to names of temporary variables, generated as if by **gensym** or **gentemp**, that will be bound by the expansion of *setf* to the values of those *subforms*. This binding permits the *forms* to be written without regard for order-of-evaluation issues. **defsetf** arranges for the temporary variables to be optimized out of the final result in cases where that is possible.

The body code in **defsetf** is implicitly enclosed in a *block* whose name is *access-fn*

**defsetf** ensures that *subforms* of the *place* are evaluated exactly once.

*Documentation* is attached to *access-fn* as a *documentation string* of kind *setf*.

If a **defsetf form** appears as a *top level form*, the *compiler* must make the *setf expander* available so that it may be used to expand calls to *setf* later on in the *file*. Users must ensure that the *forms*, if any, can be evaluated at compile time if the *access-fn* is used in a *place* later in the same *file*. The *compiler* must make these *setf expanders* available to compile-time calls to **get-setf-expansion** when its *environment* argument is a value received as the *environment parameter* of a *macro*.

### Examples:

The effect of

```
(defsetf symbol-value set)
```

is built into the Common Lisp system. This causes the form (*setf* (*symbol-value* *foo*) *fu*) to expand into (*set* *foo* *fu*).

Note that

```
(defsetf car rplaca)
```

would be incorrect because *rplaca* does not return its last argument.

```
(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
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(setf (middleguy b) 7) => 7
(setf (middleguy (middleguy c)) 'middleguy-symbol) => MIDDLEGUY-SYMBOL
a => (A 3 C D)
b => (7)
c => (1 2 3 (4 MIDDLEGUY-SYMBOL 6) 7 8 9)
```

An example of the use of the long form of **defsetf**:

```
(defsetf subseq (sequence start &optional end) (new-sequence)
  `(progn (replace ,sequence ,new-sequence
               :start1 ,start :end1 ,end)
           ,new-sequence)) => SUBSEQ

(defvar *xy* (make-array '(10 10)))
(defun xy (&key ((x x) 0) ((y y) 0)) (aref *xy* x y)) => XY
(defun set-xy (new-value &key ((x x) 0) ((y y) 0))
  (setf (aref *xy* x y) new-value)) => SET-XY
(defsetf xy (&key ((x x) 0) ((y y) 0)) (store)
  `(set-xy ,store 'x ,x 'y ,y)) => XY
(get-setf-expansion '(xy a b))
=> (#:t0 #:t1),
  (a b),
  (#:store),
  ((lambda (&key ((x #:x)) ((y #:y)))
    (set-xy #:store 'x #:x 'y #:y))
   #:t0 #:t1),
   (xy #:t0 #:t1)
  (xy 'x 1) => NIL
  (setf (xy 'x 1) 1) => 1
  (xy 'x 1) => 1
  (let ((a 'x) (b 'y))
    (setf (xy a 1 b 2) 3)
    (setf (xy b 5 a 9) 14)))
=> 14
  (xy 'y 0 'x 1) => 1
  (xy 'x 1 'y 2) => 3
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[documentation](#), [setf](#), [define-setf-expander](#), [get-setf-expansion](#), [Section 5.1 \(Generalized Reference\)](#), [Section 3.4.11 \(Syntactic Interaction of Documentation Strings and Declarations\)](#)

**Notes:**

*forms* must include provision for returning the correct value (the value or values of *store-variable*). This is handled by *forms* rather than by **defsetf** because in many cases this value can be returned at no extra cost, by calling a function that simultaneously stores into the *place* and returns the correct value.

A [setf](#) of a call on *access-fn* also evaluates all of *access-fn*'s arguments; it cannot treat any of them specially. This means that **defsetf** cannot be used to describe how to store into a [generalized reference](#) to a byte, such as (*ldb field reference*). [define-setf-expander](#) is used to handle situations that do not fit the restrictions imposed by **defsetf** and gives the user additional control.

**Macro DEFSTRUCT****Syntax:**

```
defstruct name-and-options [documentation] {slot-description}*  

=> structure-name  

name-and-options ::= structure-name | (structure-name [[options]])  

options ::= conc-name-option |  

         {constructor-option}* |  

         copier-option |  

         include-option |  

         initial-offset-option |  

         named-option |  

         predicate-option |  

         printer-option |  

         type-option  

conc-name-option ::= :conc-name | (:conc-name) | (:conc-name conc-name)  

constructor-option ::= :constructor |  

                     (:constructor) |  

                     (:constructor constructor-name) |  

                     (:constructor constructor-name constructor-arglist)  

copier-option ::= :copier | (:copier) | (:copier copier-name)  

predicate-option ::= :predicate | (:predicate) | (:predicate predicate-name)  

include-option ::= (:include included-structure-name {slot-description}*)  

printer-option ::= print-object-option | print-function-option  

print-object-option ::= (:print-object printer-name) | (:print-object)  

print-function-option ::= (:print-function printer-name) | (:print-function)  

type-option ::= (:type type)  

named-option ::= :named  

initial-offset-option ::= (:initial-offset initial-offset)  

slot-description ::= slot-name |  

                  (slot-name [slot-initform [[slot-option]]])  

slot-option ::= :type slot-type |  

              :read-only slot-read-only-p
```

**Arguments and Values:**

*conc-name*—a string designator.

## CLHS: Declaration DYNAMIC-EXTENT

*constructor-arglist*—a boa lambda list.

*constructor-name*—a symbol.

*copier-name*—a symbol.

*included-structure-name*—an already-defined structure name. Note that a derived type is not permissible, even if it would expand into a structure name.

*initial-offset*—a non-negative integer.

*predicate-name*—a symbol.

*printer-name*—a function name or a lambda expression.

*slot-name*—a symbol.

*slot-initform*—a form.

*slot-read-only-p*—a generalized boolean.

*structure-name*—a symbol.

*type*—one of the type specifiers **list**, **vector**, or **(vector size)**, or some other type specifier defined by the implementation to be appropriate.

*documentation*—a string; not evaluated.

### Description:

**defstruct** defines a structured type, named *structure-type*, with named slots as specified by the *slot-options*.

**defstruct** defines readers for the slots and arranges for setf to work properly on such reader functions. Also, unless overridden, it defines a predicate named *name-p*, defines a constructor function named *make-structure-name*, and defines a copier function named *copy-structure-name*. All names of automatically created functions might automatically be declared inline (at the discretion of the implementation).

If *documentation* is supplied, it is attached to *structure-name* as a documentation string of kind **structure**, and unless **:type** is used, the *documentation* is also attached to *structure-name* as a documentation string of kind **type** and as a documentation string to the class object for the class named *structure-name*.

**defstruct** defines a constructor function that is used to create instances of the structure created by **defstruct**. The default name is *make-structure-name*. A different name can be supplied by giving the name as the argument to the *constructor* option. **nil** indicates that no constructor function will be created.

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 form-1)
```

## CLHS: Declaration DYNAMIC-EXTENT

```
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. All the *keywords* and *forms* are evaluated. If a slot is not initialized in this way, it is initialized by evaluating *slot-initform* in the slot description at the time the constructor function is called. If no *slot-initform* is supplied, the consequences are undefined if an attempt is later made to read the slot's value before a value is explicitly assigned.

Each *slot-initform* supplied for a **defstruct** component, when used by the constructor function for an otherwise unsupplied component, is re-evaluated on every call to the constructor function. The *slot-initform* is not evaluated unless it is needed in the creation of a particular structure instance. If it is never needed, there can be no type-mismatch error, even if the *type* of the slot is specified; no warning should be issued in this case. For example, in the following sequence, only the last call is an error.

```
(defstruct person (name 007 :type string))
(make-person :name "James")
(make-person)
```

It is as if the *slot-initforms* were used as *initialization forms* for the *keyword parameters* of the constructor function.

The *symbols* which name the slots must not be used by the *implementation* as the *names* for the *lambda variables* in the constructor function, since one or more of those *symbols* might have been proclaimed **special** or might be defined as the name of a *constant variable*. The slot default init forms are evaluated in the *lexical environment* in which the **defstruct** form itself appears and in the *dynamic environment* in which the call to the constructor function appears.

For example, if the form (*gensym*) were used as an initialization form, either in the constructor-function call or as the default initialization form in **defstruct**, then every call to the constructor function would call **gensym** once to generate a new *symbol*.

Each *slot-description* in **defstruct** can specify zero or more *slot-options*. A *slot-option* consists of a pair of a keyword and a value (which is not a form to be evaluated, but the value itself). For example:

```
(defstruct ship
  (x-position 0.0 :type short-float)
  (y-position 0.0 :type short-float)
  (x-velocity 0.0 :type short-float)
  (y-velocity 0.0 :type short-float)
  (mass *default-ship-mass* :type short-float :read-only t))
```

This specifies that each slot always contains a *short float*, and that the last slot cannot be altered once a ship is constructed.

The available slot-options are:

**:type type**

This specifies that the contents of the slot is always of type *type*. This is entirely analogous to the declaration of a variable or function; it effectively declares the result type of the *reader* function. It is *implementation-dependent* whether the *type* is checked when initializing a slot or when assigning to it. *Type* is not evaluated; it must be a valid *type specifier*.

**:read-only x**

## CLHS: Declaration DYNAMIC-EXTENT

When *x* is *true*, this specifies that this slot cannot be altered; it will always contain the value supplied at construction time. **setf** will not accept the *reader* function for this slot. If *x* is *false*, this slot-option has no effect. *X* is not evaluated.

When this option is *false* or unsupplied, it is *implementation-dependent* whether the ability to *write* the slot is implemented by a *setf function* or a *setf expander*.

The following keyword options are available for use with **defstruct**. A **defstruct** option can be either a keyword or a *list* of a keyword and arguments for that keyword; specifying the keyword by itself is equivalent to specifying a list consisting of the keyword and no arguments. The syntax for **defstruct** options differs from the pair syntax used for slot-options. No part of any of these options is evaluated.

### :conc-name

This provides for automatic prefixing of names of *reader* (or *access*) functions. The default behavior is to begin the names of all the *reader* functions of a structure with the name of the structure followed by a hyphen.

:conc-name supplies an alternate prefix to be used. If a hyphen is to be used as a separator, it must be supplied as part of the prefix. If :conc-name is *nil* or no argument is supplied, then no prefix is used; then the names of the *reader* functions are the same as the slot names. If a *non-nil* prefix is given, the name of the *reader function* for each slot is constructed by concatenating that prefix and the name of the slot, and interning the resulting *symbol* in the *package* that is current at the time the **defstruct** form is expanded.

Note that no matter what is supplied for :conc-name, slot keywords that match the slot names with no prefix attached are used with a constructor function. The *reader* function name is used in conjunction with **setf**. Here is an example:

```
(defstruct (door (:conc-name dr-)) knob-color width material) => DOOR
(setq my-door (make-door :knob-color 'red :width 5.0))
=> #S(DOOR :KNOB-COLOR RED :WIDTH 5.0 :MATERIAL NIL)
(dr-width my-door) => 5.0
(setf (dr-width my-door) 43.7) => 43.7
(dr-width my-door) => 43.7
```

Whether or not the :conc-name option is explicitly supplied, the following rule governs name conflicts of generated *reader* (or *accessor*) names: For any *structure type* S1 having a *reader* function named R for a slot named X1 that is inherited by another *structure type* S2 that would have a *reader* function with the same name R for a slot named X2, no definition for R is generated by the definition of S2; instead, the definition of R is inherited from the definition of S1. (In such a case, if X1 and X2 are different slots, the *implementation* might signal a style warning.)

### :constructor

This option takes zero, one, or two arguments. If at least one argument is supplied and the first argument is not *nil*, then that argument is a *symbol* which specifies the name of the constructor function. If the argument is not supplied (or if the option itself is not supplied), the name of the constructor is produced by concatenating the string "MAKE-" and the name of the structure, interning the name in whatever *package* is current at the time **defstruct** is expanded. If the argument is provided and is *nil*, no constructor function is defined.

If :constructor is given as (:constructor *name arglist*), then instead of making a keyword driven constructor function, **defstruct** defines a "positional" constructor function, taking arguments whose meaning is determined by the argument's position and possibly by keywords. *Arglist* is used to

## CLHS: Declaration DYNAMIC-EXTENT

describe what the arguments to the constructor will be. In the simplest case something like (`:constructor make-foo (a b c)`) defines `make-foo` to be a three-argument constructor function whose arguments are used to initialize the slots named `a`, `b`, and `c`.

Because a constructor of this type operates "By Order of Arguments," it is sometimes known as a "boa constructor."

For information on how the *arglist* for a "boa constructor" is processed, see [Section 3.4.6 \(Boa Lambda Lists\)](#).

It is permissible to use the `:constructor` option more than once, so that you can define several different constructor functions, each taking different parameters.

**defstruct** creates the default-named keyword constructor function only if no explicit `:constructor` options are specified, or if the `:constructor` option is specified without a *name* argument.

(`:constructor nil`) is meaningful only when there are no other `:constructor` options specified. It prevents **defstruct** from generating any constructors at all.

Otherwise, **defstruct** creates a constructor function corresponding to each supplied `:constructor` option. It is permissible to specify multiple keyword constructor functions as well as multiple "boa constructors".

### `:copier`

This option takes one argument, a *symbol*, which specifies the name of the copier function. If the argument is not provided or if the option itself is not provided, the name of the copier is produced by concatenating the string "COPY-" and the name of the structure, interning the name in whatever *package* is current at the time **defstruct** is expanded. If the argument is provided and is `nil`, no copier function is defined.

The automatically defined copier function is a function of one *argument*, which must be of the structure type being defined. The copier function creates a *fresh* structure that has the same *type* as its *argument*, and that has the *same* component values as the original structure; that is, the component values are not copied recursively. If the **defstruct** `:type` option was not used, the following equivalence applies:

```
(copier-name x) = (copy-structure (the structure-name x))
```

### `:include`

This option is used for building a new structure definition as an extension of another structure definition. For example:

```
(defstruct person name age sex)
```

To make a new structure to represent an astronaut that has the attributes of name, age, and sex, and *functions* that operate on `person` structures, `astronaut` is defined with `:include` as follows:

```
(defstruct (astronaut (:include person)
                      (:conc-name astro-))
          helmet-size
          (favorite-beverage 'tang))
```

## CLHS: Declaration DYNAMIC-EXTENT

:include causes the structure being defined to have the same slots as the included structure. This is done in such a way that the *reader* functions for the included structure also work on the structure being defined. In this example, an astronaut therefore has five slots: the three defined in person and the two defined in astronaut itself. The *reader* functions defined by the person structure can be applied to instances of the astronaut structure, and they work correctly. Moreover, astronaut has its own *reader* functions for components defined by the person structure. The following examples illustrate the use of astronaut structures:

```
(setq x (make-astronaut :name 'buzz
                         :age 45.
                         :sex t
                         :helmet-size 17.5))
(person-name x) => BUZZ
(astro-name x) => BUZZ
(astro-favorite-beverage x) => TANG

(reduce #'+ astros :key #'person-age) ; obtains the total of the ages
; of the possibly empty
; sequence of astros
```

The difference between the *reader* functions person-name and astro-name is that person-name can be correctly applied to any person, including an astronaut, while astro-name can be correctly applied only to an astronaut. An implementation might check for incorrect use of *reader* functions.

At most one :include can be supplied in a single **defstruct**. The argument to :include is required and must be the name of some previously defined structure. If the structure being defined has no :type option, then the included structure must also have had no :type option supplied for it. If the structure being defined has a :type option, then the included structure must have been declared with a :type option specifying the same representation *type*.

If no :type option is involved, then the structure name of the including structure definition becomes the name of a *data type*, and therefore a valid *type specifier* recognizable by **typep**; it becomes a *subtype* of the included structure. In the above example, astronaut is a *subtype* of person; hence

```
(typep (make-astronaut) 'person) => true
```

indicating that all operations on persons also work on astronauts.

The structure using :include can specify default values or slot-options for the included slots different from those the included structure specifies, by giving the :include option as:

```
(:include included-structure-name slot-description*)
```

Each *slot-description* must have a *slot-name* that is the same as that of some slot in the included structure. If a *slot-description* has no *slot-initform*, then in the new structure the slot has no initial value. Otherwise its initial value form is replaced by the *slot-initform* in the *slot-description*. A normally writable slot can be made read-only. If a slot is read-only in the included structure, then it must also be so in the including structure. If a *type* is supplied for a slot, it must be a *subtype* of the *type* specified in the included structure.

For example, if the default age for an astronaut is 45, then

## CLHS: Declaration DYNAMIC-EXTENT

```
(defstruct (astronaut (:include person (age 45)))
  helmet-size
  (favorite-beverage 'tang))
```

If :include is used with the :type option, then the effect is first to skip over as many representation elements as needed to represent the included structure, then to skip over any additional elements supplied by the :initial-offset option, and then to begin allocation of elements from that point. For example:

```
(defstruct (binop (:type list) :named (:initial-offset 2))
  (operator '? :type symbol)
  operand-1
  operand-2) => BINOP
(defstruct (annotated-binop (:type list)
                           (:initial-offset 3)
                           (:include binop))
  commutative associative identity) => ANNOTATED-BINOP
(make-annotated-binop :operator '*'
  :operand-1 'x
  :operand-2 5
  :commutative t
  :associative t
  :identity 1)
=> (NIL NIL BINOP * X 5 NIL NIL NIL T T 1)
```

The first two nil elements stem from the :initial-offset of 2 in the definition of binop. The next four elements contain the structure name and three slots for binop. The next three nil elements stem from the :initial-offset of 3 in the definition of annotated-binop. The last three list elements contain the additional slots for an annotated-binop.

### :initial-offset

:initial-offset instructs **defstruct** to skip over a certain number of slots before it starts allocating the slots described in the body. This option's argument is the number of slots **defstruct** should skip. :initial-offset can be used only if :type is also supplied.

:initial-offset allows slots to be allocated beginning at a representational element other than the first. For example, the form

```
(defstruct (binop (:type list) (:initial-offset 2))
  (operator '? :type symbol)
  operand-1
  operand-2) => BINOP
```

would result in the following behavior for make-binop:

```
(make-binop :operator '+ :operand-1 'x :operand-2 5)
=> (NIL NIL + X 5)
(make-binop :operand-2 4 :operator '*')
=> (NIL NIL * NIL 4)
```

The selector functions binop-operator, binop-operand-1, and binop-operand-2 would be essentially equivalent to third, fourth, and fifth, respectively. Similarly, the form

```
(defstruct (binop (:type list) :named (:initial-offset 2))
  (operator '? :type symbol)
  operand-1
  operand-2) => BINOP
```

## CLHS: Declaration DYNAMIC-EXTENT

would result in the following behavior for `make-binop`:

```
(make-binop :operator '+ :operand-1 'x :operand-2 5) => (NIL NIL BINOP + X 5)
(make-binop :operand-2 4 :operator '* ) => (NIL NIL BINOP * NIL 4)
```

The first two nil elements stem from the `:initial-offset` of 2 in the definition of `binop`. The next four elements contain the structure name and three slots for `binop`.

`:named`

`:named` specifies that the structure is named. If no `:type` is supplied, then the structure is always named.

For example:

```
(defstruct (binop (:type list))
  (operator '? :type symbol)
  operand-1
  operand-2) => BINOP
```

This defines a constructor function `make-binop` and three selector functions, namely `binop-operator`, `binop-operand-1`, and `binop-operand-2`. (It does not, however, define a predicate `binop-p`, for reasons explained below.)

The effect of `make-binop` is simply to construct a list of length three:

```
(make-binop :operator '+ :operand-1 'x :operand-2 5) => (+ X 5)
(make-binop :operand-2 4 :operator '* ) => (* NIL 4)
```

It is just like the function `list` except that it takes keyword arguments and performs slot defaulting appropriate to the `binop` conceptual data type. Similarly, the selector functions `binop-operator`, `binop-operand-1`, and `binop-operand-2` are essentially equivalent to car, cadr, and caddr, respectively. They might not be completely equivalent because, for example, an implementation would be justified in adding error-checking code to ensure that the argument to each selector function is a length-3 list.

`binop` is a conceptual data type in that it is not made a part of the Common Lisp type system. typep does not recognize `binop` as a type specifier, and type-of returns `list` when given a `binop` structure. There is no way to distinguish a data structure constructed by `make-binop` from any other list that happens to have the correct structure.

There is not any way to recover the structure name `binop` from a structure created by `make-binop`. This can only be done if the structure is named. A named structure has the property that, given an instance of the structure, the structure name (that names the type) can be reliably recovered. For structures defined with no `:type` option, the structure name actually becomes part of the Common Lisp data-type system. type-of, when applied to such a structure, returns the structure name as the type of the object: typep recognizes the structure name as a valid type specifier.

For structures defined with a `:type` option, type-of returns a type specifier such as `list` or `(vector t)`, depending on the type supplied to the `:type` option. The structure name does not become a valid type specifier. However, if the `:named` option is also supplied, then the first component of the structure (as created by a defstruct constructor function) always contains the structure name. This allows the structure name to be recovered from an instance of the structure and allows a reasonable predicate for the conceptual type to be defined: the automatically defined `name-p`

## CLHS: Declaration DYNAMIC-EXTENT

predicate for the structure operates by first checking that its argument is of the proper type ([list](#), (`vector t`), or whatever) and then checking whether the first component contains the appropriate type name.

Consider the `binop` example shown above, modified only to include the `:named` option:

```
(defstruct (binop (:type list) :named)
  (operator '? :type symbol)
  operand-1
  operand-2) => BINOP
```

As before, this defines a constructor function `make-binop` and three selector functions `binop-operator`, `binop-operand-1`, and `binop-operand-2`. It also defines a predicate `binop-p`. The effect of `make-binop` is now to construct a list of length four:

```
(make-binop :operator '+ :operand-1 'x :operand-2 5) => (BINOP + X 5)
(make-binop :operand-2 4 :operator '* ) => (BINOP * NIL 4)
```

The structure has the same layout as before except that the structure name `binop` is included as the first list element. The selector functions `binop-operator`, `binop-operand-1`, and `binop-operand-2` are essentially equivalent to [`cadr`](#), [`caddr`](#), and [`cadddr`](#), respectively. The predicate `binop-p` is more or less equivalent to this definition:

```
(defun binop-p (x)
  (and (consp x) (eq (car x) 'binop))) => BINOP-P
```

The name `binop` is still not a valid [`type specifier`](#) recognizable to [`typep`](#), but at least there is a way of distinguishing `binop` structures from other similarly defined structures.

### `:predicate`

This option takes one argument, which specifies the name of the type predicate. If the argument is not supplied or if the option itself is not supplied, the name of the predicate is made by concatenating the name of the structure to the string "`-P`", interning the name in whatever [`package`](#) is current at the time [`defstruct`](#) is expanded. If the argument is provided and is [`nil`](#), no predicate is defined. A predicate can be defined only if the structure is named; if `:type` is supplied and `:named` is not supplied, then `:predicate` must either be unsupplied or have the value [`nil`](#).

### `:print-function`, `:print-object`

The `:print-function` and `:print-object` options specify that a [`print-object method`](#) for [`structures`](#) of type [`structure-name`](#) should be generated. These options are not synonyms, but do perform a similar service; the choice of which option (`:print-function` or `:print-object`) is used affects how the function named [`printer-name`](#) is called. Only one of these options may be used, and these options may be used only if `:type` is not supplied.

If the `:print-function` option is used, then when a structure of type [`structure-name`](#) is to be printed, the designated printer function is called on three [`arguments`](#):

- ◊ the structure to be printed (a [`generalized instance`](#) of [`structure-name`](#)).
- ◊ a [`stream`](#) to print to.
- ◊ an [`integer`](#) indicating the current depth. The magnitude of this integer may vary between [`implementations`](#); however, it can reliably be compared against [`\*print-level\*`](#) to determine whether depth abbreviation is appropriate.

Specifying (`:print-function printer-name`) is approximately equivalent to specifying:

## CLHS: Declaration DYNAMIC-EXTENT

```
(defmethod print-object ((object structure-name) stream)
  (funcall (function printer-name) object stream <<current-print-depth>>))
```

where the `<<current-print-depth>>` represents the printer's belief of how deep it is currently printing. It is implementation-dependent whether `<<current-print-depth>>` is always 0 and `*print-level*`, if non-nil, is re-bound to successively smaller values as printing descends recursively, or whether `current-print-depth` varies in value as printing descends recursively and `*print-level*` remains constant during the same traversal.

If the `:print-object` option is used, then when a structure of type `structure-name` is to be printed, the designated printer function is called on two arguments:

- ◊ the structure to be printed.
- ◊ the stream to print to.

Specifying `(:print-object printer-name)` is equivalent to specifying:

```
(defmethod print-object ((object structure-name) stream)
  (funcall (function printer-name) object stream))
```

If no `:type` option is supplied, and if either a `:print-function` or a `:print-object` option is supplied, and if no `printer-name` is supplied, then a print-object method specialized for `structure-name` is generated that calls a function that implements the default printing behavior for structures using #S notation; see [Section 22.1.3.12 \(Printing Structures\)](#).

If neither a `:print-function` nor a `:print-object` option is supplied, then defstruct does not generate a print-object method specialized for `structure-name` and some default behavior is inherited either from a structure named in an `:include` option or from the default behavior for printing structures; see the function print-object and [Section 22.1.3.12 \(Printing Structures\)](#).

When \*print-circle\* is `true`, a user-defined print function can print objects to the supplied `stream` using `write`, `prin1`, `princ`, or `format` and expect circularities to be detected and printed using the `#n#` syntax. This applies to methods on print-object in addition to `:print-function` options. If a user-defined print function prints to a `stream` other than the one that was supplied, then circularity detection starts over for that `stream`. See the variable \*print-circle\*.

### :type

`:type` explicitly specifies the representation to be used for the structure. Its argument must be one of these types:

#### vector

This produces the same result as specifying `(vector t)`. The structure is represented as a general vector, storing components as vector elements. The first component is vector element 1 if the structure is `:named`, and element 0 otherwise.

#### (vector element-type)

The structure is represented as a (possibly specialized) vector, storing components as vector elements. Every component must be of a type that can be stored in a vector of the type specified. The first component is vector element 1 if the structure is `:named`, and element 0 otherwise. The structure can be `:named` only if the type symbol is a subtype of the supplied element-type.

#### list

The structure is represented as a list. The first component is the cadr if the structure is

## CLHS: Declaration DYNAMIC-EXTENT

:named, and the car if it is not :named.

Specifying this option has the effect of forcing a specific representation and of forcing the components to be stored in the order specified in defstruct in corresponding successive elements of the specified representation. It also prevents the structure name from becoming a valid type specifier recognizable by typep.

For example:

```
(defstruct (quux (:type list) :named) x y)
```

should make a constructor that builds a list exactly like the one that list produces, with quux as its car.

If this type is defined:

```
(deftype quux () '(satisfies quux-p))
```

then this form

```
(typep (make-quux) 'quux)
```

should return precisely what this one does

```
(typep (list 'quux nil nil) 'quux)
```

If :type is not supplied, the structure is represented as an object of type structure-object.

defstruct without a :type option defines a class with the structure name as its name. The metaclass of structure instances is structure-class.

The consequences of redefining a defstruct structure are undefined.

In the case where no defstruct options have been supplied, the following functions are automatically defined to operate on instances of the new structure:

### Predicate

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. typep can 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).

### Component reader functions

Reader functions are defined to read the components of the structure. For each slot name, there is a corresponding reader function with the name structure-name-slot-name. This function reads the contents of that slot. Each reader function takes one argument, which is an instance of the structure type. setf can be used with any of these reader functions to alter the slot contents.

### Constructor function

A constructor function with the name make-structure-name is defined. This function creates and returns new instances of the structure type.

### Copier function

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

## CLHS: Declaration DYNAMIC-EXTENT

copier function creates a new structure with the same component entries as the original. Corresponding components of the two structure instances are eql.

If a defstruct form appears as a top level form, the compiler must make the structure type name recognized as a valid type name in subsequent declarations (as for deftype) and make the structure slot readers known to setf. In addition, the compiler must save enough information about the structure type so that further defstruct definitions can use :include in a subsequent deftype in the same file to refer to the structure type name. The functions which defstruct generates are not defined in the compile time environment, although the compiler may save enough information about the functions to code subsequent calls inline. The #S reader macro might or might not recognize the newly defined structure type name at compile time.

### Examples:

An example of a structure definition follows:

```
(defstruct ship
  x-position
  y-position
  x-velocity
  y-velocity
  mass)
```

This declares that every ship is an object with five named components. The evaluation of this form does the following:

1. It defines ship-x-position to be a function of one argument, a ship, that returns the x-position of the ship; ship-y-position and the other components are given similar function definitions. These functions are called the access functions, as they are used to access elements of the structure.
2. ship becomes the name of a type of which instances of ships are elements. ship becomes acceptable to typep, for example; (typep x 'ship) is true if x is a ship and false if x is any object other than a ship.
3. A function named ship-p of one argument is defined; it is a predicate that is true if its argument is a ship and is false otherwise.
4. A function called make-ship is defined that, when invoked, creates a data structure with five components, suitable for use with the access functions. Thus executing

```
(setq ship2 (make-ship))
```

sets ship2 to a newly created ship object. One can supply the initial values of any desired component in the call to make-ship by using keyword arguments in this way:

```
(setq ship2 (make-ship :mass *default-ship-mass*
                      :x-position 0
                      :y-position 0))
```

This constructs a new ship and initializes three of its components. This function is called the "constructor function" because it constructs a new structure.

5. A function called copy-ship of one argument is defined that, when given a ship object, creates a new ship object that is a copy of the given one. This function is called the "copier function."

setf can be used to alter the components of a ship:

## CLHS: Declaration DYNAMIC-EXTENT

```
(setf (ship-x-position ship2) 100)
```

This alters the x-position of ship2 to be 100. This works because **defstruct** behaves as if it generates an appropriate **defsetf** for each *access* function.

```
;;
;; Example 1
;; define town structure type
;; area, watertowers, firetrucks, population, elevation are its components
;;
(defstruct town
  area
  watertowers
  (firetrucks 1 :type fixnum)      ;an initialized slot
  population
  (elevation 5128 :read-only t)) ;a slot that can't be changed
=> TOWN
;create a town instance
(setq town1 (make-town :area 0 :watertowers 0)) => #S(TOWN...)
;town's predicate recognizes the new instance
(town-p town1) => true
;new town's area is as specified by make-town
(town-area town1) => 0
;new town's elevation has initial value
(town-elevation town1) => 5128
;setf recognizes reader function
(setf (town-population town1) 99) => 99
(town-population town1) => 99
;copier function makes a copy of town1
(setq town2 (copy-town town1)) => #S(TOWN...)
(= (town-population town1) (town-population town2)) => true
;since elevation is a read-only slot, its value can be set only
;when the structure is created
(setq town3 (make-town :area 0 :watertowers 3 :elevation 1200))
=> #S(TOWN...)
;;
;; Example 2
;; define clown structure type
;; this structure uses a nonstandard prefix
;;
(defstruct (clown (:conc-name bozo-))
  (nose-color 'red)
  frizzy-hair-p polkadots) => CLOWN
(setq funny-clown (make-clown)) => #S(CLOWN)
;use non-default reader name
(bozo-nose-color funny-clown) => RED
(defstruct (klown (:constructor make-up-klown)) ;similar def using other
  (:copier clone-klown)                      ;customizing keywords
  (:predicate is-a-bozo-p))
  nose-color frizzy-hair-p polkadots) => klown
;custom constructor now exists
(fboundp 'make-up-klown) => true
;;
;; 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)))
```

## CLHS: Declaration DYNAMIC-EXTENT

```
load-limit
  (axles 6)) =>  TRUCK
(setq x (make-truck :name 'mac :diesel t :load-limit 17))
=> #S(TRUCK...)
;vehicle readers work on trucks
(vehicle-name x)
=> MAC
;default taken from :include clause
(vehicle-year x)
=> 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...)
;:include default inherited
(pickup-year x) => 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
;a, b, and c set by position, and the rest are uninitialized
(setq x (make-dfs-boa 1 2 3)) => #(DFS-BOA...)
(dfs-boa-a x) => 1
;a and b set, c and f defaulted
(setq x (create-dfs-boa 1 2)) => #(DFS-BOA...)
(dfs-boa-b x) => 2
(eq (dfs-boa-c x) 'cc) => true
;a, b, and c set, and the rest are collected into d
(setq x (create-dfs-boa 1 2 3 4 5 6)) => #(DFS-BOA...)
(dfs-boa-d x) => (4 5 6)
```

**Affected By:** None.

**Exceptional Situations:**

If any two slot names (whether present directly or inherited by the :include option) are the same under string=, **defstruct** should signal an error of type program-error.

The consequences are undefined if the *included-structure-name* does not name a structure type.

**See Also:**

**documentation, print-object, setf, subtypep, type-of, typep**, Section 3.2 (Compilation)

**Notes:**

The *printer-name* should observe the values of such printer-control variables as **\*print-escape\***.

The restriction against issuing a warning for type mismatches between a *slot-initform* and the corresponding slot's :type option is necessary because a *slot-initform* must be specified in order to specify slot options; in some cases, no suitable default may exist.

The mechanism by which **defstruct** arranges for slot accessors to be usable with **setf** is *implementation-dependent*; for example, it may use *setf functions*, *setf expanders*, or some other *implementation-dependent* mechanism known to that *implementation's code* for **setf**.

## Macro DEFTYPE

### Syntax:

**deftype** *name lambda-list* [[*declaration\** / *documentation*]] *form\** => *name*

### Arguments and Values:

*name*—a *symbol*.

*lambda-list*—a *deftype lambda list*.

*declaration*—a **declare expression**; not evaluated.

*documentation*—a *string*; not evaluated.

*form*—a *form*.

### Description:

**deftype** defines a *derived type specifier* named *name*.

The meaning of the new *type specifier* is given in terms of a function which expands the *type specifier* into another *type specifier*, which itself will be expanded if it contains references to another *derived type specifier*.

The newly defined *type specifier* may be referenced as a list of the form (*name arg1 arg2 ...*). The number of arguments must be appropriate to the *lambda-list*. If the new *type specifier* takes no arguments, or if all of its arguments are optional, the *type specifier* may be used as an *atomic type specifier*.

The *argument expressions* to the *type specifier*, *arg1 ... argn*, are not *evaluated*. Instead, these *literal objects* become the *objects* to which corresponding *parameters* become *bound*.

The body of the **deftype form** (but not the *lambda-list*) is implicitly enclosed in a *block* named *name*, and is evaluated as an *implicit progn*, returning a new *type specifier*.

The *lexical environment* of the body is the one which was current at the time the **deftype** form was evaluated, augmented by the *variables* in the *lambda-list*.

Recursive expansion of the *type specifier* returned as the expansion must terminate, including the expansion of *type specifiers* which are nested within the expansion.

The consequences are undefined if the result of fully expanding a *type specifier* contains any circular structure, except within the *objects* referred to by **member** and **eql type specifiers**.

*Documentation* is attached to *name* as a *documentation string* of kind **type**.

## CLHS: Declaration DYNAMIC-EXTENT

If a **deftype form** appears as a *top level form*, the *compiler* must ensure that the *name* is recognized in subsequent *type declarations*. The *programmer* must ensure that the body of a **deftype** form can be *evaluated* at compile time if the *name* is referenced in subsequent *type declarations*. If the expansion of a *type specifier* is not defined fully at compile time (perhaps because it expands into an unknown *type specifier* or a **satisfies** of a named *function* that isn't defined in the compile-time environment), an *implementation* may ignore any references to this *type* in declarations and/or signal a warning.

### Examples:

```
(defun equidimensional (a)
  (or (< (array-rank a) 2)
      (apply #'= (array-dimensions a)))) => EQUIDIMENSIONAL
(deftype square-matrix (&optional type size)
  `(and (array ,type (,size ,size))
        (satisfies equidimensional))) => SQUARE-MATRIX
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[declare](#), [defmacro](#), [documentation](#), [Section 4.2.3 \(Type Specifiers\)](#), [Section 3.4.11 \(Syntactic Interaction of Documentation Strings and Declarations\)](#)

**Notes:** None.

## Macro DEFUN

### Syntax:

**defun** *function-name lambda-list [[declaration\* / documentation]] form\**

*=> function-name*

### Arguments and Values:

*function-name*—a *function name*.

*lambda-list*—an *ordinary lambda list*.

*declaration*—a **declare expression**; not evaluated.

*documentation*—a *string*; not evaluated.

*forms*—an *implicit progn*.

*block-name*—the *function block name* of the *function-name*.

### Description:

## CLHS: Declaration DYNAMIC-EXTENT

Defines a new function named *function-name* in the global environment. The body of the function defined by defun consists of *forms*; they are executed as an implicit progn when the function is called. defun can be used to define a new function, to install a corrected version of an incorrect definition, to redefine an already-defined function, or to redefine a macro as a function.

defun implicitly puts a block named *block-name* around the body *forms* (but not the forms in the lambda-list) of the function defined.

*Documentation* is attached as a documentation string to *name* (as kind function) and to the function object.

Evaluating defun causes *function-name* to be a global name for the function specified by the lambda expression.

```
(lambda lambda-list
  [[declaration* | documentation]]      (block block-name form*))
```

processed in the lexical environment in which defun was executed.

(None of the arguments are evaluated at macro expansion time.)

defun is not required to perform any compile-time side effects. In particular, defun does not make the function definition available at compile time. An implementation may choose to store information about the function for the purposes of compile-time error-checking (such as checking the number of arguments on calls), or to enable the function to be expanded inline.

### Examples:

```
(defun recur (x)
  (when (> x 0)
    (recur (1- x)))) =>  RECUR
(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)

;; This function assumes its callers have checked the types of the
;; arguments, and authorizes the compiler to build in that assumption.
(defun discriminant (a b c)
  (declare (number a b c))
  "Compute the discriminant for a quadratic equation."
  (- (* b b) (* 4 a c))) =>  DISCRIMINANT
(discriminant 1 2/3 -2) =>  76/9

;; This function assumes its callers have not checked the types of the
;; arguments, and performs explicit type checks before making any assumptions.
(defun careful-discriminant (a b c)
  "Compute the discriminant for a quadratic equation."
  (check-type a number)
  (check-type b number)
  (check-type c number)
  (locally (declare (number a b c))
    (- (* b b) (* 4 a c)))) =>  CAREFUL-DISCRIMINANT
(careful-discriminant 1 2/3 -2) =>  76/9
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[\*\*flet\*\*](#), [\*\*labels\*\*](#), [\*\*block\*\*](#), [\*\*return-from\*\*](#), [\*\*declare\*\*](#), [\*\*documentation\*\*](#), [Section 3.1 \(Evaluation\)](#), [Section 3.4.1 \(Ordinary Lambda Lists\)](#), [Section 3.4.11 \(Syntactic Interaction of Documentation Strings and Declarations\)](#)

**Notes:**

**return-from** can be used to return prematurely from a *function* defined by **defun**.

Additional side effects might take place when additional information (typically debugging information) about the function definition is recorded.

## Macro DESTRUCTURING-BIND

**Syntax:**

**destructuring-bind** *lambda-list expression declaration\* form\**

=> *result\**

**Arguments and Values:**

*lambda-list*—a [destructuring lambda list](#).

*expression*—a [form](#).

*declaration*—a [declare expression](#); not evaluated.

*forms*—an [implicit progn](#).

*results*—the [values](#) returned by the [forms](#).

**Description:**

**destructuring-bind** binds the variables specified in *lambda-list* to the corresponding values in the tree structure resulting from the evaluation of *expression*; then **destructuring-bind** evaluates *forms*.

The *lambda-list* supports destructuring as described in [Section 3.4.5 \(Destructuring Lambda Lists\)](#).

**Examples:**

```
(defun iota (n) (loop for i from 1 to n collect i)) ;helper
(destructuring-bind ((a &optional (b 'bee)) one two three)
  `((alpha) ,@(iota 3))
  (list a b three two one)) => (ALPHA BEE 3 2 1)
```

**Affected By:** None.

**Exceptional Situations:**

If the result of evaluating the *expression* does not match the destructuring pattern, an error of [type error](#) should be signaled.

**See Also:**[macrolet](#), [defmacro](#)

**Notes:** None.

**Macro DO, DO\*****Syntax:**

**do** ({var / (var [init-form [step-form]]})\*) (end-test-form result-form\*) declaration\* {tag / statement}\*  
=> result\*

**do\*** ({var / (var [init-form [step-form]]})\*) (end-test-form result-form\*) declaration\* {tag / statement}\*  
=> result\*

**Arguments and Values:**

*var*—a [symbol](#).

*init-form*—a [form](#).

*step-form*—a [form](#).

*end-test-form*—a [form](#).

*result-forms*—an [implicit progn](#).

*declaration*—a [declare expression](#); not evaluated.

*tag*—a [go tag](#); not evaluated.

*statement*—a [compound form](#); evaluated as described below.

*results*—if a [return](#) or [return-from](#) form is executed, the [values](#) passed from that [form](#); otherwise, the [values](#) returned by the *result-forms*.

**Description:**

**do** iterates over a group of *statements* while a test condition holds. **do** accepts an arbitrary number of iteration *vars* which are bound within the iteration and stepped in parallel. An initial value may be supplied for each iteration variable by use of an *init-form*. *Step-forms* may be used to specify how the *vars* should be updated on succeeding iterations through the loop. *Step-forms* may be used both to generate successive values or to

## CLHS: Declaration DYNAMIC-EXTENT

accumulate results. If the *end-test-form* condition is met prior to an execution of the body, the iteration terminates. *Tags* label *statements*.

**do\*** is exactly like **do** except that the *bindings* and steppings of the *vars* are performed sequentially rather than in parallel.

Before the first iteration, all the *init-forms* are evaluated, and each *var* is bound to the value of its respective *init-form*, if supplied. This is a *binding*, not an assignment; when the loop terminates, the old values of those variables will be restored. For **do**, all of the *init-forms* are evaluated before any *var* is bound. The *init-forms* can refer to the *bindings* of the *vars* visible before beginning execution of **do**. For **do\***, the first *init-form* is evaluated, then the first *var* is bound to that value, then the second *init-form* is evaluated, then the second *var* is bound, and so on; in general, the *k*th *init-form* can refer to the new binding of the *j*th *var* if *j* < *k*, and otherwise to the old binding of the *j*th *var*.

At the beginning of each iteration, after processing the variables, the *end-test-form* is evaluated. If the result is *false*, execution proceeds with the body of the **do** (or **do\***) form. If the result is *true*, the *result-forms* are evaluated in order as an *implicit progn*, and then **do** or **do\*** returns.

At the beginning of each iteration other than the first, *vars* are updated as follows. All the *step-forms*, if supplied, are evaluated, from left to right, and the resulting values are assigned to the respective *vars*. Any *var* that has no associated *step-form* is not assigned to. For **do**, all the *step-forms* are evaluated before any *var* is updated; the assignment of values to *vars* is done in parallel, as if by **psetq**. Because all of the *step-forms* are evaluated before any of the *vars* are altered, a *step-form* when evaluated always has access to the old values of all the *vars*, even if other *step-forms* precede it. For **do\***, the first *step-form* is evaluated, then the value is assigned to the first *var*, then the second *step-form* is evaluated, then the value is assigned to the second *var*, and so on; the assignment of values to variables is done sequentially, as if by **setq**. For either **do** or **do\***, after the *vars* have been updated, the *end-test-form* is evaluated as described above, and the iteration continues.

The remainder of the **do** (or **do\***) form constitutes an *implicit tagbody*. *Tags* may appear within the body of a **do** loop for use by **go** statements appearing in the body (but such **go** statements may not appear in the variable specifiers, the *end-test-form*, or the *result-forms*). When the end of a **do** body is reached, the next iteration cycle (beginning with the evaluation of *step-forms*) occurs.

An *implicit block* named **nil** surrounds the entire **do** (or **do\***) form. A **return** statement may be used at any point to exit the loop immediately.

*Init-form* is an initial value for the *var* with which it is associated. If *init-form* is omitted, the initial value of *var* is **nil**. If a *declaration* is supplied for a *var*, *init-form* must be consistent with the *declaration*.

*Declarations* can appear at the beginning of a **do** (or **do\***) body. They apply to code in the **do** (or **do\***) body, to the *bindings* of the **do** (or **do\***) *vars*, to the *step-forms*, to the *end-test-form*, and to the *result-forms*.

### Examples:

```
(do ((temp-one 1 (1+ temp-one))
     (temp-two 0 (1- temp-two)))
    ((> (- temp-one temp-two) 5) temp-one)) => 4

(do ((temp-one 1 (1+ temp-one))
     (temp-two 0 (1+ temp-one)))
    ((= 3 temp-two) temp-one)) => 3
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(do* ((temp-one 1 (1+ temp-one))
      (temp-two 0 (1+ temp-one)))
     (= 3 temp-two) temp-one)) => 2

(do ((j 0 (+ j 1)))
    (nil) ;Do forever.
    (format t "~%Input ~D:" j)
    (let ((item (read)))
      (if (null item) (return) ;Process items until NIL seen.
          (format t "~&Output ~D: ~S" j item)))
  >> Input 0: banana
  >> Output 0: BANANA
  >> Input 1: (57 boxes)
  >> Output 1: (57 BOXES)
  >> Input 2: NIL
=> NIL

(setq a-vector (vector 1 nil 3 nil))
(do ((i 0 (+ i 1))) ;Sets every null element of a-vector to zero.
    (n (array-dimension a-vector 0)))
    ((= i n))
    (when (null (aref a-vector i))
      (setf (aref a-vector i) 0))) => NIL
a-vector => #(1 0 3 0)

(do ((x e (cdr x)))
    (oldx x x))
    ((null x))
  body)
```

is an example of parallel assignment to index variables. On the first iteration, the value of `oldx` is whatever value `x` had before the `do` was entered. On succeeding iterations, `oldx` contains the value that `x` had on the previous iteration.

```
(do ((x foo (cdr x))
      (y bar (cdr y))
      (z '() (cons (f (car x) (car y)) z)))
     ((or (null x) (null y))
      (nreverse z)))
```

does the same thing as `(mapcar #'f foo bar)`. The step computation for `z` is an example of the fact that variables are stepped in parallel. Also, the body of the loop is empty.

```
(defun list-reverse (list)
  (do ((x list (cdr x))
        (y '() (cons (car x) y)))
       ((endp x) y)))
```

As an example of nested iterations, consider a data structure that is a list of conses. The car of each cons is a list of symbols, and the cdr of each cons is a list of equal length containing corresponding values. Such a data structure is similar to an association list, but is divided into "frames"; the overall structure resembles a rib-cage. A lookup function on such a data structure might be:

```
(defun ribcage-lookup (sym ribcage)
  (do ((r ribcage (cdr r)))
       ((null r) nil)
     (do ((s (caar r) (cdr s))
          (v (cdar r) (cdr v))))
```

## CLHS: Declaration DYNAMIC-EXTENT

```
((null s))
  (when (eq (car s) sym)
    (return-from ribcage-lookup (car v))))) => RIBCAGE-LOOKUP
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

other iteration functions ([dolist](#), [dotimes](#), and [loop](#)) and more primitive functionality ([tagbody](#), [go](#), [block](#), [return](#), [let](#), and [setq](#))

**Notes:**

If *end-test-form* is [nil](#), the test will never succeed. This provides an idiom for "do forever": the body of the [do](#) or [do\\*](#) is executed repeatedly. The infinite loop can be terminated by the use of [return](#), [return-from](#), [go](#) to an outer level, or [throw](#).

A [do form](#) may be explained in terms of the more primitive [forms](#) [block](#), [return](#), [let](#), [loop](#), [tagbody](#), and [psetq](#) as follows:

```
(block nil
  (let ((var1 init1)
        (var2 init2)
        ...
        (varn initn))
    declarations
    (loop (when end-test (return (progn . result)))
          (tagbody . tagbody)
          (psetq var1 step1
                 var2 step2
                 ...
                 varn stepn))))
```

[do\\*](#) is similar, except that [let\\*](#) and [setq](#) replace the [let](#) and [psetq](#), respectively.

## Macro DO-SYMBOLS, DO-EXTERNAL-SYMBOLS, DO-ALL-SYMBOLS

**Syntax:**

**do-symbols** (*var* [*package* [*result-form*]]) *declaration\** {*tag / statement*}\*

=> *result\**

**do-external-symbols** (*var* [*package* [*result-form*]]) *declaration\** {*tag / statement*}\*

=> *result\**

**do-all-symbols** (*var* [*result-form*]) *declaration\** {*tag / statement*}\*

=> *result*\*

### Arguments and Values:

*var*—a *variable name*; not evaluated.

*package*—a *package designator*; evaluated. The default in **do-symbols** and **do-external-symbols** is the *current package*.

*result-form*—a *form*; evaluated as described below. The default is **nil**.

*declaration*—a *declare expression*; not evaluated.

*tag*—a *go tag*; not evaluated.

*statement*—a *compound form*; evaluated as described below.

*results*—the *values* returned by the *result-form* if a *normal return* occurs, or else, if an *explicit return* occurs, the *values* that were transferred.

### Description:

**do-symbols**, **do-external-symbols**, and **do-all-symbols** iterate over the *symbols* of *packages*. For each *symbol* in the set of *packages* chosen, the *var* is bound to the *symbol*, and the *statements* in the body are executed. When all the *symbols* have been processed, *result-form* is evaluated and returned as the value of the macro.

**do-symbols** iterates over the *symbols accessible* in *package*. *Statements* may execute more than once for *symbols* that are inherited from multiple *packages*.

**do-all-symbols** iterates on every *registered package*. **do-all-symbols** will not process every *symbol* whatsoever, because a *symbol* not *accessible* in any *registered package* will not be processed.

**do-all-symbols** may cause a *symbol* that is *present* in several *packages* to be processed more than once.

**do-external-symbols** iterates on the external symbols of *package*.

When *result-form* is evaluated, *var* is bound and has the value **nil**.

An *implicit block* named **nil** surrounds the entire **do-symbols**, **do-external-symbols**, or **do-all-symbols form**. **return** or **return-from** may be used to terminate the iteration prematurely.

If execution of the body affects which *symbols* are contained in the set of *packages* over which iteration is occurring, other than to remove the *symbol* currently the value of *var* by using **unintern**, the consequences are undefined.

For each of these macros, the *scope* of the name binding does not include any initial value form, but the optional result forms are included.

Any *tag* in the body is treated as with **tagbody**.

### Examples:

## CLHS: Declaration DYNAMIC-EXTENT

```
(make-package 'temp :use nil) => #<PACKAGE "TEMP">
(intern "SHY" 'temp) => TEMP::SHY, NIL ;SHY will be an internal symbol
                           ;in the package TEMP
(export (intern "BOLD" 'temp) 'temp) => T ;BOLD will be external
(let ((lst ()))
  (do-symbols (s (find-package 'temp)) (push s lst))
  lst)
=> (TEMP::SHY TEMP:BOLD)
OR=> (TEMP:BOLD TEMP::SHY)
(let ((lst ()))
  (do-external-symbols (s (find-package 'temp) lst) (push s lst)))
  lst)
=> (TEMP:BOLD)
(let ((lst ()))
  (do-all-symbols (s lst)
    (when (eq (find-package 'temp) (symbol-package s)) (push s lst)))
  lst)
=> (TEMP::SHY TEMP:BOLD)
OR=> (TEMP:BOLD TEMP::SHY)
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[intern, export](#), Section 3.6 (Traversal Rules and Side Effects)

**Notes:** None.

## Macro DOLIST

**Syntax:**

**dolist** (*var list-form [result-form]*) *declaration\** {*tag / statement*}\*

=> *result\**

**Arguments and Values:**

*var*---a symbol.

*list-form*---a form.

*result-form*---a form.

*declaration*---a declare expression; not evaluated.

*tag*---a go tag; not evaluated.

*statement*---a compound form; evaluated as described below.

## CLHS: Declaration DYNAMIC-EXTENT

*results*—if a **return** or **return-from** form is executed, the *values* passed from that *form*; otherwise, the *values* returned by the *result-form* or **nil** if there is no *result-form*.

### Description:

**dolist** iterates over the elements of a *list*. The body of **dolist** is like a **tagbody**. It consists of a series of *tags* and *statements*.

**dolist** evaluates *list-form*, which should produce a *list*. It then executes the body once for each element in the *list*, in the order in which the *tags* and *statements* occur, with *var* bound to the element. Then *result-form* is evaluated. *tags* label *statements*.

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

An *implicit block* named **nil** surrounds **dolist**. **return** may be used to terminate the loop immediately without performing any further iterations, returning zero or more *values*.

The *scope* of the binding of *var* does not include the *list-form*, but the *result-form* is included.

It is *implementation-dependent* whether **dolist** *establishes* a new *binding* of *var* on each iteration or whether it *establishes* a binding for *var* once at the beginning and then *assigns* it on any subsequent iterations.

### Examples:

```
(setq temp-two '()) => NIL
(dolist (temp-one '(1 2 3 4) temp-two) (push temp-one temp-two)) => (4 3 2 1)

(setq temp-two 0) => 0
(dolist (temp-one '(1 2 3 4)) (incf temp-two)) => NIL
temp-two => 4

(dolist (x '(a b c d)) (print1 x) (princ " "))
>> A B C D
=> NIL
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

### See Also:

**do**, **dotimes**, **tagbody**, Section 3.6 (Traversal Rules and Side Effects)

### Notes:

**go** may be used within the body of **dolist** to transfer control to a statement labeled by a *tag*.

## Macro DOTIMES

### Syntax:

**dotimes** (*var count-form [result-form]*) *declaration\** {*tag / statement*}\*  
*=> result\**

### Arguments and Values:

*var*—a symbol.

*count-form*—a form.

*result-form*—a form.

*declaration*—a declare expression; not evaluated.

*tag*—a go tag; not evaluated.

*statement*—a compound form; evaluated as described below.

*results*—if a return or return-from form is executed, the values passed from that form; otherwise, the values returned by the *result-form* or nil if there is no *result-form*.

### Description:

**dotimes** iterates over a series of integers.

**dotimes** evaluates *count-form*, which should produce an integer. If *count-form* is zero or negative, the body is not executed. **dotimes** then executes the body once for each integer from 0 up to but not including the value of *count-form*, in the order in which the *tags* and *statements* occur, with *var* bound to each integer. Then *result-form* is evaluated. At the time *result-form* is processed, *var* is bound to the number of times the body was executed. *Tags* label *statements*.

An implicit block named nil surrounds **dotimes**. return may be used to terminate the loop immediately without performing any further iterations, returning zero or more values.

The body of the loop is an implicit tagbody; it may contain tags to serve as the targets of go statements. Declarations may appear before the body of the loop.

The scope of the binding of *var* does not include the *count-form*, but the *result-form* is included.

It is implementation-dependent whether **dotimes** establishes a new binding of *var* on each iteration or whether it establishes a binding for *var* once at the beginning and then assigns it on any subsequent iterations.

### Examples:

```
(dotimes (temp-one 10 temp-one)) => 10
(setq temp-two 0) => 0
(dotimes (temp-one 10 t) (incf temp-two)) => T
```

```
temp-two => 10
```

Here is an example of the use of `dotimes` in processing strings:

```
; ; True if the specified subsequence of the string is a
; ; palindrome (reads the same forwards and backwards).
(defun palindromep (string &optional
                      (start 0)
                      (end (length string)))
  (dotimes (k (floor (- end start) 2) t)
    (unless (char-equal (char string (+ start k))
                         (char string (- end k 1)))
      (return nil))))
(palindromep "Able was I ere I saw Elba") => T
(palindromep "A man, a plan, a canal--Panama!") => NIL
(remove-if-not #'alpha-char-p ;Remove punctuation.
               "A man, a plan, a canal--Panama!")
=> "AmanaplanacanalPanama"
(palindromep
  (remove-if-not #'alpha-char-p
                 "A man, a plan, a canal--Panama!")) => T
(palindromep
  (remove-if-not
   #'alpha-char-p
   "Unremarkable was I ere I saw Elba Kramer, nu?")) => T
(palindromep
  (remove-if-not
   #'alpha-char-p
   "A man, a plan, a cat, a ham, a yak,
   a yam, a hat, a canal--Panama!")) => T
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**do, dolist, tagbody**

**Notes:**

**go** may be used within the body of `dotimes` to transfer control to a statement labeled by a *tag*.

## Macro FORMATTER

**Syntax:**

**formatter** *control-string* => *function*

**Arguments and Values:**

*control-string*—a *format string*; not evaluated.

*function*—a function.

### Description:

Returns a function which has behavior equivalent to:

```
#'(lambda (*standard-output* &rest arguments)
  (apply #'format t control-string arguments)
  arguments-tail)
```

where *arguments-tail* is either the tail of *arguments* which has as its car the argument that would be processed next if there were more format directives in the *control-string*, or else nil if no more *arguments* follow the most recently processed argument.

### Examples:

```
(funcall (formatter "~&~A~A") *standard-output* 'a 'b 'c)
>> AB
=> (C)

(format t (formatter "~&~A~A") 'a 'b 'c)
>> AB
=> NIL
```

**Side Effects:** None.

**Affected By:** None.

### Exceptional Situations:

Might signal an error (at macro expansion time or at run time) if the argument is not a valid format string.

### See Also:

**format**

**Notes:** None.

## Macro HANDLER-CASE

### Syntax:

**handler-case** *expression* [[{*error-clause*}\* / *no-error-clause*]] => *result*\*

```
clause ::= error-clause | no-error-clause
error-clause ::= (typespec ([var]) declaration* form*)
no-error-clause ::= (:no-error lambda-list declaration* form*)
```

### Arguments and Values:

*expression*—a form.

*typespec*—a type specifier.

*var*—a variable name.

*lambda-list*—an ordinary lambda list.

*declaration*—a declare expression; not evaluated.

*form*—a form.

*results*—In the normal situation, the values returned are those that result from the evaluation of *expression*; in the exceptional situation when control is transferred to a *clause*, the value of the last *form* in that *clause* is returned.

### Description:

**handler-case** executes *expression* in a dynamic environment where various handlers are active. Each *error-clause* specifies how to handle a condition matching the indicated *typespec*. A *no-error-clause* allows the specification of a particular action if control returns normally.

If a condition is signaled for which there is an appropriate *error-clause* during the execution of *expression* (i.e., one for which (*typep condition 'typespec*) returns *true*) and if there is no intervening handler for a condition of that *type*, then control is transferred to the body of the relevant *error-clause*. In this case, the dynamic state is unwound appropriately (so that the handlers established around the *expression* are no longer active), and *var* is bound to the condition that had been signaled. If more than one case is provided, those cases are made accessible in parallel. That is, in

```
(handler-case form
  (typespec1 (var1) form1)
  (typespec2 (var2) form2))
```

if the first *clause* (containing *form1*) has been selected, the handler for the second is no longer visible (or vice versa).

The *clauses* are searched sequentially from top to bottom. If there is *type* overlap between *typespecs*, the earlier of the *clauses* is selected.

If *var* is not needed, it can be omitted. That is, a *clause* such as:

```
(typespec (var) (declare (ignore var)) form)
```

can be written (*typespec () form*).

If there are no *forms* in a selected *clause*, the case, and therefore **handler-case**, returns nil. If execution of *expression* returns normally and no *no-error-clause* exists, the values returned by *expression* are returned by **handler-case**. If execution of *expression* returns normally and a *no-error-clause* does exist, the values returned are used as arguments to the function described by constructing (*lambda lambda-list form\**) from the *no-error-clause*, and the values of that function call are returned by **handler-case**. The handlers which were established around the *expression* are no longer active at the time of this call.

### Examples:

## CLHS: Declaration DYNAMIC-EXTENT

```
(defun assess-condition (condition)
  (handler-case (signal condition)
    (warning () "Lots of smoke, but no fire.")
    ((or arithmetic-error control-error cell-error stream-error)
     (condition)
     (format nil "~S looks especially bad." condition))
    (serious-condition (condition)
      (format nil "~S looks serious." condition))
    (condition () "Hardly worth mentioning."))
=> ASSESS-CONDITION
  (assess-condition (make-condition 'stream-error :stream *terminal-io*))
=> "#<STREAM-ERROR 12352256> looks especially bad."
(define-condition random-condition (condition) ()
  (:report (lambda (condition stream)
    (declare (ignore condition))
    (princ "Yow" stream))))
=> RANDOM-CONDITION
  (assess-condition (make-condition 'random-condition))
=> "Hardly worth mentioning."
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[handler-bind, ignore-errors, Section 9.1 \(Condition System Concepts\)](#)

**Notes:**

```
(handler-case form
  (type1 (var1) . body1)
  (type2 (var2) . body2) ...)
```

is approximately equivalent to:

```
(block #1=#:g0001
  (let ((#2=#:g0002 nil))
    (tagbody
      (handler-bind ((type1 #'(lambda (temp)
        (setq #1# temp)
        (go #3=#:g0003)))
        (type2 #'(lambda (temp)
        (setq #2# temp)
        (go #4=#:g0004))) ...)

      (return-from #1# form)
      #3# (return-from #1# (let ((var1 #2#)) . body1))
      #4# (return-from #1# (let ((var2 #2#)) . body2)) ...))

  (handler-case form
    (type1 (var1) . body1)
    ...
    (:no-error (varN-1 varN-2 ...) . bodyN))
```

is approximately equivalent to:

```
(block #1=#:error-return
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(multiple-value-call #'(lambda (varN-1 varN-2 ...) . bodyN)
  (block #2=#:normal-return
    (return-from #1#
      (handler-case (return-from #2# form)
        (type1 (var1) . body1) ...))))
```

## Macro HANDLER-BIND

### Syntax:

**handler-bind** (*{binding}\* form\** => *result\**)

*binding*::= (type *handler*)

### Arguments and Values:

*type*—a type specifier.

*handler*—a form; evaluated to produce a handler-function.

*handler-function*—a designator for a function of one argument.

*forms*—an implicit progn.

*results*—the values returned by the forms.

### Description:

Executes *forms* in a dynamic environment where the indicated handler bindings are in effect.

Each *handler* should evaluate to a handler-function, which is used to handle conditions of the given *type* during execution of the *forms*. This function should take a single argument, the condition being signaled.

If more than one handler binding is supplied, the handler bindings are searched sequentially from top to bottom in search of a match (by visual analogy with typecase). If an appropriate *type* is found, the associated handler is run in a dynamic environment where none of these handler bindings are visible (to avoid recursive errors). If the handler declines, the search continues for another handler.

If no appropriate handler is found, other handlers are sought from dynamically enclosing contours. If no handler is found outside, then signal returns or error enters the debugger.

### Examples:

In the following code, if an unbound variable error is signaled in the body (and not handled by an intervening handler), the first function is called.

```
(handler-bind ((unbound-variable #'(lambda ...))
  (error #'(lambda ...)))
  ...)
```

If any other kind of error is signaled, the second function is called. In either case, neither handler is active while executing the code in the associated function.

## CLHS: Declaration DYNAMIC-EXTENT

```
(defun trap-error-handler (condition)
  (format *error-output* "~~A~&" condition)
  (throw 'trap-errors nil))

(defmacro trap-errors (&rest forms)
  `(catch 'trap-errors
    (handler-bind ((error #'trap-error-handler))
      ,@forms)))

(list (trap-errors (signal "Foo.") 1)
      (trap-errors (error "Bar.") 2)
      (+ 1 2))
>> Bar.
=> (1 NIL 3)
```

Note that "Foo." is not printed because the condition made by signal is a *simple condition*, which is not of type error, so it doesn't trigger the handler for error set up by trap-errors.

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[handler-case](#)

**Notes:** None.

## Macro IGNORE-ERRORS

**Syntax:**

ignore-errors *form\** => *result\**

**Arguments and Values:**

*forms*—an implicit progn.

*results*—In the normal situation, the values of the forms are returned; in the exceptional situation, two values are returned: nil and the condition.

**Description:**

ignore-errors is used to prevent conditions of type error from causing entry into the debugger.

Specifically, ignore-errors *executes forms* in a dynamic environment where a handler for conditions of type error has been established; if invoked, it handles such conditions by returning two values, nil and the condition that was signaled, from the ignore-errors form.

If a normal return from the forms occurs, any values returned are returned by ignore-errors.

**Examples:**

```
(defun load-init-file (program)
  (let ((win nil))
    (ignore-errors ;if this fails, don't enter debugger
      (load (merge-pathnames (make-pathname :name program :type :lisp)
                            (user-homedir-pathname))))
    (setq win t))
  (unless win (format t "~&Init file failed to load.~%"))
  win)

(load-init-file "no-such-program")
>> Init file failed to load.
NIL
```

**Affected By:** None.**Exceptional Situations:** None.**See Also:**[handler-case, Section 9.1 \(Condition System Concepts\)](#)**Notes:**

(ignore-errors . forms)

is equivalent to:

```
(handler-case (progn . forms)
  (error (condition) (values nil condition)))
```

Because the second return value is a *condition* in the exceptional case, it is common (but not required) to arrange for the second return value in the normal case to be missing or **nil** so that the two situations can be distinguished.

**Macro IN-PACKAGE****Syntax:****in-package** *name* => *package***Arguments and Values:***name*---a *string designator*; not evaluated.*package*---the *package* named by *name*.**Description:**

Causes the the *package* named by *name* to become the *current package*---that is, the *value* of \*package\*. If no such *package* already exists, an error of type **package-error** is signaled.

## CLHS: Declaration DYNAMIC-EXTENT

Everything in-package does is also performed at compile time if the call appears as a top level form.

**Examples:** None.

**Side Effects:**

The variable \*package\* is assigned. If the in-package form is a top level form, this assignment also occurs at compile time.

**Affected By:** None.

**Exceptional Situations:**

An error of type package-error is signaled if the specified package does not exist.

**See Also:**

\*package\*

**Notes:** None.

## Macro INCF, DECF

**Syntax:**

**incf** *place* [*delta-form*] => *new-value*

**decf** *place* [*delta-form*] => *new-value*

**Arguments and Values:**

*place*—a place.

*delta-form*—a form; evaluated to produce a *delta*. The default is 1.

*delta*—a number.

*new-value*—a number.

**Description:**

**incf** and **decf** are used for incrementing and decrementing the value of *place*, respectively.

The *delta* is added to (in the case of **incf**) or subtracted from (in the case of **decf**) the number in *place* and the result is stored in *place*.

Any necessary type conversions are performed automatically.

For information about the evaluation of subforms of *places*, see Section 5.1.1.1 (Evaluation of Subforms to Places).

**Examples:**

```
(setq n 0)
(incf n) => 1
n => 1
(decf n 3) => -2
n => -2
(decf n -5) => 3
(decf n) => 2
(incf n 0.5) => 2.5
(decf n) => 1.5
n => 1.5
```

**Side Effects:**

*Place* is modified.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[±](#), [\\_](#), [1+](#), [1-](#), [setf](#)

**Notes:** None.

**Macro LAMBDA****Syntax:**

**lambda** *lambda-list* [[*declaration\** / *documentation*]] *form\** => *function*

**Arguments and Values:**

*lambda-list*—an ordinary lambda list.

*declaration*—a **declare expression**; not evaluated.

*documentation*—a string; not evaluated.

*form*—a form.

*function*—a function.

**Description:**

Provides a shorthand notation for a **function special form** involving a lambda expression such that:

```
(lambda lambda-list [[declaration* | documentation]] form*)
== (function (lambda lambda-list [[declaration* | documentation]] form*))
== #'(lambda lambda-list [[declaration* | documentation]] form*)
```

**Examples:**

```
(funcall (lambda (x) (+ x 3)) 4) => 7
```

**Side Effects:** None.**Affected By:** None.**Exceptional Situations:** None.**See Also:****lambda** (symbol)**Notes:**

This macro could be implemented by:

```
(defmacro lambda (&whole form &rest bvl-decls-and-body)
  (declare (ignore bvl-decls-and-body))
  `#,form)
```

**Macro LOOP****Syntax:**

The "simple" **loop form**:

**loop compound-form\*** => *result\**

The "extended" **loop form**:

**loop [name-clause] {variable-clause}\* {main-clause}\* => result\***

name-clause ::= named name

variable-clause ::= with-clause | initial-final | for-as-clause

with-clause ::= with var1 [type-spec] [= form1] {and var2 [type-spec] [= form2]}\*

main-clause ::= unconditional | accumulation | conditional | termination-test | initial-final

initial-final ::= initially compound-form+ | finally compound-form+

unconditional ::= {do | doing} compound-form+ | return {form | it}

accumulation ::= list-accumulation | numeric-accumulation

list-accumulation ::= {collect | collecting | append | appending | nconc | nconcning} {form | it} [into simple-var]

numeric-accumulation ::= {count | counting | sum | summing | } maximize | maximizing | minimize | minimizing {form | it}

## CLHS: Declaration DYNAMIC-EXTENT

```

[into simple-var] [type-spec]

conditional ::= {if | when | unless} form selectable-clause {and selectable-clause}*  

               [else selectable-clause {and selectable-clause}*]  

               [end]

selectable-clause ::= unconditional | accumulation | conditional

termination-test ::= while form | until form | repeat form | always form | never form | thereis

for-as-clause ::= {for | as} for-as-subclause {and for-as-subclause}*  

for-as-subclause ::= for-as-arithmetic | for-as-in-list | for-as-on-list | for-as>equals-then |  

                   for-as-across | for-as-hash | for-as-package

for-as-arithmetic ::= var [type-spec] for-as-arithmetic-subclause

for-as-arithmetic-subclause ::= arithmetic-up | arithmetic-downto | arithmetic-downfrom

arithmetic-up ::= [{from | upfrom} form1 | {to | upto | below} form2 | by form3]+

arithmetic-downto ::= [{from form1}]1 | [{downto | above} form2]1 | by form3]

arithmetic-downfrom ::= [{downfrom form1}]1 | {to | downto | above} form2 | by form3]

for-as-in-list ::= var [type-spec] in form1 [by step-fun]

for-as-on-list ::= var [type-spec] on form1 [by step-fun]

for-as>equals-then ::= var [type-spec] = form1 [then form2]

for-as-across ::= var [type-spec] across vector

for-as-hash ::= var [type-spec] being {each | the}  

               {{hash-key | hash-keys} {in | of} hash-table  

                [using (hash-value other-var)] |  

                {hash-value | hash-values} {in | of} hash-table  

                [using (hash-key other-var)]}

for-as-package ::= var [type-spec] being {each | the}  

                  {symbol | symbols |  

                   present-symbol | present-symbols |  

                   external-symbol | external-symbols}  

                  [{in | of} package]

type-spec ::= simple-type-spec | destructured-type-spec

simple-type-spec ::= fixnum | float | t | nil

destructured-type-spec ::= of-type d-type-spec

d-type-spec ::= type-specifier | (d-type-spec . d-type-spec)

var ::= d-var-spec

var1 ::= d-var-spec

```

## CLHS: Declaration DYNAMIC-EXTENT

```
var2 ::= d-var-spec  
  
other-var ::= d-var-spec  
  
d-var-spec ::= simple-var | nil | (d-var-spec . d-var-spec)
```

### Arguments and Values:

*compound-form*—a compound form.

*name*—a symbol.

*simple-var*—a symbol (a variable name).

*form, form1, form2, form3*—a form.

*step-fun*—a form that evaluates to a function of one argument.

*vector*—a form that evaluates to a vector.

*hash-table*—a form that evaluates to a hash table.

*package*—a form that evaluates to a package designator.

*type-specifier*—a type specifier. This might be either an atomic type specifier or a compound type specifier, which introduces some additional complications to proper parsing in the face of destructuring; for further information, see [Section 6.1.1.7 \(Destructuring\)](#).

*result*—an object.

### Description:

For details, see [Section 6.1 \(The LOOP Facility\)](#).

### Examples:

```
; ; An example of the simple form of LOOP.  
(defun sqrt-advisor ()  
  (loop (format t "~&Number: ")  
        (let ((n (parse-integer (read-line) :junk-allowed t)))  
          (when (not n) (return))  
          (format t "~&The square root of ~D is ~D.~%" n (sqrt n))))  
=> SQRT-ADVISOR  
(sqrt-advisor)  
>> Number: 5<NEWLINE>  
>> The square root of 5 is 2.236068.  
>> Number: 4<NEWLINE>  
>> The square root of 4 is 2.  
>> Number: done<NEWLINE>  
=> NIL  
  
; ; An example of the extended form of LOOP.  
(defun square-advisor ()  
  (loop as n = (progn (format t "~&Number: ")  
                      (parse-integer (read-line) :junk-allowed t)))
```

## CLHS: Declaration DYNAMIC-EXTENT

```
while n
  do (format t "~&The square of ~D is ~D.~%" n (* n n)))
=> SQUARE-ADVISOR
  (square-advisor)
>> Number: 4<NEWLINE>
>> The square of 4 is 16.
>> Number: 23<NEWLINE>
>> The square of 23 is 529.
>> Number: done<NEWLINE>
=> NIL

;; Another example of the extended form of LOOP.
(loop for n from 1 to 10
      when (oddp n)
      collect n)
=> (1 3 5 7 9)
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**do, dolist, dotimes, return, go, throw**, Section 6.1.1.7 (Destructuring)

**Notes:**

Except that **loop-finish** cannot be used within a simple **loop form**, a simple **loop form** is related to an extended **loop form** in the following way:

```
(loop compound-form*) == (loop do compound-form*)
```

## Local Macro LOOP-FINISH

**Syntax:**

**loop-finish <no arguments>** =>|

**Description:**

The **loop-finish macro** can be used lexically within an extended **loop form** to terminate that *form* "normally." That is, it transfers control to the loop epilogue of the lexically innermost extended **loop form**. This permits execution of any **finally** clause (for effect) and the return of any accumulated result.

**Examples:**

```
; ; Terminate the loop, but return the accumulated count.
(loop for i in '(1 2 3 stop-here 4 5 6)
      when (symbolp i) do (loop-finish)
      count i)
=> 3

; ; The preceding loop is equivalent to:
(loop for i in '(1 2 3 stop-here 4 5 6)
      until (symbolp i))
```

## CLHS: Declaration DYNAMIC-EXTENT

```
        count i)
=> 3

;; While LOOP-FINISH can be used can be used in a variety of
;; situations it is really most needed in a situation where a need
;; to exit is detected at other than the loop's `top level'
;; (where UNTIL or WHEN often work just as well), or where some
;; computation must occur between the point where a need to exit is
;; detected and the point where the exit actually occurs.  For example:
(defun tokenize-sentence (string)
  (macrolet ((add-word (wvar svar)
              ` (when ,wvar
                  (push (coerce (nreverse ,wvar) 'string) ,svar)
                  (setq ,wvar nil))))
    (loop with word = '() and sentence = '() and endpos = nil
          for i below (length string)
          do (let ((char (aref string i)))
               (case char
                 (#\Space (add-word word sentence))
                 (#\. (setq endpos (1+ i)) (loop-finish))
                 (otherwise (push char word)))
               finally (add-word word sentence)
               (return (values (nreverse sentence) endpos))))))
=> TOKENIZE-SENTENCE

(tokenize-sentence "this is a sentence. this is another sentence.")
=> ("this" "is" "a" "sentence"), 19

(tokenize-sentence "this is a sentence")
=> ("this" "is" "a" "sentence"), NIL
```

### Side Effects:

Transfers control.

Affected By: None.

### Exceptional Situations:

Whether or not **loop-finish** is *fbound* in the *global environment* is *implementation-dependent*; however, the restrictions on redefinition and shadowing of **loop-finish** are the same as for *symbols* in the COMMON-LISP package which are *fbound* in the *global environment*. The consequences of attempting to use **loop-finish** outside of **loop** are undefined.

### See Also:

**loop**, Section 6.1 (The LOOP Facility)

### Notes:

## Macro MULTIPLE-VALUE-LIST

### Syntax:

**multiple-value-list** *form* => *list*

**Arguments and Values:**

*form*—a form; evaluated as described below.

*list*—a list of the values returned by *form*.

**Description:**

**multiple-value-list** evaluates *form* and creates a list of the multiple values[2] it returns.

**Examples:**

```
(multiple-value-list (floor -3 4)) => (-1 1)
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

values-list, multiple-value-call

**Notes:**

**multiple-value-list** and values-list are inverses of each other.

```
(multiple-value-list form) == (multiple-value-call #'list form)
```

**Macro MULTIPLE-VALUE-SETQ****Syntax:**

**multiple-value-setq** *vars form* => *result*

**Arguments and Values:**

*vars*—a list of symbols that are either variable names or names of symbol macros.

*form*—a form.

*result*—The primary value returned by the *form*.

**Description:**

**multiple-value-setq** assigns values to *vars*.

The *form* is evaluated, and each *var* is *assigned* to the corresponding value returned by that *form*. If there are more *vars* than values returned, nil is *assigned* to the extra *vars*. If there are more values than *vars*, the extra values are discarded.

## CLHS: Declaration DYNAMIC-EXTENT

If any *var* is the *name* of a *symbol macro*, then it is *assigned* as if by **setf**. Specifically,

```
(multiple-value-setq (symbol1 ... symboln) value-producing-form)
```

is defined to always behave in the same way as

```
(values (setf (values symbol1 ... symboln) value-producing-form))
```

in order that the rules for order of evaluation and side-effects be consistent with those used by **setf**. See Section 5.1.2.3 (VALUES Forms as Places).

### Examples:

```
(multiple-value-setq (quotient remainder) (truncate 3.2 2)) => 1
quotient => 1
remainder => 1.2
(multiple-value-setq (a b c) (values 1 2)) => 1
a => 1
b => 2
c => NIL
(multiple-value-setq (a b) (values 4 5 6)) => 4
a => 4
b => 5
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

### See Also:

**setq**, **symbol-macrolet**

**Notes:** None.

## Macro MULTIPLE-VALUE-BIND

### Syntax:

**multiple-value-bind** (*var*\**values-form declaration\***form*\*

=> *result*\*

### Arguments and Values:

*var*—a *symbol* naming a variable; not evaluated.

*values-form*—a *form*; evaluated.

*declaration*—a **declare expression**; not evaluated.

*forms*—an *implicit progn*.

*results*—the values returned by the *forms*.

### Description:

Creates new variable bindings for the *vars* and executes a series of *forms* that use these bindings.

The variable bindings created are lexical unless special declarations are specified.

*Values-form* is evaluated, and each of the *vars* is bound to the respective value returned by that *form*. If there are more *vars* than values returned, extra values of nil are given to the remaining *vars*. If there are more values than *vars*, the excess values are discarded. The *vars* are bound to the values over the execution of the *forms*, which make up an implicit progn. The consequences are unspecified if a type *declaration* is specified for a *var*, but the value to which that *var* is bound is not consistent with the type *declaration*.

The scopes of the name binding and *declarations* do not include the *values-form*.

### Examples:

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

**Affected By:** None.

**Exceptional Situations:** None.

### See Also:

let, multiple-value-call

### Notes:

```
(multiple-value-bind (var*) values-form form*)
== (multiple-value-call #'(lambda (&optional var* &rest #1=#:ignore)
                           (declare (ignore #1#))
                           form*)
                           values-form)
```

## Macro NTH-VALUE

### Syntax:

**nth-value** *n form* => *object*

### Arguments and Values:

*n*—a non-negative integer; evaluated.

*form*—a form; evaluated as described below.

*object*—an object.

**Description:**

Evaluates *n* and then *form*, returning as its only value the *n*th value *yielded* by *form*, or nil if *n* is greater than or equal to the number of values returned by *form*. (The first returned value is numbered 0.)

**Examples:**

```
(nth-value 0 (values 'a 'b)) => A
(nth-value 1 (values 'a 'b)) => B
(nth-value 2 (values 'a 'b)) => NIL
(let* ((x 83927472397238947423879243432432432)
       (y 32423489732)
       (a (nth-value 1 (floor x y))))
  (b (mod x y)))
  (values a b (= a b)))
=> 3332987528, 3332987528, true
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[multiple-value-list](#), [nth](#)

**Notes:**

Operationally, the following relationship is true, although nth-value might be more efficient in some implementations because, for example, some *consing* might be avoided.

```
(nth-value n form) == (nth n (multiple-value-list form))
```

**Macro OR****Syntax:**

**or** *form\**  $\Rightarrow$  *results\**

**Arguments and Values:**

*form*—a *form*.

*results*—the values or primary value (see below) resulting from the evaluation of the last *form* executed or nil.

**Description:**

**or** evaluates each *form*, one at a time, from left to right. The evaluation of all *forms* terminates when a *form* evaluates to true (i.e., something other than nil).

## CLHS: Declaration DYNAMIC-EXTENT

If the evaluation of any form other than the last returns a primary value that is true, or immediately returns that value (but no additional values) without evaluating the remaining forms. If every form but the last returns false as its primary value, or returns all values returned by the last form. If no forms are supplied, or returns nil.

### Examples:

```
(or) => NIL
(setq temp0 nil temp1 10 temp2 20 temp3 30) => 30
(or temp0 temp1 (setq temp2 37)) => 10
temp2 => 20
(or (incf temp1) (incf temp2) (incf temp3)) => 11
temp1 => 11
temp2 => 20
temp3 => 30
(or (values) temp1) => 11
(or (values temp1 temp2) temp3) => 11
(or temp0 (values temp1 temp2)) => 11, 20
(or (values temp0 temp1) (values temp2 temp3)) => 20, 30
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

### See Also:

and, some, unless

**Notes:** None.

## Macro POP

### Syntax:

**pop** *place* => *element*

### Arguments and Values:

*place*—a *place*, the value of which is a list (possibly, but necessarily, a dotted list or circular list).

*element*—an *object* (the car of the contents of *place*).

### Description:

**pop** reads the value of *place*, remembers the car of the list which was retrieved, writes the cdr of the list back into the *place*, and finally yields the car of the originally retrieved list.

For information about the evaluation of subforms of *place*, see Section 5.1.1.1 (Evaluation of Subforms to Places).

### Examples:

## CLHS: Declaration DYNAMIC-EXTENT

```
(setq stack '(a b c)) => (A B C)
(pop stack) => A
stack => (B C)
(setq llst '((1 2 3 4))) => ((1 2 3 4))
(pop (car llst)) => 1
llst => ((2 3 4))
```

### Side Effects:

The contents of *place* are modified.

**Affected By:** None.

**Exceptional Situations:** None.

### See Also:

[push, pushnew, Section 5.1 \(Generalized Reference\)](#)

### Notes:

The effect of (*pop place*) is roughly equivalent to

```
(prog1 (car place) (setf place (cdr place)))
```

except that the latter would evaluate any *subforms* of *place* three times, while [pop](#) evaluates them only once.

## **Local Macro PPRINT-EXIT-IF-LIST-EXHAUSTED**

### Syntax:

**pprint-exit-if-list-exhausted** <no arguments> => [nil](#)

**Arguments and Values:** None.

### Description:

Tests whether or not the *list* passed to the *lexically current logical block* has been exhausted; see [Section 22.2.1.1 \(Dynamic Control of the Arrangement of Output\)](#). If this *list* has been reduced to [nil](#), **pprint-exit-if-list-exhausted** terminates the execution of the *lexically current logical block* except for the printing of the suffix. Otherwise **pprint-exit-if-list-exhausted** returns [nil](#).

Whether or not **pprint-exit-if-list-exhausted** is *fbound* in the *global environment* is *implementation-dependent*; however, the restrictions on redefinition and shadowing of **pprint-exit-if-list-exhausted** are the same as for *symbols* in the COMMON-LISP package which are *fbound* in the *global environment*. The consequences of attempting to use **pprint-exit-if-list-exhausted** outside of **pprint-logical-block** are undefined.

**Examples:** None.

**Side Effects:** None.

**Affected By:** None.

#### Exceptional Situations:

An error is signaled (at macro expansion time or at run time) if **pprint-exit-if-list-exhausted** is used anywhere other than lexically within a call on **pprint-logical-block**. Also, the consequences of executing **pprint-if-list-exhausted** outside of the dynamic extent of the **pprint-logical-block** which lexically contains it are undefined.

**See Also:**

**pprint-logical-block, pprint-pop.**

**Notes:** None.

## Macro PPRINT-LOGICAL-BLOCK

#### Syntax:

**pprint-logical-block** (*stream-symbol object &key prefix per-line-prefix suffix*) *declaration\** *form\**

=> **nil**

#### Arguments and Values:

*stream-symbol*—a stream variable designator.

*object*—an object; evaluated.

*:prefix*—a string; evaluated. Complicated defaulting behavior; see below.

*:per-line-prefix*—a string; evaluated. Complicated defaulting behavior; see below.

*:suffix*—a string; evaluated. The default is the null string.

*declaration*—a declare expression; not evaluated.

*forms*—an implicit progn.

#### Description:

Causes printing to be grouped into a logical block.

The logical block is printed to the stream that is the value of the variable denoted by *stream-symbol*. During the execution of the forms, that variable is bound to a pretty printing stream that supports decisions about the arrangement of output and then forwards the output to the destination stream. All the standard printing functions (e.g., **write**, **princ**, and **terpri**) can be used to print output to the pretty printing stream. All and only the output sent to this pretty printing stream is treated as being in the logical block.

The *prefix* specifies a prefix to be printed before the beginning of the logical block. The *per-line-prefix* specifies a prefix that is printed before the block and at the beginning of each new line in the block. The

## CLHS: Declaration DYNAMIC-EXTENT

:prefix and :pre-line-prefix *arguments* are mutually exclusive. If neither :prefix nor :per-line-prefix is specified, a *prefix* of the *null string* is assumed.

The *suffix* specifies a suffix that is printed just after the logical block.

The *object* is normally a *list* that the body *forms* are responsible for printing. If *object* is not a *list*, it is printed using write. (This makes it easier to write printing functions that are robust in the face of malformed arguments.) If \*print-circle\* is *non-nil* and *object* is a circular (or shared) reference to a *cons*, then an appropriate "#n#" marker is printed. (This makes it easy to write printing functions that provide full support for circularity and sharing abbreviation.) If \*print-level\* is not *nil* and the logical block is at a dynamic nesting depth of greater than \*print-level\* in logical blocks, "#" is printed. (This makes it easy to write printing functions that provide full support for depth abbreviation.)

If either of the three conditions above occurs, the indicated output is printed on *stream-symbol* and the body *forms* are skipped along with the printing of the :prefix and :suffix. (If the body *forms* are not to be responsible for printing a list, then the first two tests above can be turned off by supplying *nil* for the *object* argument.)

In addition to the *object* argument of pprint-logical-block, the arguments of the standard printing functions (such as write, print, prin1, and pprint, as well as the arguments of the standard format directives such as ~A, ~S, (and ~W) are all checked (when necessary) for circularity and sharing. However, such checking is not applied to the arguments of the functions write-line, write-string, and write-char or to the literal text output by format. A consequence of this is that you must use one of the latter functions if you want to print some literal text in the output that is not supposed to be checked for circularity or sharing.

The body *forms* of a pprint-logical-block form must not perform any side-effects on the surrounding environment; for example, no *variables* must be assigned which have not been *bound* within its scope.

The pprint-logical-block macro may be used regardless of the *value* of \*print-pretty\*.

**Examples:** None.

**Side Effects:** None.

**Affected By:**

\*print-circle\*, \*print-level\*.

**Exceptional Situations:**

An error of type type-error is signaled if any of the :suffix, :prefix, or :per-line-prefix is supplied but does not evaluate to a *string*.

An error is signaled if :prefix and :pre-line-prefix are both used.

pprint-logical-block and the pretty printing stream it creates have dynamic extent. The consequences are undefined if, outside of this extent, output is attempted to the pretty printing stream it creates.

It is also unspecified what happens if, within this extent, any output is sent directly to the underlying destination stream.

**See Also:**

[\*\*pprint-pop\*\*](#), [\*\*pprint-exit-if-list-exhausted\*\*](#), [Section 22.3.5.2 \(Tilde Less-Than-Sign: Logical Block\)](#)

**Notes:**

One reason for using the [\*\*pprint-logical-block macro\*\*](#) when the *value* of [\*\*\\*print-pretty\\*\*\*](#) is [\*\*nil\*\*](#) would be to allow it to perform checking for *dotted lists*, as well as (in conjunction with [\*\*pprint-pop\*\*](#)) checking for [\*\*\\*print-level\\*\*\*](#) or [\*\*\\*print-length\\*\*\*](#) being exceeded.

Detection of circularity and sharing is supported by the *pretty printer* by in essence performing requested output twice. On the first pass, circularities and sharing are detected and the actual outputting of characters is suppressed. On the second pass, the appropriate "#n=" and "#n#" markers are inserted and characters are output. This is why the restriction on side-effects is necessary. Obeying this restriction is facilitated by using [\*\*pprint-pop\*\*](#), instead of an ordinary [\*\*pop\*\*](#) when traversing a list being printed by the body *forms* of the [\*\*pprint-logical-block form\*\*](#).)

**Local Macro PPRINT-POP****Syntax:**

**pprint-pop** <no arguments> => *object*

**Arguments and Values:**

*object*—an *element* of the *list* being printed in the *lexically current logical block*, or [\*\*nil\*\*](#).

**Description:**

Pops one *element* from the *list* being printed in the *lexically current logical block*, obeying [\*\*\\*print-length\\*\*\*](#) and [\*\*\\*print-circle\\*\*\*](#) as described below.

Each time [\*\*pprint-pop\*\*](#) is called, it pops the next value off the *list* passed to the *lexically current logical block* and returns it. However, before doing this, it performs three tests:

- If the remaining `list' is not a *list*, ". ." is printed followed by the remaining `list.' (This makes it easier to write printing functions that are robust in the face of malformed arguments.)
- If [\*\*\\*print-length\\*\*\*](#) is *non-nil*, and [\*\*pprint-pop\*\*](#) has already been called [\*\*\\*print-length\\*\*\*](#) times within the immediately containing logical block, "... ." is printed. (This makes it easy to write printing functions that properly handle [\*\*\\*print-length\\*\*\*\).](#)
- If [\*\*\\*print-circle\\*\*\*](#) is *non-nil*, and the remaining list is a circular (or shared) reference, then ". ." is printed followed by an appropriate "#n#" marker. (This catches instances of *cdr* circularity and sharing in lists.)

If either of the three conditions above occurs, the indicated output is printed on the *pretty printing stream* created by the immediately containing [\*\*pprint-logical-block\*\*](#) and the execution of the immediately containing [\*\*pprint-logical-block\*\*](#) is terminated except for the printing of the suffix.

If [\*\*pprint-logical-block\*\*](#) is given a `list' argument of [\*\*nil\*\*](#)—because it is not processing a list—[\*\*pprint-pop\*\*](#) can still be used to obtain support for [\*\*\\*print-length\\*\*\*](#). In this situation, the first and third tests above are disabled and [\*\*pprint-pop\*\*](#) always returns [\*\*nil\*\*](#). See [Section 22.2.2 \(Examples of using the Pretty](#)

Printer)---specifically, the **pprint–vector** example.

Whether or not **pprint–pop** is *fbound* in the *global environment* is *implementation-dependent*; however, the restrictions on redefinition and shadowing of **pprint–pop** are the same as for **symbols** in the COMMON-LISP package which are *fbound* in the *global environment*. The consequences of attempting to use **pprint–pop** outside of **pprint–logical–block** are undefined.

**Examples:** None.

#### Side Effects:

Might cause output to the *pretty printing stream* associated with the lexically current logical block.

#### Affected By:

**\*print-length\*, \*print-circle\*.**

#### Exceptional Situations:

An error is signaled (either at macro expansion time or at run time) if a usage of **pprint–pop** occurs where there is no lexically containing **pprint–logical–block form**.

The consequences are undefined if **pprint–pop** is executed outside of the *dynamic extent* of this **pprint–logical–block**.

#### See Also:

**pprint–exit–if–list–exhausted, pprint–logical–block**.

#### Notes:

It is frequently a good idea to call **pprint–exit–if–list–exhausted** before calling **pprint–pop**.

## Macro PRINT–UNREADABLE–OBJECT

#### Syntax:

**print–unreadable–object** (*object stream &key type identity*) *form\** => **nil**

#### Arguments and Values:

*object*---an *object*; evaluated.

*stream*---a *stream designator*; evaluated.

*type*---a *generalized boolean*; evaluated.

*identity*---a *generalized boolean*; evaluated.

*forms*---an *implicit progn*.

**Description:**

Outputs a printed representation of *object* on *stream*, beginning with "#<" and ending with ">". Everything output to *stream* by the body *forms* is enclosed in the the angle brackets. If *type* is *true*, the output from *forms* is preceded by a brief description of the *object's type* and a space character. If *identity* is *true*, the output from *forms* is followed by a space character and a representation of the *object's identity*, typically a storage address.

If either *type* or *identity* is not supplied, its value is *false*. It is valid to omit the body *forms*. If *type* and *identity* are both true and there are no body *forms*, only one space character separates the type and the identity.

**Examples:**

;; Note that in this example, the precise form of the output ;; is *implementation-dependent*.

```
(defmethod print-object ((obj airplane) stream)
  (print-unreadable-object (obj stream :type t :identity t)
    (princ (tail-number obj) stream)))

(prin1-to-string my-airplane)
=> "#<Airplane NW0773 36000123135>"
OR=> "#<FAA:AIRPLANE NW0773 17>"
```

**Affected By:** None.

**Exceptional Situations:**

If **\*print-readably\*** is *true*, **print-unreadable-object** signals an error of *type print-not-readable* without printing anything.

**See Also:** None.

**Notes:** None.

**Macro PROG1, PROG2****Syntax:**

**prog1** *first-form* *form\** => *result-1*

**prog2** *first-form* *second-form* *form\** => *result-2*

**Arguments and Values:**

*first-form*—*a form*; evaluated as described below.

*second-form*—*a form*; evaluated as described below.

*forms*—*an implicit progn*; evaluated as described below.

*result-1*—*the primary value* resulting from the *evaluation* of *first-form*.

## CLHS: Declaration DYNAMIC-EXTENT

*result-2*—the primary value resulting from the evaluation of *second-form*.

### Description:

prog1 evaluates *first-form* and then *forms*, yielding as its only value the primary value yielded by *first-form*.

prog2 evaluates *first-form*, then *second-form*, and then *forms*, yielding as its only value the primary value yielded by *first-form*.

### Examples:

```
(setq temp 1) => 1
(prog1 temp (print temp) (incf temp) (print temp))
=> 1
=> 2
=> 1
(prog1 temp (setq temp nil)) => 2
temp => NIL
(prog1 (values 1 2 3) 4) => 1
(setq temp (list 'a 'b 'c))
(prog1 (car temp) (setf (car temp) 'alpha)) => A
temp => (ALPHA B C)
(flet ((swap-symbol-values (x y)
    (setf (symbol-value x)
        (prog1 (symbol-value y)
            (setf (symbol-value y) (symbol-value x)))))))
(let ((*foo* 1) (*bar* 2))
    (declare (special *foo* *bar*))
    (swap-symbol-values '*foo* '*bar*)
    (values *foo* *bar*)))
=> 2, 1
(setq temp 1) => 1
(prog2 (incf temp) (incf temp) (incf temp)) => 3
temp => 4
(prog2 1 (values 2 3 4) 5) => 2
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[multiple-value-prog1, progn](#)

**Notes:**

prog1 and prog2 are typically used to evaluate one or more forms with side effects and return a value that must be computed before some or all of the side effects happen.

```
(prog1 form*) == (values (multiple-value-prog1 form*))
(prog2 form1 form*) == (let () form1 (prog1 form*))
```

**Macro PROG, PROG\*****Syntax:**

**prog** ({var / (var [init-form])}\*) declaration\* {tag / statement}\*  
 => result\*

**prog\*** ({var / (var [init-form])}\*) declaration\* {tag / statement}\*  
 => result\*

**Arguments and Values:**

*var*—variable name.

*init-form*—a form.

*declaration*—a declare expression; not evaluated.

*tag*—a go tag; not evaluated.

*statement*—a compound form; evaluated as described below.

*results*—nil if a normal return occurs, or else, if an explicit return occurs, the values that were transferred.

**Description:**

Three distinct operations are performed by **prog** and **prog\***: they bind local variables, they permit use of the return statement, and they permit use of the go statement. A typical **prog** looks like this:

```
(prog (var1 var2 (var3 init-form-3) var4 (var5 init-form-5))
      declaration*
      statement1
      tag1
      statement2
      statement3
      statement4
      tag2
      statement5
      ...
    )
```

For **prog**, *init-forms* are evaluated first, in the order in which they are supplied. The *vars* are then bound to the corresponding values in parallel. If no *init-form* is supplied for a given *var*, that *var* is bound to nil.

The body of **prog** is executed as if it were a tagbody form; the go statement can be used to transfer control to a *tag*. *Tags* label *statements*.

**prog** implicitly establishes a block named nil around the entire prog form, so that return can be used at any time to exit from the prog form.

## CLHS: Declaration DYNAMIC-EXTENT

The difference between prog\* and prog is that in prog\* the *binding* and initialization of the *vars* is done *sequentially*, so that the *init-form* for each one can use the values of previous ones.

### Examples:

```
(prog* ((y z) (x (car y)))
       (return x))
```

returns the car of the value of z.

```
(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

(defun king-of-confusion (w)
  "Take a cons of two lists and make a list of conses.
   Think of this function as being like a zipper."
  (prog (x y z) ;Initialize x, y, z to NIL
        (setq y (car w) z (cdr w))
        loop
        (cond ((null y) (return x))
              ((null z) (go err)))
        rejoin
        (setq x (cons (cons (car y) (car z)) x))
        (setq y (cdr y) z (cdr z))
        (go loop)
        err
        (cerror "Will self-pair extraneous items"
                "Mismatch - gleep! ~S" y)
        (setq z y)
        (go rejoин))) => KING-OF-CONFUSION
```

This can be accomplished more perspicuously as follows:

```
(defun prince-of-clarity (w)
  "Take a cons of two lists and make a list of conses.
   Think of this function as being like a zipper."
  (do ((y (car w) (cdr y))
       (z (cdr w) (cdr z))
       (x '() (cons (cons (car y) (car z)) x)))
       ((null y) x)
       (when (null z)
         (cerror "Will self-pair extraneous items"
                 "Mismatch - gleep! ~S" y)
         (setq z y)))) => PRINCE-OF-CLARITY
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

block, let, tagbody, go, return, Section 3.1 (Evaluation)

**Notes:**

**prog** can be explained in terms of **block**, **let**, and **tagbody** as follows:

```
(prog variable-list declaration . body)
  == (block nil (let variable-list declaration (tagbody . body)))
```

## Macro PSETQ

**Syntax:**

**psetq** {pair}\* => **nil**

pair ::= var form

**Pronunciation:**

**psetq**: [;pee'set,kyoo]

**Arguments and Values:**

*var*—a symbol naming a variable other than a constant variable.

*form*—a form.

**Description:**

Assigns values to variables.

This is just like **setq**, except that the assignments happen "in parallel." That is, first all of the forms are evaluated, and only then are the variables set to the resulting values. In this way, the assignment to one variable does not affect the value computation of another in the way that would occur with **setq**'s sequential assignment.

If any *var* refers to a binding made by **symbol-macrolet**, then that *var* is treated as if **psetf** (not **psetq**) had been used.

**Examples:**

```
; ; A simple use of PSETQ to establish values for variables.
; ; As a matter of style, many programmers would prefer SETQ
; ; in a simple situation like this where parallel assignment
; ; is not needed, but the two have equivalent effect.
(psetq a 1 b 2 c 3) => NIL
a => 1
b => 2
c => 3

; ; Use of PSETQ to update values by parallel assignment.
; ; The effect here is very different than if SETQ had been used.
(psetq a (1+ b) b (1+ a) c (+ a b)) => NIL
a => 3
b => 2
c => 3

; ; Use of PSETQ on a symbol macro.
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(let ((x (list 10 20 30)))
  (symbol-macrolet ((y (car x)) (z (cadr x)))
    (psetq y (1+ z) z (1+ y))
    (list x y z)))
=> ((21 11 30) 21 11)

;; Use of parallel assignment to swap values of A and B.
(let ((a 1) (b 2))
  (psetq a b b a)
  (values a b))
=> 2, 1
```

### Side Effects:

The values of *forms* are assigned to *vars*.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**psetf, setq**

**Notes:** None.

## Macro PUSHNEW

### Syntax:

**pushnew** *item place &key key test test-not*

=> *new-place-value*

### Arguments and Values:

*item*—an object.

*place*—a place, the value of which is a proper list.

*test*—a designator for a function of two arguments that returns a generalized boolean.

*test-not*—a designator for a function of two arguments that returns a generalized boolean.

*key*—a designator for a function of one argument, or **nil**.

*new-place-value*—a list (the new value of *place*).

### Description:

**pushnew** tests whether *item* is the same as any existing element of the list stored in *place*. If *item* is not, it is prepended to the list, and the new list is stored in *place*.

**pushnew** returns the new *list* that is stored in *place*.

Whether or not *item* is already a member of the *list* that is in *place* is determined by comparisons using :test or :test-not. The first argument to the :test or :test-not function is *item*; the second argument is an element of the *list* in *place* as returned by the :key function (if supplied).

If :key is supplied, it is used to extract the part to be tested from both *item* and the *list* element, as for **adjoin**.

The argument to the :key function is an element of the *list* stored in *place*. The :key function typically returns part of the element of the *list*. If :key is not supplied or **nil**, the *list* element is used.

For information about the *evaluation* of *subforms* of *place*, see [Section 5.1.1.1 \(Evaluation of Subforms to Places\)](#).

It is *implementation-dependent* whether or not **pushnew** actually executes the storing form for its *place* in the situation where the *item* is already a member of the *list* held by *place*.

### Examples:

```
(setq x '(a (b c) d)) => (A (B C) D)
(pushnew 5 (cadr x)) => (5 B C)
x => (A (5 B C) D)
(pushnew 'b (cadr x)) => (5 B C)
x => (A (5 B C) D)
(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))
```

### Side Effects:

The contents of *place* may be modified.

**Affected By:** None.

**Exceptional Situations:** None.

### See Also:

**push**, **adjoin**, [Section 5.1 \(Generalized Reference\)](#)

### Notes:

The effect of

```
(pushnew item place :test p)
```

is roughly equivalent to

```
(setf place (adjoin item place :test p))
```

except that the *subforms* of *place* are evaluated only once, and *item* is evaluated before *place*.

**Macro PUSH****Syntax:**

**push** *item place => new-place-value*

**Arguments and Values:**

*item*—an object.

*place*—a place, the value of which may be any object.

*new-place-value*—a list (the new value of *place*).

**Description:**

**push** prepends *item* to the list that is stored in *place*, stores the resulting list in *place*, and returns the list.

For information about the evaluation of subforms of *place*, see Section 5.1.1.1 (Evaluation of Subforms to Places).

**Examples:**

```
(setq llst '(nil)) => (NIL)
(push 1 (car llst)) => (1)
llst => ((1))
(push 1 (car llst)) => (1 1)
llst => ((1 1))
(setq x '(a (b c) d)) => (A (B C) D)
(push 5 (cadr x)) => (5 B C)
x => (A (5 B C) D)
```

**Side Effects:**

The contents of *place* are modified.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[pop](#), [pushnew](#), [Section 5.1 \(Generalized Reference\)](#)

**Notes:**

The effect of (**push** *item place*) is equivalent to

```
(setf place (cons item place))
```

except that the subforms of *place* are evaluated only once, and *item* is evaluated before *place*.

**Macro REMF****Syntax:**

**remf** *place indicator => generalized-boolean*

**Arguments and Values:**

*place*—a place.

*indicator*—an object.

*generalized-boolean*—a generalized boolean.

**Description:**

**remf** removes from the property list stored in *place* a property[1] with a property indicator identical to *indicator*. If there are multiple properties[1] with the identical key, **remf** only removes the first such property. **remf** returns false if no such property was found, or true if a property was found.

The property indicator and the corresponding property value are removed in an undefined order by destructively splicing the property list. **remf** is permitted to either setf *place* or to setf any part, car or cdr, of the list structure held by that *place*.

For information about the evaluation of subforms of *place*, see [Section 5.1.1.1 \(Evaluation of Subforms to Places\)](#).

**Examples:**

```
(setq x (cons () ())) => (NIL)
(setf (getf (car x) 'prop1) 'val1) => VAL1
(remf (car x) 'prop1) => true
(remf (car x) 'prop1) => false
```

**Side Effects:**

The property list stored in *place* is modified.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[remprop](#), [getf](#)

**Notes:** None.

## **Macro RETURN**

### Syntax:

**return** [*result*] =>|

### Arguments and Values:

*result*—a *form*; evaluated. The default is **nil**.

### Description:

Returns, as if by **return-from**, from the *block* named **nil**.

### Examples:

```
(block nil (return) 1) => NIL
(block nil (return 1) 2) => 1
(block nil (return (values 1 2)) 3) => 1, 2
(block nil (block alpha (return 1) 2)) => 1
(block alpha (block nil (return 1)) 2) => 2
(block nil (block nil (return 1) 2)) => 1
```

Affected By: None.

Conditions: None.

### See Also:

**block**, **return-from**, Section 3.1 (Evaluation)

### Notes:

```
(return) == (return-from nil)
(return form) == (return-from nil form)
```

The *implicit blocks* established by *macros* such as **do** are often named **nil**, so that **return** can be used to exit from such *forms*.

## **Macro ROTATEF**

### Syntax:

**rotatef** *place*\* => **nil**

### Arguments and Values:

*place*—a *place*.

### Description:

**rotatef** modifies the values of each *place* by rotating values from one *place* into another.

## CLHS: Declaration DYNAMIC-EXTENT

If a *place* produces more values than there are store variables, the extra values are ignored. If a *place* produces fewer values than there are store variables, the missing values are set to **nil**.

In the form (`rotatef place1 place2 ... placen`), the values in *place1* through *placen* are *read* and *written*. Values 2 through *n* and value 1 are then stored into *place1* through *placen*. It is as if all the places form an end-around shift register that is rotated one place to the left, with the value of *place1* being shifted around the end to *placen*.

For information about the *evaluation* of *subforms* of *places*, see [Section 5.1.1.1 \(Evaluation of Subforms to Places\)](#).

### 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)
```

**Affected By:** None.

**Exceptional Situations:** None.

### See Also:

[define-setf-expander](#), [defsetf](#), [setf](#), [shiftf](#), [\\*macroexpand-hook\\*](#), [Section 5.1 \(Generalized Reference\)](#)

### Notes:

The effect of (`rotatef place1 place2 ... placen`) is roughly equivalent to

```
(psetf place1 place2
       place2 place3
       ...
       placen place1)
```

except that the latter would evaluate any *subforms* of each *place* twice, whereas **rotatef** evaluates them once.

## Macro RESTART-BIND

### Syntax:

**restart-bind** ({(name function {key-val-pair}\*)}) form\*

=> result\*

```
key-val-pair ::= :interactive-function interactive-function |
               :report-function report-function |
               :test-function test-function
```

### Arguments and Values:

Macro RESTART-BIND

614

*name*---a symbol; not evaluated.

*function*---a form; evaluated.

*forms*---an implicit progn.

*interactive-function*---a form; evaluated.

*report-function*---a form; evaluated.

*test-function*---a form; evaluated.

*results*---the values returned by the forms.

#### Description:

**restart-bind** executes the body of *forms* in a dynamic environment where restarts with the given *names* are in effect.

If a *name* is nil, it indicates an anonymous restart; if a *name* is a non-nil symbol, it indicates a named restart.

The *function*, *interactive-function*, and *report-function* are unconditionally evaluated in the current lexical and dynamic environment prior to evaluation of the body. Each of these forms must evaluate to a function.

If **invoke-restart** is done on that restart, the function which resulted from evaluating *function* is called, in the dynamic environment of the **invoke-restart**, with the arguments given to **invoke-restart**. The function may either perform a non-local transfer of control or may return normally.

If the restart is invoked interactively from the debugger (using **invoke-restart-interactively**), the arguments are defaulted by calling the function which resulted from evaluating *interactive-function*. That function may optionally prompt interactively on query I/O, and should return a list of arguments to be used by **invoke-restart-interactively** when invoking the restart.

If a restart is invoked interactively but no *interactive-function* is used, then an argument list of nil is used. In that case, the function must be compatible with an empty argument list.

If the restart is presented interactively (e.g., by the debugger), the presentation is done by calling the function which resulted from evaluating *report-function*. This function must be a function of one argument, a stream. It is expected to print a description of the action that the restart takes to that stream. This function is called any time the restart is printed while **\*print-escape\*** is nil.

In the case of interactive invocation, the result is dependent on the value of :*interactive-function* as follows.

##### :*interactive-function*

*Value* is evaluated in the current lexical environment and should return a function of no arguments which constructs a list of arguments to be used by **invoke-restart-interactively** when invoking this restart. The function may prompt interactively using query I/O if necessary.

##### :*report-function*

*Value* is evaluated in the current lexical environment and should return a function of one argument, a stream, which prints on the stream a summary of the action that this restart takes. This function is

## CLHS: Declaration DYNAMIC-EXTENT

called whenever the restart is reported (printed while **\*print-escape\*** is **nil**). If no **:report-function** option is provided, the manner in which the **restart** is reported is **implementation-dependent**.

**:test-function**

*Value* is evaluated in the current lexical environment and should return a **function** of one argument, a **condition**, which returns **true** if the restart is to be considered visible.

**Side Effects:** None.

**Affected By:**

**\*query-io\***

**Exceptional Situations:** None.

**See Also:**

**restart-case, with-simple-restart**

**Notes:**

**restart-bind** is primarily intended to be used to implement **restart-case** and might be useful in implementing other macros. Programmers who are uncertain about whether to use **restart-case** or **restart-bind** should prefer **restart-case** for the cases where it is powerful enough, using **restart-bind** only in cases where its full generality is really needed.

## Macro RESTART-CASE

**Syntax:**

**restart-case** *restartable-form* {*clause*} => *result*\*

```
clause ::= (case-name lambda-list
                  [[:interactive interactive-expression | :report report-expression | :test test-expression]
                   declaration* form*])
```

**Arguments and Values:**

*restartable-form*---a **form**.

*case-name*---a **symbol** or **nil**.

*lambda-list*---an **ordinary lambda list**.

*interactive-expression*---a **symbol** or a **lambda expression**.

*report-expression*---a **string**, a **symbol**, or a **lambda expression**.

*test-expression*---a **symbol** or a **lambda expression**.

*declaration*---a **declare expression**; not evaluated.

*form*—a form.

*results*—the values resulting from the evaluation of restartable-form, or the values returned by the last form executed in a chosen clause, or nil.

#### Description:

restart-case evaluates restartable-form in a dynamic environment where the clauses have special meanings as points to which control may be transferred. If restartable-form finishes executing and returns any values, all values returned are returned by restart-case and processing has completed. While restartable-form is executing, any code may transfer control to one of the clauses (see invoke-restart). If a transfer occurs, the forms in the body of that clause is evaluated and any values returned by the last such form are returned by restart-case. In this case, the dynamic state is unwound appropriately (so that the restarts established around the restartable-form are no longer active) prior to execution of the clause.

If there are no forms in a selected clause, restart-case returns nil.

If case-name is a symbol, it names this restart.

It is possible to have more than one clause use the same case-name. In this case, the first clause with that name is found by find-restart. The other clauses are accessible using compute-restarts.

Each arglist is an ordinary lambda list to be bound during the execution of its corresponding forms. These parameters are used by the restart-case clause to receive any necessary data from a call to invoke-restart.

By default, invoke-restart-interactively passes no arguments and all arguments must be optional in order to accomodate interactive restarting. However, the arguments need not be optional if the :interactive keyword has been used to inform invoke-restart-interactively about how to compute a proper argument list.

*Keyword* options have the following meaning.

##### :interactive

The value supplied by :interactive value must be a suitable argument to function. (function value) is evaluated in the current lexical environment. It should return a function of no arguments which returns arguments to be used by invoke-restart-interactively when it is invoked.

invoke-restart-interactively is called in the dynamic environment available prior to any restart attempt, and uses query I/O for user interaction.

If a restart is invoked interactively but no :interactive option was supplied, the argument list used in the invocation is the empty list.

##### :report

If the value supplied by :report value is a lambda expression or a symbol, it must be acceptable to function. (function value) is evaluated in the current lexical environment. It should return a function of one argument, a stream, which prints on the stream a description of the restart. This function is called whenever the restart is printed while \*print-escape\* is nil.

If value is a string, it is a shorthand for

```
(lambda (stream) (write-string value stream))
```

## CLHS: Declaration DYNAMIC-EXTENT

If a named restart is asked to report but no report information has been supplied, the name of the restart is used in generating default report text.

When **\*print-escape\*** is **nil**, the printer uses the report information for a restart. For example, a debugger might announce the action of typing a "continue" command by:

```
(format t "~~S -- ~A~%" ':continue some-restart)
```

which might then display as something like:

```
:CONTINUE -- Return to command level
```

The consequences are unspecified if an unnamed restart is specified but no :report option is provided.

**:test**

The *value* supplied by :test *value* must be a suitable argument to **function**. (*function value*) is evaluated in the current lexical environment. It should return a **function** of one **argument**, the **condition**, that returns **true** if the restart is to be considered visible.

The default for this option is equivalent to (lambda (c) (declare (ignore c)) t).

If the *restartable-form* is a **list** whose **car** is any of the **symbols signal, error, perror**, or **warn** (or is a **macro form** which macroexpands into such a **list**), then **with-condition-restarts** is used implicitly to associate the indicated **restarts** with the **condition** to be signaled.

### Examples:

```
(restart-case
  (handler-bind ((error #'(lambda (c)
                           (declare (ignore condition))
                           (invoke-restart 'my-restart 7)))
                 (error "Foo."))
    (my-restart (&optional v) v))
=> 7

(define-condition food-error (error) ())
=> FOOD-ERROR
(define-condition bad-tasting-sundae (food-error)
  ((ice-cream :initarg :ice-cream :reader bad-tasting-sundae-ice-cream)
   (sauce :initarg :sauce :reader bad-tasting-sundae-sauce)
   (topping :initarg :topping :reader bad-tasting-sundae-topping))
  (:report (lambda (condition stream)
             (format stream "Bad tasting sundae with ~S, ~S, and ~S"
                     (bad-tasting-sundae-ice-cream condition)
                     (bad-tasting-sundae-sauce condition)
                     (bad-tasting-sundae-topping condition)))))

=> BAD-TASTING-SUNDAE
(defun all-start-with-same-letter (symbol1 symbol2 symbol3)
  (let ((first-letter (char (symbol-name symbol1) 0)))
    (and (eql first-letter (char (symbol-name symbol2) 0))
         (eql first-letter (char (symbol-name symbol3) 0)))))
=> ALL-START-WITH-SAME-LETTER
(defun read-new-value ()
  (format t "Enter a new value: ")
  (multiple-value-list (eval (read))))
=> READ-NEW-VALUE
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(defun verify-or-fix-perfect-sundae (ice-cream sauce topping)
  (do ()
    ((all-start-with-same-letter ice-cream sauce topping)))
  (restart-case
    (error 'bad-tasting-sundae
           :ice-cream ice-cream
           :sauce sauce
           :topping topping)
    (use-new-ice-cream (new-ice-cream)
      :report "Use a new ice cream."
      :interactive read-new-value
      (setq ice-cream new-ice-cream))
    (use-new-sauce (new-sauce)
      :report "Use a new sauce."
      :interactive read-new-value
      (setq sauce new-sauce))
    (use-new-topping (new-topping)
      :report "Use a new topping."
      :interactive read-new-value
      (setq topping new-topping))))
  (values ice-cream sauce topping))
=> VERIFY-OR-FIX-PERFECT-SUNDAE
(verify-or-fix-perfect-sundae 'vanilla 'caramel 'cherry)
>> Error: Bad tasting sundae with VANILLA, CARAMEL, and CHERRY.
>> To continue, type :CONTINUE followed by an option number:
>> 1: Use a new ice cream.
>> 2: Use a new sauce.
>> 3: Use a new topping.
>> 4: Return to Lisp Toplevel.
>> Debug> :continue 1
>> Use a new ice cream.
>> Enter a new ice cream: 'chocolate
=> CHOCOLATE, CARAMEL, CHERRY
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[restart-bind](#), [with-simple-restart](#).

**Notes:**

```
(restart-case expression
  (name1 arglist1 ...options1... . body1)
  (name2 arglist2 ...options2... . body2))
```

is essentially equivalent to

```
(block #1=#:g0001
  (let ((#2=#:g0002 nil))
    (tagbody
      (restart-bind ((name1 #'(lambda (&rest temp)
                                (setq #2# temp)
                                (go #3=#:g0003))))
```

## CLHS: Declaration DYNAMIC-EXTENT

```
...slightly-transformed-options1...)
(name2 #'(lambda (&rest temp)
  (setq #2# temp)
  (go #4=#:g0004))
...slightly-transformed-options2...))
(return-from #1# expression))
#3# (return-from #1#
  (apply #'(lambda arglist1 . body1) #2#))
#4# (return-from #1#
  (apply #'(lambda arglist2 . body2) #2#)))))
```

Unnamed restarts are generally only useful interactively and an interactive option which has no description is of little value. Implementations are encouraged to warn if an unnamed restart is used and no report information is provided at compilation time. At runtime, this error might be noticed when entering the debugger. Since signaling an error would probably cause recursive entry into the debugger (causing yet another recursive error, etc.) it is suggested that the debugger print some indication of such problems when they occur but not actually signal errors.

```
(restart-case (signal fred)
  (a ...)
  (b ...))
===
(restart-case
  (with-condition-restarts fred
    (list (find-restart 'a)
          (find-restart 'b)))
  (signal fred))
(a ...)
(b ...))
```

## Macro SETF, PSETF

### Syntax:

**setf** {pair}\* => *result*\*

**psetf** {pair}\* => nil

*pair*::= *place* *newvalue*

### Arguments and Values:

*place*---a *place*.

*newvalue*---a form.

*results*---the multiple values[2] returned by the storing form for the last *place*, or nil if there are no *pairs*.

### Description:

**setf** changes the value of *place* to be *newvalue*.

(**setf** *place* *newvalue*) expands into an update form that stores the result of evaluating *newvalue* into

## CLHS: Declaration DYNAMIC-EXTENT

the location referred to by *place*. Some *place* forms involve uses of accessors that take optional arguments. Whether those optional arguments are permitted by setf, or what their use is, is up to the setf expander function and is not under the control of setf. The documentation for any function that accepts &optional, &rest, or . . . . . key arguments and that claims to be usable with setf must specify how those arguments are treated.

If more than one *pair* is supplied, the *pairs* are processed sequentially; that is,

```
(setf place-1 newvalue-1
      place-2 newvalue-2
      ...
      place-N newvalue-N)
```

is precisely equivalent to

```
(progn (setf place-1 newvalue-1)
       (setf place-2 newvalue-2)
       ...
       (setf place-N newvalue-N))
```

For psetf, if more than one *pair* is supplied then the assignments of new values to places are done in parallel. More precisely, all subforms (in both the *place* and *newvalue forms*) that are to be evaluated are evaluated from left to right; after all evaluations have been performed, all of the assignments are performed in an unpredictable order.

For detailed treatment of the expansion of setf and psetf, see [Section 5.1.2 \(Kinds of Places\)](#).

### Examples:

```
(setq x (cons 'a 'b) y (list 1 2 3)) => (1 2 3)
(setf (car x) 'x (cadr y) (car x) (cdr x) y) => (1 X 3)
x => (X 1 X 3)
y => (1 X 3)
(setq x (cons 'a 'b) y (list 1 2 3)) => (1 2 3)
(psetf (car x) 'x (cadr y) (car x) (cdr x) y) => NIL
x => (X 1 A 3)
y => (1 A 3)
```

### Affected By:

define-setf-expander, defsetf, \*macroexpand-hook\*

**Exceptional Situations:** None.

### See Also:

define-setf-expander, defsetf, macroexpand-1, rotatef, shiftf, [Section 5.1 \(Generalized Reference\)](#)

**Notes:** None.

**Macro SHIFTF****Syntax:**

**shiftf** *place+ newvalue => old-value-1*

**Arguments and Values:**

*place*—a *place*.

*newvalue*—a *form*; evaluated.

*old-value-1*—an *object* (the old *value* of the first *place*).

**Description:**

**shiftf** modifies the values of each *place* by storing *newvalue* into the last *place*, and shifting the values of the second through the last *place* into the remaining *places*.

If *newvalue* produces more values than there are store variables, the extra values are ignored. If *newvalue* produces fewer values than there are store variables, the missing values are set to **nil**.

In the form (**shiftf** *place1 place2 ... placen newvalue*), the values in *place1* through *placen* are *read* and saved, and *newvalue* is evaluated, for a total of *n+1* values in all. Values 2 through *n+1* are then stored into *place1* through *placen*, respectively. It is as if all the *places* form a shift register; the *newvalue* is shifted in from the right, all values shift over to the left one place, and the value shifted out of *place1* is returned.

For information about the *evaluation of subforms* of *places*, see [Section 5.1.1.1 \(Evaluation of Subforms to Places\)](#).

**Examples:**

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

(setq x (list 'a 'b 'c)) => (A B C)
(shiftf (cadr x) 'z) => B
x => (A Z C)
(shiftf (cadr x) (caddr x) 'q) => Z
x => (A (C) . Q)
(setq n 0) => 0
(setq x (list 'a 'b 'c 'd)) => (A B C D)
(shiftf (nth (setq n (+ n 1)) x) 'z) => B
x => (A Z C D)
```

**Affected By:**

[\*\*define-setf-expander\*\*](#), [\*\*defsetf\*\*](#), [\*\*\\*macroexpand-hook\\*\*\*](#)

**Exceptional Situations:** None.

**See Also:**

[\*\*setf, rotatef\*\*](#), [Section 5.1 \(Generalized Reference\)](#)

**Notes:**

The effect of (`shiftf place1 place2 ... placen newvalue`) is roughly equivalent to

```
(let ((var1 place1)
      (var2 place2)
      ...
      (varn placen)
      (var0 newvalue))
  (setf place1 var2)
  (setf place2 var3)
  ...
  (setf placen var0)
  var1)
```

except that the latter would evaluate any *subforms* of each `place` twice, whereas `shiftf` evaluates them once. For example,

```
(setq n 0) => 0
(setq x (list 'a 'b 'c 'd)) => (A B C D)
(prog1 (nth (setq n (+ n 1)) x)
        (setf (nth (setq n (+ n 1)) x) 'z)) => B
x => (A B Z D)
```

## Macro STEP

**Syntax:**

**step** *form* => *result\**

**Arguments and Values:**

*form*—a *form*; evaluated as described below.

*results*—the *values* returned by the *form*.

**Description:**

**step** implements a debugging paradigm wherein the programmer is allowed to *step* through the *evaluation* of a *form*. The specific nature of the interaction, including which I/O streams are used and whether the stepping has lexical or dynamic scope, is *implementation-defined*.

**step** evaluates *form* in the current *environment*. A call to **step** can be compiled, but it is acceptable for an implementation to interactively step through only those parts of the computation that are interpreted.

It is technically permissible for a *conforming implementation* to take no action at all other than normal *execution* of the *form*. In such a situation, (`step form`) is equivalent to, for example, (`let () form`). In implementations where this is the case, the associated documentation should mention that fact.

**Examples:** None.**Affected By:** None.**Exceptional Situations:** None.**See Also:**[trace](#)**Notes:**

*Implementations* are encouraged to respond to the typing of ? or the pressing of a "help key" by providing help including a list of commands.

## Macro TIME

**Syntax:****time** *form => result\****Arguments and Values:**

*form*—a form; evaluated as described below.

*results*—the values returned by the *form*.

**Description:**

**time** evaluates *form* in the current environment (lexical and dynamic). A call to **time** can be compiled.

**time** prints various timing data and other information to trace output. The nature and format of the printed information is implementation-defined. Implementations are encouraged to provide such information as elapsed real time, machine run time, and storage management statistics.

**Examples:** None.**Affected By:**

The accuracy of the results depends, among other things, on the accuracy of the corresponding functions provided by the underlying operating system.

The magnitude of the results may depend on the hardware, the operating system, the lisp implementation, and the state of the global environment. Some specific issues which frequently affect the outcome are hardware speed, nature of the scheduler (if any), number of competing processes (if any), system paging, whether the call is interpreted or compiled, whether functions called are compiled, the kind of garbage collector involved and whether it runs, whether internal data structures (e.g., hash tables) are implicitly reorganized, etc.

**Exceptional Situations:** None.**See Also:****Macro TIME**

**get-internal-real-time, get-internal-run-time****Notes:**

In general, these timings are not guaranteed to be reliable enough for marketing comparisons. Their value is primarily heuristic, for tuning purposes.

For useful background information on the complicated issues involved in interpreting timing results, see *Performance and Evaluation of Lisp Programs*.

**Macro TYPECASE, CTYPECASE, ETYPECASE****Syntax:**

**typecase** *keyform* {*normal-clause*}\* [*otherwise-clause*] => *result*\*

**ctypecase** *keyplace* {*normal-clause*}\* => *result*\*

**etypecase** *keyform* {*normal-clause*}\* => *result*\*

*normal-clause*::= (*type form*\*)

*otherwise-clause*::= ({*otherwise* | *t*} *form*\*)

*clause*::= *normal-clause* | *otherwise-clause*

**Arguments and Values:**

*keyform*---a form; evaluated to produce a *test-key*.

*keyplace*---a form; evaluated initially to produce a *test-key*. Possibly also used later as a place if no *types* match.

*test-key*---an object produced by evaluating *keyform* or *keyplace*.

*type*---a type specifier.

*forms*---an implicit progn.

*results*---the values returned by the *forms* in the matching *clause*.

**Description:**

These macros allow the conditional execution of a body of *forms* in a *clause* that is selected by matching the *test-key* on the basis of its type.

The *keyform* or *keyplace* is *evaluated* to produce the *test-key*.

## CLHS: Declaration DYNAMIC-EXTENT

Each of the *normal-clauses* is then considered in turn. If the *test-key* is of the *type* given by the *clauses's type*, the *forms* in that *clause* are *evaluated* as an *implicit progn*, and the *values* it returns are returned as the value of the **typecase**, **ctypecase**, or **etypecase form**.

These *macros* differ only in their *behavior* when no *normal-clause* matches; specifically:

### **typecase**

If no *normal-clause* matches, and there is an *otherwise-clause*, then that *otherwise-clause* automatically matches; the *forms* in that *clause* are *evaluated* as an *implicit progn*, and the *values* it returns are returned as the value of the **typecase**.

If there is no *otherwise-clause*, **typecase** returns **nil**.

### **ctypecase**

If no *normal-clause* matches, a *correctable error* of *type type-error* is signaled. The offending datum is the *test-key* and the expected type is *type equivalent* to (or *type1 type2 ...*). The **store-value restart** can be used to correct the error.

If the **store-value restart** is invoked, its *argument* becomes the new *test-key*, and is stored in *keyplace* as if by (*setf keyplace test-key*). Then **ctypecase** starts over, considering each *clause* anew.

If the **store-value restart** is invoked interactively, the user is prompted for a new *test-key* to use.

The subforms of *keyplace* might be evaluated again if none of the cases holds.

### **etypecase**

If no *normal-clause* matches, a *non-correctable error* of *type type-error* is signaled. The offending datum is the *test-key* and the expected type is *type equivalent* to (or *type1 type2 ...*).

Note that in contrast with **ctypecase**, the caller of **etypecase** may rely on the fact that **etypecase** does not return if a *normal-clause* does not match.

In all three cases, is permissible for more than one *clause* to specify a matching *type*, particularly if one is a *subtype* of another; the earliest applicable *clause* is chosen.

### Examples:

```
;; (Note that the parts of this example which use TYPE-OF
;; are implementation-dependent.)
(defun what-is-it (x)
  (format t "~&~S is ~A.~%" x
          (typecase x
            (float "a float")
            (null "a symbol, boolean false, or the empty list")
            (list "a list")
            (t (format nil "a(n) ~(~A~)" (type-of x))))))
=> WHAT-IS-IT
(map #'what-is-it '(nil (a b) 7.0 7 box))
>> NIL is a symbol, boolean false, or the empty list.
>> (A B) is a list.
>> 7.0 is a float.
>> 7 is a(n) integer.
>> BOX is a(n) symbol.
=> NIL
(setq x 1/3)
```

```
=> 1/3
(ctypecase x
  (integer (* x 4))
  (symbol (symbol-value x)))
>> Error: The value of X, 1/3, is neither an integer nor a symbol.
>> To continue, type :CONTINUE followed by an option number:
>> 1: Specify a value to use instead.
>> 2: Return to Lisp Toplevel.
>> Debug> :CONTINUE 1
>> Use value: 3.7
>> Error: The value of X, 3.7, is neither an integer nor a symbol.
>> To continue, type :CONTINUE followed by an option number:
>> 1: Specify a value to use instead.
>> 2: Return to Lisp Toplevel.
>> Debug> :CONTINUE 1
>> Use value: 12
=> 48
x => 12
```

**Affected By:**

ctypecase and etypecase, since they might signal an error, are potentially affected by existing *handlers* and \*debug-io\*.

**Exceptional Situations:**

ctypecase and etypecase signal an error of *type type-error* if no *normal-clause* matches.

The compiler may choose to issue a warning of *type style-warning* if a *clause* will never be selected because it is completely shadowed by earlier clauses.

**See Also:**

case, cond, setf, Section 5.1 (Generalized Reference)

**Notes:**

```
(typecase test-key
  { (type form*) }*)
===
(let ((#1=#:g0001 test-key))
  (cond { ((typep #1# 'type) form*) }*))
```

The specific error message used by etypecase and ctypecase can vary between implementations. In situations where control of the specific wording of the error message is important, it is better to use typecase with an *otherwise-clause* that explicitly signals an error with an appropriate message.

**Macro TRACE, UNTRACE****Syntax:**

**trace** *function-name\** => *trace-result*

**untrace** *function-name\** => *untrace-result*

#### Arguments and Values:

*function-name*—a *function name*.

*trace-result*—*implementation-dependent*, unless no *function-names* are supplied, in which case *trace-result* is a *list of function names*.

*untrace-result*—*implementation-dependent*.

#### Description:

**trace** and **untrace** control the invocation of the trace facility.

Invoking **trace** with one or more *function-names* causes the denoted *functions* to be "traced." Whenever a traced *function* is invoked, information about the call, about the arguments passed, and about any eventually returned values is printed to *trace output*. If **trace** is used with no *function-names*, no tracing action is performed; instead, a list of the *functions* currently being traced is returned.

Invoking **untrace** with one or more function names causes those functions to be "untraced" (i.e., no longer traced). If **untrace** is used with no *function-names*, all *functions* currently being traced are untraced.

If a *function* to be traced has been open-coded (e.g., because it was declared **inline**), a call to that *function* might not produce trace output.

#### Examples:

```
(defun fact (n) (if (zerop n) 1 (* n (fact (- n 1)))))
=> FACT
  (trace fact)
=> (FACT)
;; Of course, the format of traced output is implementation-dependent.
(fact 3)
>> 1 Enter FACT 3
>> | 2 Enter FACT 2
>> |   3 Enter FACT 1
>> |     4 Enter FACT 0
>> |     4 Exit FACT 1
>> |   3 Exit FACT 1
>> | 2 Exit FACT 2
>> 1 Exit FACT 6
=> 6
```

#### Side Effects:

Might change the definitions of the *functions* named by *function-names*.

#### Affected By:

Whether the functions named are defined or already being traced.

#### Exceptional Situations:

## CLHS: Declaration DYNAMIC-EXTENT

Tracing an already traced function, or untracing a function not currently being traced, should produce no harmful effects, but might signal a warning.

### See Also:

**\*trace-output\*, step**

### Notes:

**trace** and **untrace** may also accept additional *implementation-dependent* argument formats. The format of the trace output is *implementation-dependent*.

Although **trace** can be extended to permit non-standard options, *implementations* are nevertheless encouraged (but not required) to warn about the use of syntax or options that are neither specified by this standard nor added as an extension by the *implementation*, since they could be symptomatic of typographical errors or of reliance on features supported in *implementations* other than the current *implementation*.

## Macro WITH-ACCESSORS

### Syntax:

**with-accessors** (*slot-entry\**) *instance-form declaration\* form\**

=> *result\**

*slot-entry*::= (*variable-name accessor-name*)

### Arguments and Values:

*variable-name*---a *variable name*; not evaluated.

*accessor-name*---a *function name*; not evaluated.

*instance-form*---a *form*; evaluated.

*declaration*---a *declare expression*; not evaluated.

*forms*---an *implicit progn*.

*results*---the *values* returned by the *forms*.

### Description:

Creates a lexical environment in which the slots specified by *slot-entry* are lexically available through their accessors as if they were variables. The macro **with-accessors** invokes the appropriate accessors to *access* the *slots* specified by *slot-entry*. Both **setf** and **setq** can be used to set the value of the *slot*.

### Examples:

```
(defclass thing ()  
  ((x :initarg :x :accessor thing-x))
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(y :initarg :y :accessor thing-y)))
=> #<STANDARD-CLASS THING 250020173>
(defmethod (setf thing-x) :before (new-x (thing thing))
  (format t "~&Changing X from ~D to ~D in ~S.~%" 
          (thing-x thing) new-x thing))
(setq thing1 (make-instance 'thing :x 1 :y 2)) => #<THING 43135676>
(setq thing2 (make-instance 'thing :x 7 :y 8)) => #<THING 43147374>
(with-accessors ((x1 thing-x) (y1 thing-y))
  thing1
  (with-accessors ((x2 thing-x) (y2 thing-y))
    thing2
    (list (list x1 (thing-x thing1) y1 (thing-y thing1)
                x2 (thing-x thing2) y2 (thing-y thing2))
      (setq x1 (+ y1 x2))
      (list x1 (thing-x thing1) y1 (thing-y thing1)
            x2 (thing-x thing2) y2 (thing-y thing2))
      (setf (thing-x thing2) (list x1))
      (list x1 (thing-x thing1) y1 (thing-y thing1)
            x2 (thing-x thing2) y2 (thing-y thing2))))))
>> Changing X from 1 to 9 in #<THING 43135676>.
>> Changing X from 7 to (9) in #<THING 43147374>.
=> ((1 1 2 2 7 7 8 8)
   9
   (9 9 2 2 7 7 8 8)
   (9)
   (9 9 2 2 (9) (9) 8 8))
```

### Affected By:

#### defclass

#### Exceptional Situations:

The consequences are undefined if any *accessor-name* is not the name of an accessor for the *instance*.

#### See Also:

#### with-slots, symbol-macrolet

#### Notes:

A with-accessors expression of the form:

```
(with-accessors (slot-entry1 ... slot-entryn) instance-form form1 ... formk)
```

expands into the equivalent of

```
(let ((in instance-form))
  (symbol-macrolet (Q1 ... Qn) form1 ... formk))
```

where *Qi* is

```
(variable-namei () (accessor-namei in))
```

***Macro WITH-CONDITION-RESTARTS*****Syntax:**

**with-condition-restarts** *condition-form* *restarts-form* *form\**

=> *result\**

**Arguments and Values:**

*condition-form*—a form; evaluated to produce a *condition*.

*condition*—a condition object resulting from the evaluation of *condition-form*.

*restart-form*—a form; evaluated to produce a *restart-list*.

*restart-list*—a list of restart objects resulting from the evaluation of *restart-form*.

*forms*—an implicit progn; evaluated.

*results*—the values returned by *forms*.

**Description:**

First, the *condition-form* and *restarts-form* are evaluated in normal left-to-right order; the primary values yielded by these evaluations are respectively called the *condition* and the *restart-list*.

Next, the *forms* are evaluated in a dynamic environment in which each restart in *restart-list* is associated with the *condition*. See [Section 9.1.4.2.4 \(Associating a Restart with a Condition\)](#).

**Examples:** None.

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[restart-case](#)

**Notes:**

Usually this macro is not used explicitly in code, since [restart-case](#) handles most of the common cases in a way that is syntactically more concise.

***Macro WITH-COMPILE-UNIT*****Syntax:**

## CLHS: Declaration DYNAMIC-EXTENT

**with-compilation-unit** (*[option]*) *form\** => *result\**

*option*::= :override override

### Arguments and Values:

*override*—a *generalized boolean*; evaluated. The default is **nil**.

*forms*—an *implicit progn.*

*results*—the *values* returned by the *forms*.

### Description:

Executes *forms* from left to right. Within the *dynamic environment* of **with-compilation-unit**, actions deferred by the compiler until the end of compilation will be deferred until the end of the outermost call to **with-compilation-unit**.

The set of *options* permitted may be extended by the implementation, but the only *standardized* keyword is :override.

If nested dynamically only the outer call to **with-compilation-unit** has any effect unless the value associated with :override is *true*, in which case warnings are deferred only to the end of the innermost call for which *override* is *true*.

The function **compile-file** provides the effect of

```
(with-compilation-unit (:override nil) ...)
```

around its *code*.

Any *implementation-dependent* extensions can only be provided as the result of an explicit programmer request by use of an *implementation-dependent* keyword. *Implementations* are forbidden from attaching additional meaning to a use of this macro which involves either no keywords or just the keyword :override.

### Examples:

If an *implementation* would normally defer certain kinds of warnings, such as warnings about undefined functions, to the end of a compilation unit (such as a *file*), the following example shows how to cause those warnings to be deferred to the end of the compilation of several files.

```
(defun compile-files (&rest files)
  (with-compilation-unit ()
    (mapcar #'(lambda (file) (compile-file file)) files)))

(compile-files "A" "B" "C")
```

Note however that if the implementation does not normally defer any warnings, use of **with-compilation-unit** might not have any effect.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[compile](#), [compile-file](#)

**Notes:** None.

## Macro WITH-HASH-TABLE-ITERATOR

**Syntax:**

**with-hash-table-iterator** (*name hash-table*) *declaration\** *form\** => *result\**

**Arguments and Values:**

*name*—a name suitable for the first argument to [macrolet](#).

*hash-table*—a *form*, evaluated once, that should produce a [hash table](#).

*declaration*—a [declare expression](#); not evaluated.

*forms*—an [implicit progn](#).

*results*—the [values](#) returned by *forms*.

**Description:**

Within the lexical scope of the body, *name* is defined via [macrolet](#) such that successive invocations of (*name*) return the items, one by one, from the [hash table](#) that is obtained by evaluating *hash-table* only once.

An invocation (*name*) returns three values as follows:

1. A [generalized boolean](#) that is [true](#) if an entry is returned.
2. The key from the hash-table entry.
3. The value from the hash-table entry.

After all entries have been returned by successive invocations of (*name*), then only one value is returned, namely [nil](#).

It is unspecified what happens if any of the implicit interior state of an iteration is returned outside the dynamic extent of the [with-hash-table-iterator form](#) such as by returning some [closure](#) over the invocation *form*.

Any number of invocations of [with-hash-table-iterator](#) can be nested, and the body of the innermost one can invoke all of the locally *established macros*, provided all of those *macros* have [distinct](#) names.

**Examples:**

## CLHS: Declaration DYNAMIC-EXTENT

The following function should return `t` on any *hash table*, and signal an error if the usage of **with-hash-table-iterator** does not agree with the corresponding usage of **maphash**.

```
(defun test-hash-table-iterator (hash-table)
  (let ((all-entries '())
        (generated-entries '())
        (unique (list nil)))
    (maphash #'(lambda (key value) (push (list key value) all-entries))
             hash-table)
    (with-hash-table-iterator (generator-fn hash-table)
      (loop
        (multiple-value-bind (more? key value) (generator-fn)
          (unless more? (return))
          (unless (eql value (gethash key hash-table unique))
            (error "Key ~S not found for value ~S" key value))
          (push (list key value) generated-entries)))
      (unless (= (length all-entries)
                 (length generated-entries)
                 (length (union all-entries generated-entries
                               :key #'car :test (hash-table-test hash-table))))
        (error "Generated entries and Maphash entries don't correspond")))
    t))
```

The following could be an acceptable definition of **maphash**, implemented by **with-hash-table-iterator**.

```
(defun maphash (function hash-table)
  (with-hash-table-iterator (next-entry hash-table)
    (loop (multiple-value-bind (more key value) (next-entry)
            (unless more (return nil))
            (funcall function key value))))
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:**

The consequences are undefined if the local function named *name established by with-hash-table-iterator* is called after it has returned *false* as its *primary value*.

**See Also:**

[Section 3.6 \(Traversal Rules and Side Effects\)](#)

**Notes:** None.

## Macro WITH-INPUT-FROM-STRING

**Syntax:**

**with-input-from-string** (*var string &key index start end declaration\* form\**)

=> *result\**

**Arguments and Values:**

Macro WITH-INPUT-FROM-STRING

634

*var*---a variable name.

*string*---a form; evaluated to produce a string.

*index*---a place.

*start*, *end*---bounding index designators of *string*. The defaults for *start* and *end* are 0 and nil, respectively.  
*declaration*---a declare expression; not evaluated.

*forms*---an implicit progn.

*result*---the values returned by the *forms*.

### Description:

Creates an input string stream, provides an opportunity to perform operations on the stream (returning zero or more values), and then closes the string stream.

*String* is evaluated first, and *var* is bound to a character input string stream that supplies characters from the subsequence of the resulting string bounded by *start* and *end*. The body is executed as an implicit progn.

The input string stream is automatically closed on exit from with-input-from-string, no matter whether the exit is normal or abnormal. The input string stream to which the variable *var* is bound has dynamic extent; its extent ends when the form is exited.

The *index* is a pointer within the *string* to be advanced. If with-input-from-string is exited normally, then *index* will have as its value the index into the *string* indicating the first character not read which is (*length string*) if all characters were used. The place specified by *index* is not updated as reading progresses, but only at the end of the operation.

*start* and *index* may both specify the same variable, which is a pointer within the *string* to be advanced, perhaps repeatedly by some containing loop.

The consequences are undefined if an attempt is made to assign the variable *var*.

### 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
(with-input-from-string (s "Animal Crackers" :index j :start 6)
  (read s)) => CRACKERS
```

The variable *j* is set to 15.

### Side Effects:

The value of the place named by *index*, if any, is modified.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[make-string-input-stream](#), [Section 3.6 \(Traversal Rules and Side Effects\)](#)

**Notes:** None.

## Macro WITH-OPEN-STREAM

**Syntax:**

**with-open-stream** (*var stream*) *declaration\** *form\**

=> *result\**

**Arguments and Values:**

*var*—a *variable name*.

*stream*—a *form*; evaluated to produce a *stream*.

*declaration*—a *declare expression*; not evaluated.

*forms*—an *implicit progn*.

*results*—the *values* returned by the *forms*.

**Description:**

**with-open-stream** performs a series of operations on *stream*, returns a value, and then closes the *stream*.

*Var* is bound to the value of *stream*, and then *forms* are executed as an *implicit progn*. *stream* is automatically closed on exit from **with-open-stream**, no matter whether the exit is normal or abnormal. The *stream* has *dynamic extent*; its *extent* ends when the *form* is exited.

The consequences are undefined if an attempt is made to *assign* the the *variable var* with the *forms*.

**Examples:**

```
(with-open-stream (s (make-string-input-stream "1 2 3 4"))
  (+ (read s) (read s) (read s))) => 6
```

**Side Effects:**

The *stream* is closed (upon exit).

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:****close****Notes:** None.**macro WITH-OPEN-FILE****Syntax:****with-open-file** (*stream filespec options\**) *declaration\* form\***=> results***Arguments and Values:***stream* — a variable.*filespec*—a pathname designator.*options* — forms; evaluated.*declaration*—a declare expression; not evaluated.*forms*—an implicit progn.*results*—the values returned by the *forms*.**Description:**

**with-open-file** uses open to create a file stream to file named by *filespec*. *Filespec* is the name of the file to be opened. *Options* are used as keyword arguments to open.

The stream object to which the stream variable is bound has dynamic extent; its extent ends when the form is exited.

**with-open-file** evaluates the *forms* as an implicit progn with *stream* bound to the value returned by open.

When control leaves the body, either normally or abnormally (such as by use of throw), the file is automatically closed. If a new output file is being written, and control leaves abnormally, the file is aborted and the file system is left, so far as possible, as if the file had never been opened.

It is possible by the use of :if-exists nil or :if-does-not-exist nil for *stream* to be bound to nil. Users of :if-does-not-exist nil should check for a valid stream.

The consequences are undefined if an attempt is made to assign the stream variable. The compiler may choose to issue a warning if such an attempt is detected.

**Examples:**

```
(setq p (merge-pathnames "test"))
```

## CLHS: Declaration DYNAMIC-EXTENT

```
=> #<PATHNAME :HOST NIL :DEVICE device-name :DIRECTORY directory-name
    :NAME "test" :TYPE NIL :VERSION :NEWEST>
  (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.

;; Normally one would not do this intentionally because it is
;; not perspicuous, but beware when using :IF-DOES-NOT-EXIST NIL
;; that this doesn't happen to you accidentally...
  (with-open-file (foo "no-such-file" :if-does-not-exist nil)
    (read foo))
>> hello?
=> HELLO? ;This value was read from the terminal, not a file!

;; Here's another bug to avoid...
  (with-open-file (foo "no-such-file" :direction :output :if-does-not-exist nil)
    (format foo "Hello"))
=> "Hello" ;FORMAT got an argument of NIL!
```

### Side Effects:

Creates a stream to the file named by filename (upon entry), and closes the stream (upon exit). In some implementations, the file might be locked in some way while it is open. If the stream is an output stream, a file might be created.

### Affected By:

The host computer's file system.

### Exceptional Situations:

See the function open.

### See Also:

open, close, pathname, logical-pathname, Section 19.1.2 (Pathnames as Filenames)

**Notes:** None.

## Macro WITH-OUTPUT-TO-STRING

### Syntax:

```
with-output-to-string (var &optional string-form &key element-type) declaration* form*
=> result*
```

### Arguments and Values:

*var*—a *variable name*.

*string-form*—a *form* or **nil**; if *non-nil*, evaluated to produce *string*.

*string*—a *string* that has a *fill pointer*.

*element-type*—a *type specifier*; evaluated. The default is **character**.

*declaration*—a **declare expression**; not evaluated.

*forms*—an *implicit progn*.

*results*—If a *string-form* is not supplied or **nil**, a *string*; otherwise, the *values* returned by the *forms*.

### Description:

**with-output-to-string** creates a character *output stream*, performs a series of operations that may send results to this *stream*, and then closes the *stream*.

The *element-type* names the *type* of the elements of the *stream*; a *stream* is constructed of the most specialized *type* that can accommodate elements of the given *type*.

The body is executed as an *implicit progn* with *var* bound to an *output string stream*. All output to that *string stream* is saved in a *string*.

If *string* is supplied, *element-type* is ignored, and the output is incrementally appended to *string* as if by use of **vector-push-extend**.

The *output stream* is automatically closed on exit from **with-output-from-string**, no matter whether the exit is normal or abnormal. The *output string stream* to which the *variable var* is *bound* has *dynamic extent*; its *extent* ends when the *form* is exited.

If no *string* is provided, then **with-output-from-string** produces a *stream* that accepts characters and returns a *string* of the indicated *element-type*. If *string* is provided, **with-output-to-string** returns the results of evaluating the last *form*.

The consequences are undefined if an attempt is made to *assign* the *variable var*.

### Examples:

```
(setq fstr (make-array '(0) :element-type 'base-char
                         :fill-pointer 0 :adjustable t)) => ""
(with-output-to-string (s fstr)
  (format s "here's some output")
  (input-stream-p s)) => false
fstr => "here's some output"
```

### Side Effects:

The *string* is modified.

**Affected By:** None.

**Exceptional Situations:**

The consequences are undefined if destructive modifications are performed directly on the *string* during the *dynamic extent* of the call.

**See Also:**

[make-string-output-stream](#), [vector-push-extend](#), Section 3.6 (Traversal Rules and Side Effects)

**Notes:** None.

**Macro WITH-PACKAGE-ITERATOR****Syntax:**

**with-package-iterator** (*name package-list-form &rest symbol-types*) *declaration\** *form\**

=> *result\**

**Arguments and Values:**

*name*—a symbol.

*package-list-form*—a form; evaluated once to produce a *package-list*.

*package-list*—a designator for a list of package designators.

*symbol-type*—one of the symbols :internal, :external, or :inherited.

*declaration*—a declare expression; not evaluated.

*forms*—an implicit progn.

*results*—the values of the *forms*.

**Description:**

Within the lexical scope of the body *forms*, the *name* is defined via macrolet such that successive invocations of (*name*) will return the symbols, one by one, from the packages in *package-list*.

It is unspecified whether symbols inherited from multiple packages are returned more than once. The order of symbols returned does not necessarily reflect the order of packages in *package-list*. When *package-list* has more than one element, it is unspecified whether duplicate symbols are returned once or more than once.

*Symbol-types* controls which symbols that are accessible in a package are returned as follows:

:internal

The symbols that are present in the package, but that are not exported.

:external

The symbols that are present in the package and are exported.

:inherited

## CLHS: Declaration DYNAMIC-EXTENT

The symbols that are exported by used packages and that are not shadowed.

When more than one argument is supplied for symbol-types, a symbol is returned if its accessibility matches any one of the symbol-types supplied. Implementations may extend this syntax by recognizing additional symbol accessibility types.

An invocation of (*name*) returns four values as follows:

1. A flag that indicates whether a symbol is returned (true means that a symbol is returned).
2. A symbol that is accessible in one the indicated packages.
3. The accessibility type for that symbol; i.e., one of the symbols :internal, :external, or :inherited.
4. The package from which the symbol was obtained. The package is one of the packages present or named in package-list.

After all symbols have been returned by successive invocations of (*name*), then only one value is returned, namely nil.

The meaning of the second, third, and fourth values is that the returned symbol is accessible in the returned package in the way indicated by the second return value as follows:

- :internal  
Means present and not exported.
- :external  
Means present and exported.
- :inherited  
Means not present (thus not shadowed) but inherited from some used package.

It is unspecified what happens if any of the implicit interior state of an iteration is returned outside the dynamic extent of the with-package-iterator form such as by returning some closure over the invocation form.

Any number of invocations of with-package-iterator can be nested, and the body of the innermost one can invoke all of the locally *established macros*, provided all those macros have distinct names.

### Examples:

The following function should return t on any package, and signal an error if the usage of with-package-iterator does not agree with the corresponding usage of do-symbols.

```
(defun test-package-iterator (package)
  (unless (packagep package)
    (setq package (find-package package)))
  (let ((all-entries '()))
    (generated-entries '())
    (do-symbols (x package)
      (multiple-value-bind (symbol accessibility)
          (find-symbol (symbol-name x) package)
        (push (list symbol accessibility) all-entries)))
    (with-package-iterator (generator-fn package
                                         :internal :external :inherited)
      (loop
        (multiple-value-bind (more? symbol accessibility pkg)
          ()))
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(generator-fn)
(unless more? (return))
(let ((l (multiple-value-list (find-symbol (symbol-name symbol)
                                             package))))
  (unless (equal l (list symbol accessibility))
    (error "Symbol ~S not found as ~S in package ~A [~S]"
           symbol accessibility (package-name package) l))
  (push l generated-entries)))
(unless (and (subsetp all-entries generated-entries :test #'equal)
             (subsetp generated-entries all-entries :test #'equal))
  (error "Generated entries and Do-Symbols entries don't correspond"))
t))
```

The following function prints out every *present symbol* (possibly more than once):

```
(defun print-all-symbols ()
  (with-package-iterator (next-symbol (list-all-packages)
                                       :internal :external)
    (loop
      (multiple-value-bind (more? symbol) (next-symbol)
        (if more?
            (print symbol)
            (return))))))
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:**

**with-package-iterator** signals an error of *type program-error* if no *symbol-types* are supplied or if a *symbol-type* is not recognized by the implementation is supplied.

The consequences are undefined if the local function named *name established* by **with-package-iterator** is called after it has returned *false* as its *primary value*.

**See Also:**

[Section 3.6 \(Traversal Rules and Side Effects\)](#)

**Notes:** None.

## Macro WITH-SLOTS

**Syntax:**

**with-slots** (*slot-entry\**) *instance-form declaration\* form\**

=> *result\**

```
slot-entry ::= slot-name | (variable-name slot-name)
```

**Arguments and Values:**

## CLHS: Declaration DYNAMIC-EXTENT

*slot-name*---a slot name; not evaluated.

*variable-name*---a variable name; not evaluated.

*instance-form*---a form; evaluated to produce *instance*.

*instance*---an object.

*declaration*---a declare expression; not evaluated.

*forms*---an implicit progn.

*results*---the values returned by the *forms*.

### Description:

The macro with-slots establishes a lexical environment for referring to the slots in the *instance* named by the given *slot-names* as though they were variables. Within such a context the value of the slot can be specified by using its slot name, as if it were a lexically bound variable. Both setf and setq can be used to set the value of the slot.

The macro with-slots translates an appearance of the slot name as a variable into a call to slot-value.

### Examples:

```
(defclass thing ()
  ((x :initarg :x :accessor thing-x)
   (y :initarg :y :accessor thing-y)))
=> #<STANDARD-CLASS THING 250020173>
(defmethod (setf thing-x) :before (new-x (thing thing))
  (format t "~&Changing X from ~D to ~D in ~S.~%" 
          (thing-x thing) new-x thing))
(setq thing (make-instance 'thing :x 0 :y 1)) => #<THING 62310540>
(with-slots (x y) thing (incf x) (incf y)) => 2
(values (thing-x thing) (thing-y thing)) => 1, 2
(setq thing1 (make-instance 'thing :x 1 :y 2)) => #<THING 43135676>
(setq thing2 (make-instance 'thing :x 7 :y 8)) => #<THING 43147374>
(with-slots ((x1 x) (y1 y))
  thing1
  (with-slots ((x2 x) (y2 y))
    thing2
    (list (list x1 (thing-x thing1) y1 (thing-y thing1)
                x2 (thing-x thing2) y2 (thing-y thing2))
      (setq x1 (+ y1 x2))
      (list x1 (thing-x thing1) y1 (thing-y thing1)
            x2 (thing-x thing2) y2 (thing-y thing2))
      (setf (thing-x thing2) (list x1))
      (list x1 (thing-x thing1) y1 (thing-y thing1)
            x2 (thing-x thing2) y2 (thing-y thing2))))))
>> Changing X from 7 to (9) in #<THING 43147374>.
=> ((1 1 2 2 7 7 8 8)
  9
  (9 9 2 2 7 7 8 8)
  (9)
  (9 9 2 2 (9) (9) 8 8))
```

**Affected By:****defclass****Exceptional Situations:**

The consequences are undefined if any *slot-name* is not the name of a *slot* in the *instance*.

**See Also:****with-accessors, slot-value, symbol-macrolet****Notes:**

A with-slots expression of the form:

```
(with-slots (slot-entry1 ... slot-entryn) instance-form form1 ... formk)
```

expands into the equivalent of

```
(let ((in instance-form))
  (symbol-macrolet (Q1 ... Qn) form1 ... formk))
```

where *Qi* is

```
(slot-entryi () (slot-value in 'slot-entryi))
```

if *slot-entryi* is a symbol and is

```
(variable-namei () (slot-value in 'slot-namei))
```

if *slot-entryi* is of the form

```
(variable-namei 'slot-namei))
```

**Macro WITH-SIMPLE-RESTART****Syntax:**

**with-simple-restart** (*name* *format-control* *format-argument*\* ) *form*\*

=> *result*\*

**Arguments and Values:**

*name*—a symbol.

*format-control*—a format control.

*format-argument*—an object (i.e., a format argument).

*forms*—an implicit progn.

*results*—in the normal situation, the values returned by the *forms*; in the exceptional situation where the restart named *name* is invoked, two values—nil and t.

### Description:

with-simple-restart establishes a restart.

If the restart designated by *name* is not invoked while executing *forms*, all values returned by the last of *forms* are returned. If the restart designated by *name* is invoked, control is transferred to with-simple-restart, which returns two values, nil and t.

If *name* is nil, an anonymous restart is established.

The *format-control* and *format-arguments* are used report the restart.

### Examples:

```
(defun read-eval-print-loop (level)
  (with-simple-restart (abort "Exit command level ~D." level)
    (loop
      (with-simple-restart (abort "Return to command level ~D." level)
        (let ((form (prog2 (fresh-line) (read) (fresh-line)))
              (prinl (eval form))))))
    => READ-EVAL-PRINT-LOOP
  (read-eval-print-loop 1)
  (+ 'a 3)
  >> Error: The argument, A, to the function + was of the wrong type.
  >>           The function expected a number.
  >> To continue, type :CONTINUE followed by an option number:
  >> 1: Specify a value to use this time.
  >> 2: Return to command level 1.
  >> 3: Exit command level 1.
  >> 4: Return to Lisp Toplevel.

  (defun compute-fixnum-power-of-2 (x)
    (with-simple-restart (nil "Give up on computing 2^~D." x)
      (let ((result 1))
        (dotimes (i x result)
          (setq result (* 2 result))
          (unless (fixnump result)
            (error "Power of 2 is too large."))))
    COMPUTE-FIXNUM-POWER-OF-2
    (defun compute-power-of-2 (x)
      (or (compute-fixnum-power-of-2 x) 'something big))
  COMPUTE-POWER-OF-2
  (compute-power-of-2 10)
  1024
  (compute-power-of-2 10000)
  >> Error: Power of 2 is too large.
  >> To continue, type :CONTINUE followed by an option number.
  >> 1: Give up on computing 2^10000.
  >> 2: Return to Lisp Toplevel
  >> Debug> :continue 1
  => SOMETHING-BIG
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[restart-case](#)

**Notes:**

[with-simple-restart](#) is shorthand for one of the most common uses of [restart-case](#).

[with-simple-restart](#) could be defined by:

```
(defmacro with-simple-restart ((restart-name format-control
                                &rest format-arguments)
                               &body forms)
  `(restart-case (progn ,@forms)
    (,restart-name ()
      :report (lambda (stream)
                 (format stream ,format-control ,@format-arguments)))
    (values nil t)))
```

Because the second return value is `t` in the exceptional case, it is common (but not required) to arrange for the second return value in the normal case to be missing or `nil` so that the two situations can be distinguished.

## Macro WITH-STANDARD-IO-SYNTAX

**Syntax:**

`with-standard-io-syntax form* => result*`

**Arguments and Values:**

*forms*—an [implicit progn](#).

*results*—the [values](#) returned by the [forms](#).

**Description:**

Within the dynamic extent of the body of *forms*, all reader/print control variables, including any [implementation-defined](#) ones not specified by this standard, are bound to values that produce standard read/print behavior. The values for the variables specified by this standard are listed in the next figure.

Variable	Value
<code>*package*</code>	The CL-USER package
<code>*print-array*</code>	<code>t</code>
<code>*print-base*</code>	10
<code>*print-case*</code>	:upcase
<code>*print-circle*</code>	<code>nil</code>
<code>*print-escape*</code>	<code>t</code>

<code>*print-gensym*</code>	<code>t</code>
<code>*print-length*</code>	<code>nil</code>
<code>*print-level*</code>	<code>nil</code>
<code>*print-lines*</code>	<code>nil</code>
<code>*print-miser-width*</code>	<code>nil</code>
<code>*print-pprint-dispatch*</code>	The <u>standard pprint dispatch table</u>
<code>*print-pretty*</code>	<code>nil</code>
<code>*print-radix*</code>	<code>nil</code>
<code>*print-readably*</code>	<code>t</code>
<code>*print-right-margin*</code>	<code>nil</code>
<code>*read-base*</code>	<code>10</code>
<code>*read-default-float-format*</code>	<u>single-float</u>
<code>*read-eval*</code>	<code>t</code>
<code>*read-suppress*</code>	<code>nil</code>
<code>*readtable*</code>	The <u>standard readtable</u>

**Figure 23–1. Values of standard control variables****Examples:**

```
(with-open-file (file pathname :direction :output)
  (with-standard-io-syntax
    (print data file)))

;;; ... Later, in another Lisp:

(with-open-file (file pathname :direction :input)
  (with-standard-io-syntax
    (setq data (read file))))
```

**Affected By:** None.**Exceptional Situations:** None.**See Also:** None.**Notes:** None.***Macro WHEN, UNLESS*****Syntax:****when** *test-form form\* => result\****unless** *test-form form\* => result\****Arguments and Values:***test-form*—a form.*forms*—an implicit progn.*results*—the values of the forms in a when form if the test-form yields true or in an unless form if the test-form yields false; otherwise nil.

**Description:**

**when** and **unless** allow the execution of *forms* to be dependent on a single *test-form*.

In a **when form**, if the *test-form* *yields true*, the *forms* are *evaluated* in order from left to right and the *values* returned by the *forms* are returned from the **when form**. Otherwise, if the *test-form* *yields false*, the *forms* are not *evaluated*, and the **when form** returns **nil**.

In an **unless form**, if the *test-form* *yields false*, the *forms* are *evaluated* in order from left to right and the *values* returned by the *forms* are returned from the **unless form**. Otherwise, if the *test-form* *yields true*, the *forms* are not *evaluated*, and the **unless form** returns **nil**.

**Examples:**

```
(when t 'hello) => HELLO
(unless t 'hello) => NIL
(when nil 'hello) => NIL
(unless nil 'hello) => HELLO
(when t) => NIL
(unless nil) => NIL
(when t (prinl 1) (prinl 2) (prinl 3))
>> 123
=> 3
(unless t (prinl 1) (prinl 2) (prinl 3)) => NIL
(when nil (prinl 1) (prinl 2) (prinl 3)) => NIL
(unless nil (prinl 1) (prinl 2) (prinl 3))
>> 123
=> 3
(let ((x 3))
  (list (when (oddp x) (incf x) (list x))
        (when (oddp x) (incf x) (list x))
        (unless (oddp x) (incf x) (list x))
        (unless (oddp x) (incf x) (list x))
        (if (oddp x) (incf x) (list x))
        (if (oddp x) (incf x) (list x))
        (if (not (oddp x)) (incf x) (list x))
        (if (not (oddp x)) (incf x) (list x))))
=> ((4) NIL (5) NIL 6 (6) 7 (7)))
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**and, cond, if, or**

**Notes:**

```
(when test {form}+) == (and test (progn {form}+))
(when test {form}+) == (cond (test {form}+))
(when test {form}+) == (if test (progn {form}+) nil)
(when test {form}+) == (unless (not test) {form}+)
(unless test {form}+) == (cond ((not test) {form}+))
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(unless test {form}+) == (if test nil (progn {form}+))
(unless test {form}+) == (when (not test) {form}+)
```

## **Restart ABORT**

### **Data Arguments Required:**

None.

### **Description:**

The intent of the **abort** restart is to allow return to the innermost "command level." Implementors are encouraged to make sure that there is always a restart named **abort** around any user code so that user code can call **abort** at any time and expect something reasonable to happen; exactly what the reasonable thing is may vary somewhat. Typically, in an interactive listener, the invocation of **abort** returns to the *Lisp reader* phase of the *Lisp read–eval–print loop*, though in some batch or multi-processing situations there may be situations in which having it kill the running process is more appropriate.

### **See Also:**

## **Restart CONTINUE**

### **Data Arguments Required:**

None.

### **Description:**

The **continue** restart is generally part of protocols where there is a single "obvious" way to continue, such as **in-break** and **cerror**. Some user-defined protocols may also wish to incorporate it for similar reasons. In general, however, it is more reliable to design a special purpose restart with a name that more directly suits the particular application.

### **Examples:**

```
(let ((x 3))
  (handler-bind ((error #'(lambda (c)
                           (let ((r (find-restart 'continue c)))
                             (when r (invoke-restart r))))))
    (cond ((not (floatp x))
           (cerror "Try floating it." "~D is not a float." x)
           (float x))
          (t x))) => 3.0
```

### **See Also:**

## **Restart MUFLLE–WARNING**

### **Data Arguments Required:**

None.

**Description:**

This restart is established by warn so that handlers of warning conditions have a way to tell warn that a warning has already been dealt with and that no further action is warranted.

**Examples:**

```
(defvar *all-quiet* nil) => *ALL-QUIET*
(defvar *saved-warnings* '()) => *SAVED-WARNINGS*
(defun quiet-warning-handler (c)
  (when *all-quiet*
    (let ((r (find-restart 'muffle-warning c)))
      (when r
        (push c *saved-warnings*)
        (invoke-restart r))))
  => CUSTOM-WARNING-HANDLER
(defmacro with-quiet-warnings (&body forms)
  `(let ((*all-quiet* t)
         (*saved-warnings* '()))
    (handler-bind ((warning #'quiet-warning-handler)
                  ,@forms
                  *saved-warnings*)))
  => WITH-QUIET-WARNINGS
  (setq saved
        (with-quiet-warnings
          (warn "Situation #1.")
          (let ((*all-quiet* nil))
            (warn "Situation #2."))
          (warn "Situation #3.")))
  >> Warning: Situation #2.
  => (#<SIMPLE-WARNING 42744421> #<SIMPLE-WARNING 42744365>)
  (dolist (s saved) (format t "~&~A~%" s))
  >> Situation #3.
  >> Situation #1.
  => NIL
```

**See Also:*****Restart STORE-VALUE*****Data Arguments Required:**

a value to use instead (on an ongoing basis).

**Description:**

The store-value restart is generally used by handlers trying to recover from errors of types such as cell-error or type-error, which may wish to supply a replacement datum to be stored permanently.

**Examples:**

```
(defun type-error-auto-coerce (c)
  (when (typep c 'type-error)
    (let ((r (find-restart 'store-value c)))
      (handler-case (let ((v (coerce (type-error-datum c)
                                    (type-error-expected-type c))))

```

## CLHS: Declaration DYNAMIC-EXTENT

```
(invoke-restart r v))  
(error ())))) => TYPE-ERROR-AUTO-COERCE  
(let ((x 3))  
  (handler-bind ((type-error #'type-error-auto-coerce))  
    (check-type x float)  
    x)) => 3.0
```

See Also:

### **Restart USE-VALUE**

**Data Arguments Required:**

a value to use instead (once).

**Description:**

The use-value restart is generally used by handlers trying to recover from errors of types such as cell-error, where the handler may wish to supply a replacement datum for one-time use.

See Also:

### **Special Operator BLOCK**

**Syntax:**

**block** *name* *form\** => *result\**

**Arguments and Values:**

*name*—a symbol.

*form*—a form.

*results*—the values of the forms if a normal return occurs, or else, if an explicit return occurs, the values that were transferred.

**Description:**

block establishes a block named *name* and then evaluates *forms* as an implicit progn.

The special operators block and return-from work together to provide a structured, lexical, non-local exit facility. At any point lexically contained within forms, return-from can be used with the given *name* to return control and values from the block form, except when an intervening block with the same name has been established, in which case the outer block is shadowed by the inner one.

The block named *name* has lexical scope and dynamic extent.

Once established, a block may only be exited once, whether by normal return or explicit return.

**Examples:**

## CLHS: Declaration DYNAMIC-EXTENT

```
(block empty) => NIL
(block whocares (values 1 2) (values 3 4)) => 3, 4
(let ((x 1))
  (block stop (setq x 2) (return-from stop) (setq x 3)))
  x) => 2
(block early (return-from early (values 1 2)) (values 3 4)) => 1, 2
(block outer (block inner (return-from outer 1)) 2) => 1
(block twin (block twin (return-from twin 1)) 2) => 2
;; Contrast behavior of this example with corresponding example of CATCH.
(block b
  (flet ((b1 () (return-from b 1)))
    (block b (b1) (print 'unreachable))
  2)) => 1
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[return](#), [return-from](#), [Section 3.1 \(Evaluation\)](#)

**Notes:**

## Special Operator CATCH

**Syntax:**

**catch** *tag form\** => *result\**

**Arguments and Values:**

*tag*—a [catch tag](#); evaluated.

*forms*—an [implicit progn](#).

*results*—if the *forms* exit normally, the [values](#) returned by the *forms*; if a throw occurs to the *tag*, the [values](#) that are thrown.

**Description:**

**catch** is used as the destination of a non-local control transfer by [throw](#). Tags are used to find the [catch](#) to which a [throw](#) is transferring control. (`(catch 'foo form)`) catches a (`(throw 'foo form)`) but not a (`(throw 'bar form)`).

The order of execution of [catch](#) follows:

1. Tag is evaluated. It serves as the name of the [catch](#).
2. Forms are then evaluated as an [implicit progn](#), and the results of the last form are returned unless a [throw](#) occurs.
3. If a [throw](#) occurs during the execution of one of the forms, control is transferred to the [catch form](#) whose tag is [eq](#) to the tag argument of the [throw](#) and which is the most recently established [catch](#) with that tag. No further evaluation of forms occurs.

## CLHS: Declaration DYNAMIC-EXTENT

4. The tag established by **catch** is disestablished just before the results are returned.

If during the execution of one of the *forms*, a **throw** is executed whose tag is **eq** to the **catch** tag, then the values specified by the **throw** are returned as the result of the dynamically most recently established **catch** form with that tag.

The mechanism for **catch** and **throw** works even if **throw** is not within the lexical scope of **catch**. **throw** must occur within the *dynamic extent* of the *evaluation* of the body of a **catch** with a corresponding *tag*.

### Examples:

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

;; Contrast behavior of this example with corresponding example of BLOCK.
(catch 'c
  (flet ((cl () (throw 'c 1)))
    (catch 'c (cl) (print 'unreachable))
  2)) => 2
```

**Affected By:** None.

### Exceptional Situations:

An error of **type control-error** is signaled if **throw** is done when there is no suitable **catch tag**.

### See Also:

**throw**, Section 3.1 (Evaluation)

### Notes:

It is customary for *symbols* to be used as *tags*, but any *object* is permitted. However, numbers should not be used because the comparison is done using **eq**.

**catch** differs from **block** in that **catch** tags have dynamic *scope* while **block** names have *lexical scope*.

## Symbol DECLARE

### Syntax:

**declare declaration-specifier\***

### Arguments:

*declaration-specifier*—a *declaration specifier*; not evaluated.

### Description:

A **declare expression**, sometimes called a *declaration*, can occur only at the beginning of the bodies of certain

## CLHS: Declaration DYNAMIC-EXTENT

*forms*; that is, it may be preceded only by other **declare expressions**, or by a **documentation string** if the context permits.

A **declare expression** can occur in a **lambda expression** or in any of the *forms* listed in the next figure.

<u>defgeneric</u>	<u>do-external-symbols</u>	<u>prog</u>
<u>define-compiler-macro</u>	<u>do-symbols</u>	<u>prog*</u>
<u>define-method-combination</u>	<u>dolist</u>	<u>restart-case</u>
<u>define-setf-expander</u>	<u>dotimes</u>	<u>symbol-macrolet</u>
<u>defmacro</u>	<u>flet</u>	<u>with-accessors</u>
<u>defmethod</u>	<u>handler-case</u>	<u>with-hash-table-iterator</u>
<u>defsetf</u>	<u>labels</u>	<u>with-input-from-string</u>
<u>deftype</u>	<u>let</u>	<u>with-open-file</u>
<u>defun</u>	<u>let*</u>	<u>with-open-stream</u>
<u>destructuring-bind</u>	<u>locally</u>	<u>with-output-to-string</u>
<u>do</u>	<u>macrolet</u>	<u>with-package-iterator</u>
<u>do*</u>	<u>multiple-value-bind</u>	<u>with-slots</u>
<u>do-all-symbols</u>	<u>pprint-logical-block</u>	

**Figure 3–23. Standardized Forms In Which Declarations Can Occur**

A **declare expression** can only occur where specified by the syntax of these *forms*. The consequences of attempting to evaluate a **declare expression** are undefined. In situations where such *expressions* can appear, explicit checks are made for their presence and they are never actually evaluated; it is for this reason that they are called "**declare expressions**" rather than "**declare forms**".

**Macro forms** cannot expand into declarations; **declare expressions** must appear as actual **subexpressions** of the *form* to which they refer.

The next figure shows a list of **declaration identifiers** that can be used with **declare**.

<u>dynamic-extent</u>	<u>ignore</u>	<u>optimize</u>
<u>ftype</u>	<u>inline</u>	<u>special</u>
<u>ignorable</u>	<u>notinline</u>	<u>type</u>

**Figure 3–24. Local Declaration Specifiers**

An implementation is free to support other (*implementation-defined*) **declaration identifiers** as well.

### Examples:

```
(defun nonsense (k x z)
  (foo z x)                                ;First call to foo
  (let ((j (foo k x)))                      ;Second call to foo
    (x (* k k)))
  (declare (inline foo) (special x z))
  (foo x j z)))                            ;Third call to foo
```

In this example, the **inline** declaration applies only to the third call to **foo**, but not to the first or second ones. The **special** declaration of **x** causes **let** to make a dynamic **binding** for **x**, and causes the reference to **x** in the body of **let** to be a dynamic reference. The reference to **x** in the second call to **foo** is a local reference to the second parameter of **nonsense**. The reference to **x** in the first call to **foo** is a local reference, not a **special** one. The **special** declaration of **z** causes the reference to **z** in the third call to **foo** to be a dynamic reference; it does not refer to the parameter to **nonsense** named **z**, because that parameter **binding** has not been

## CLHS: Declaration DYNAMIC-EXTENT

declared to be **special**. (The **special** declaration of *z* does not appear in the body of **defun**, but in an inner *form*, and therefore does not affect the *binding* of the *parameter*.)

**Affected By:** None.

**Exceptional Situations:**

The consequences of trying to use a **declare expression** as a *form* to be *evaluated* are undefined.

**See Also:**

**proclaim**, [Section 4.2.3 \(Type Specifiers\)](#), **declaration**, **dynamic-extent**, **ftype**, **ignorable**, **ignore**, **inline**, **notinline**, **optimize**, **type**

**Notes:** None.

## **Special Operator EVAL-WHEN**

**Syntax:**

**eval-when** (*situation\**) *form\** => *result\**

**Arguments and Values:**

*situation*—One of the *symbols* :compile-toplevel, :load-toplevel, :execute, **compile**, **load**, or **eval**.

The use of **eval**, **compile**, and **load** is deprecated.

*forms*—an *implicit progn*.

*results*—the *values* of the *forms* if they are executed, or **nil** if they are not.

**Description:**

The body of an **eval-when** form is processed as an *implicit progn*, but only in the *situations* listed.

The use of the *situations* :compile-toplevel (or **compile**) and :load-toplevel (or **load**) controls whether and when *evaluation* occurs when **eval-when** appears as a *top level form* in code processed by **compile-file**. See [Section 3.2.3 \(File Compilation\)](#).

The use of the *situation* :execute (or **eval**) controls whether evaluation occurs for other **eval-when forms**; that is, those that are not *top level forms*, or those in code processed by **eval** or **compile**. If the :execute situation is specified in such a *form*, then the body *forms* are processed as an *implicit progn*; otherwise, the **eval-when form** returns **nil**.

**eval-when** normally appears as a *top level form*, but it is meaningful for it to appear as a *non-top-level form*. However, the compile-time side effects described in [Section 3.2 \(Compilation\)](#) only take place when **eval-when** appears as a *top level form*.

**Examples:**

## CLHS: Declaration DYNAMIC-EXTENT

One example of the use of **eval-when** is that for the compiler to be able to read a file properly when it uses user-defined *reader macros*, it is necessary to write

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (set-macro-character #\$ #'(lambda (stream char)
    (declare (ignore char))
    (list 'dollar (read stream)))))) => T
```

This causes the call to **set-macro-character** to be executed in the compiler's execution environment, thereby modifying its reader syntax table.

```
; ; The EVAL-WHEN in this case is not at toplevel, so only the :EXECUTE
; ; keyword is considered. At compile time, this has no effect.
; ; At load time (if the LET is at toplevel), or at execution time
; ; (if the LET is embedded in some other form which does not execute
; ; until later) this sets (SYMBOL-FUNCTION 'FOO1) to a function which
; ; returns 1.
(let ((x 1))
  (eval-when (:execute :load-toplevel :compile-toplevel)
    (setf (symbol-function 'foo1) #'(lambda () x)))

; ; If this expression occurs at the toplevel of a file to be compiled,
; ; it has BOTH a compile time AND a load-time effect of setting
; ; (SYMBOL-FUNCTION 'FOO2) to a function which returns 2.
(eval-when (:execute :load-toplevel :compile-toplevel)
  (let ((x 2))
    (eval-when (:execute :load-toplevel :compile-toplevel)
      (setf (symbol-function 'foo2) #'(lambda () x)))

; ; If this expression occurs at the toplevel of a file to be compiled,
; ; it has BOTH a compile time AND a load-time effect of setting the
; ; function cell of FOO3 to a function which returns 3.
(eval-when (:execute :load-toplevel :compile-toplevel)
  (setf (symbol-function 'foo3) #'(lambda () 3)))

; ; #4: This always does nothing. It simply returns NIL.
(eval-when (:compile-toplevel)
  (eval-when (:compile-toplevel)
    (print 'foo4)))

; ; If this form occurs at toplevel of a file to be compiled, FOO5 is
; ; printed at compile time. If this form occurs in a non-top-level
; ; position, nothing is printed at compile time. Regardless of context,
; ; nothing is ever printed at load time or execution time.
(eval-when (:compile-toplevel)
  (eval-when (:execute)
    (print 'foo5)))

; ; If this form occurs at toplevel of a file to be compiled, FOO6 is
; ; printed at compile time. If this form occurs in a non-top-level
; ; position, nothing is printed at compile time. Regardless of context,
; ; nothing is ever printed at load time or execution time.
(eval-when (:execute :load-toplevel)
  (eval-when (:compile-toplevel)
    (print 'foo6)))
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[compile-file](#), [Section 3.2 \(Compilation\)](#)

**Notes:**

The following effects are logical consequences of the definition of [eval-when](#):

- \* *Execution of a single [eval-when](#) expression executes the body code at most once.*
- \* *Macros intended for use in [top level forms](#) should be written so that side-effects are done by the [forms](#) in the macro expansion. The macro-expander itself should not do the side-effects.*

For example:

Wrong:

```
(defmacro foo ()
  (really-foo)
  `(really-foo))
```

Right:

```
(defmacro foo ()
  `(~(eval-when (:compile-toplevel :execute :load-toplevel) (really-foo))))
```

Adherence to this convention means that such [macros](#) behave intuitively when appearing as [non-top-level forms](#).

- \* *Placing a variable binding around an [eval-when](#) reliably captures the binding because the compile-time-too mode cannot occur (i.e., introducing a variable binding means that the [eval-when](#) is not a top level form). For example,*

```
(let ((x 3))
  (eval-when (:execute :load-toplevel :compile-toplevel) (print x)))
```

prints 3 at execution (i.e., load) time, and does not print anything at compile time. This is important so that expansions of [defun](#) and [defmacro](#) can be done in terms of [eval-when](#) and can correctly capture the [lexical environment](#).

```
(defun bar (x) (defun foo () (+ x 3)))
```

might expand into

```
(defun bar (x)
  (progn (eval-when (:compile-toplevel)
    (compiler::notice-function-definition 'foo '(x)))
    (eval-when (:execute :load-toplevel)
      (setf (symbol-function 'foo) #'(lambda () (+ x 3)))))
```

which would be treated by the above rules the same as

```
(defun bar (x)
  (setf (symbol-function 'foo) #'(lambda () (+ x 3))))
```

when the definition of [bar](#) is not a [top level form](#).

***Special Operator FLET, LABELS, MACROLET*****Syntax:**

**flet** ((*function-name lambda-list* [[*local-declaration\** / *local-documentation*]] *local-form\**)\*)  
*declaration\** *form\**

=> *result\**

**labels** ((*function-name lambda-list* [[*local-declaration\** / *local-documentation*]] *local-form\**)\*)  
*declaration\** *form\**

=> *result\**

**macrolet** ((*name lambda-list* [[*local-declaration\** / *local-documentation*]] *local-form\**)\*) *declaration\**  
*form\**

=> *result\**

**Arguments and Values:**

*function-name*—a function name.

*name*—a symbol.

*lambda-list*—a lambda list; for **flet** and **labels**, it is an ordinary lambda list; for **macrolet**, it is a macro lambda list.

*local-declaration*—a declare expression; not evaluated.

*declaration*—a declare expression; not evaluated.

*local-documentation*—a string; not evaluated.

*local-forms, forms*—an implicit progn.

*results*—the values of the *forms*.

**Description:**

**flet**, **labels**, and **macrolet** define local functions and macros, and execute *forms* using the local definitions. *Forms* are executed in order of occurrence.

The body forms (but not the lambda list) of each function created by **flet** and **labels** and each macro created by **macrolet** are enclosed in an implicit block whose name is the function block name of the *function-name* or *name*, as appropriate.

The scope of the *declarations* between the list of local function/macro definitions and the body *forms* in **flet** and **labels** does not include the bodies of the locally defined functions, except that for **labels**, any inline, notinline, or ftype declarations that refer to the locally defined functions do apply to the local function bodies. That is, their scope is the same as the function name that they affect. The scope of these *declarations*

does not include the bodies of the macro expander functions defined by **macrolet**.

**flet**

**flet** defines locally named *functions* and executes a series of *forms* with these definition *bindings*. Any number of such local *functions* can be defined.

The *scope* of the name *binding* encompasses only the body. Within the body of **flet**, *function-names* matching those defined by **flet** refer to the locally defined *functions* rather than to the global function definitions of the same name. Also, within the scope of **flet**, global *setf expander* definitions of the *function-name* defined by **flet** do not apply. Note that this applies to (defsetf *f* . . .), not (defmethod (setf *f*) . . .).

The names of *functions* defined by **flet** are in the *lexical environment*; they retain their local definitions only within the body of **flet**. The function definition bindings are visible only in the body of **flet**, not the definitions themselves. Within the function definitions, local function names that match those being defined refer to *functions* or *macros* defined outside the **flet**. **flet** can locally *shadow* a global function name, and the new definition can refer to the global definition.

Any *local-documentation* is attached to the corresponding local *function* (if one is actually created) as a *documentation string*.

**labels**

**labels** is equivalent to **flet** except that the scope of the defined function names for **labels** encompasses the function definitions themselves as well as the body.

**macrolet**

**macrolet** establishes local *macro* definitions, using the same format used by **defmacro**.

Within the body of **macrolet**, global *setf expander* definitions of the *names* defined by the **macrolet** do not apply; rather, *setf* expands the *macro form* and recursively process the resulting *form*.

The macro-expansion functions defined by **macrolet** are defined in the *lexical environment* in which the **macrolet** form appears. Declarations and **macrolet** and **symbol-macrolet** definitions affect the local macro definitions in a **macrolet**, but the consequences are undefined if the local macro definitions reference any local *variable* or *function bindings* that are visible in that *lexical environment*.

Any *local-documentation* is attached to the corresponding local *macro function* as a *documentation string*.

**Examples:**

```
(defun foo (x flag)
  (macrolet ((fudge (z)
                  ;The parameters x and flag are not accessible
                  ;at this point; a reference to flag would be to
                  ;the global variable of that name.
                  ` (if flag (* ,z ,z) ,z)))
    ;The parameters x and flag are accessible here.
    (+ x
       (fudge x)
       (fudge (+ x 1)))))

==

(defun foo (x flag)
  (+ x
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(if flag (* x x) x)
(if flag (* (+ x 1) (+ x 1)))
```

after macro expansion. The occurrences of `x` and `flag` legitimately refer to the parameters of the function `foo` because those parameters are visible at the site of the macro call which produced the expansion.

```
(flet ((flet1 (n) (+ n n)))
  (flet ((flet1 (n) (+ 2 (flet1 n))))
    (flet1 2))) => 6

(defun dummy-function () 'top-level) => DUMMY-FUNCTION
(funcall #'dummy-function) => TOP-LEVEL
(flet ((dummy-function () 'shadow))
  (funcall #'dummy-function)) => SHADOW
(eq (funcall #'dummy-function) (funcall 'dummy-function))
=> true
(flet ((dummy-function () 'shadow))
  (eq (funcall #'dummy-function)
       (funcall 'dummy-function)))
=> false

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

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

(flet ((safesqrt (x) (sqrt (abs x))))
  ; The safesqrt function is used in two places.
  (safesqrt (apply #'+ (map 'list #'safesqrt '(1 2 3 4 5 6)))))
=> 3.291173

(defun integer-power (n k)
  (declare (integer n))
  (declare (type (integer 0 *) k))
  (labels ((expt0 (x k a)
              (declare (integer x a) (type (integer 0 *) k))
              (cond ((zerop k) a)
                    ((evenp k) (expt1 (* x x) (floor k 2) a))
                    (t (expt0 (* x x) (floor k 2) (* x a)))))

            (expt1 (x k a)
              (declare (integer x a) (type (integer 0 *) k))
              (cond ((evenp k) (expt1 (* x x) (floor k 2) a))
                    (t (expt0 (* x x) (floor k 2) (* x a))))))

            (expt0 n k 1))) => INTEGER-POWER

(defun example (y l)
  (flet ((attach (x)
            (setq l (append l (list x)))))
    (declare (inline attach))
    (dolist (x y)
      (unless (null (cdr x))
        (attach x)))
    l))
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(example '((a apple apricot) (b banana) (c cherry) (d) (e))
          '((1) (2) (3) (4 2) (5) (6 3 2)))
=> ((1) (2) (3) (4 2) (5) (6 3 2) (A APPLE APRICOT) (B BANANA) (C CHERRY))
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[declare](#), [defmacro](#), [defun](#), [documentation](#), [let](#), [Section 3.1 \(Evaluation\)](#), [Section 3.4.11 \(Syntactic Interaction of Documentation Strings and Declarations\)](#)

**Notes:**

It is not possible to define recursive *functions* with [flet](#). [labels](#) can be used to define mutually recursive *functions*.

If a [macrolet form](#) is a *top level form*, the body *forms* are also processed as *top level forms*. See [Section 3.2.3 \(File Compilation\)](#).

## Special Operator FUNCTION

**Syntax:**

**function** *name* => *function*

**Arguments and Values:**

*name*—a [function name](#) or [lambda expression](#).

*function*—a [function object](#).

**Description:**

The *value* of [function](#) is the [functional value](#) of *name* in the current [lexical environment](#).

If *name* is a [function name](#), the functional definition of that name is that established by the innermost lexically enclosing [flet](#), [labels](#), or [macrolet form](#), if there is one. Otherwise the global functional definition of the [function name](#) is returned.

If *name* is a [lambda expression](#), then a [lexical closure](#) is returned. In situations where a [closure](#) over the same set of [bindings](#) might be produced more than once, the various resulting [closures](#) might or might not be [eq](#).

It is an error to use [function](#) on a [function name](#) that does not denote a [function](#) in the lexical environment in which the [function](#) form appears. Specifically, it is an error to use [function](#) on a [symbol](#) that denotes a [macro](#) or [special form](#). An implementation may choose not to signal this error for performance reasons, but implementations are forbidden from defining the failure to signal an error as a useful behavior.

**Examples:**

## CLHS: Declaration DYNAMIC-EXTENT

```
(defun adder (x) (function (lambda (y) (+ x y))))
```

The result of (`adder 3`) is a function that adds 3 to its argument:

```
(setq add3 (adder 3))
(funcall add3 5) => 8
```

This works because **function** creates a *closure* of the *lambda expression* that is able to refer to the *value* 3 of the variable `x` even after control has returned from the function `adder`.

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[\*\*defun\*\*](#), [\*\*fdefinition\*\*](#), [\*\*flet\*\*](#), [\*\*labels\*\*](#), [\*\*symbol-function\*\*](#), [Section 3.1.2.1.1 \(Symbols as Forms\)](#), [Section 2.4.8.2 \(Sharpsign Single-Quote\)](#), [Section 22.1.3.13 \(Printing Other Objects\)](#)

**Notes:**

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

## Special Operator GO

**Syntax:**

```
go tag =>
```

**Arguments and Values:**

`tag`—[a go tag](#).

**Description:**

**go** transfers control to the point in the body of an enclosing **tagbody** form labeled by a tag **eql** to `tag`. If there is no such `tag` in the body, the bodies of lexically containing **tagbody forms** (if any) are examined as well. If several tags are **eql** to `tag`, control is transferred to whichever matching `tag` is contained in the innermost **tagbody** form that contains the **go**. The consequences are undefined if there is no matching `tag` lexically visible to the point of the **go**.

The transfer of control initiated by **go** is performed as described in [Section 5.2 \(Transfer of Control to an Exit Point\)](#).

**Examples:**

```
(tagbody
  (setq val 2)
  (go lp)
  (incf val 3))
```

## CLHS: Declaration DYNAMIC-EXTENT

```
lp (incf val 4)) => NIL
val => 6
```

The following is in error because there is a normal exit of the tagbody before the go is executed.

```
(let ((a nil))
  (tagbody t (setq a #'(lambda () (go t))))
  (funcall a))
```

The following is in error because the tagbody is passed over before the go form is executed.

```
(funcall (block nil
            (tagbody a (return #'(lambda () (go a))))))
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**tagbody**

**Notes:** None.

## **Special Operator IF**

**Syntax:**

**if** *test-form* *then-form* [*else-form*] => *result*\*

**Arguments and Values:**

*Test-form*—a form.

*Then-form*—a form.

*Else-form*—a form. The default is nil.

*results*—if the test-form yielded true, the values returned by the then-form; otherwise, the values returned by the else-form.

**Description:**

**if** allows the execution of a form to be dependent on a single test-form.

First *test-form* is evaluated. If the result is true, then *then-form* is selected; otherwise *else-form* is selected. Whichever form is selected is then evaluated.

**Examples:**

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

```
(defun test ()
  (dolist (truth-value '(t nil 1 (a b c)))
    (if truth-value (print 'true) (print 'false))
    (prin1 truth-value))) => TEST
(test)
>> TRUE T
>> FALSE NIL
>> TRUE 1
>> TRUE (A B C)
=> NIL
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[cond](#), [unless](#), [when](#)

**Notes:**

```
(if test-form then-form else-form)
== (cond (test-form then-form) (t else-form))
```

## Symbol LAMBDA

**Syntax:**

**lambda** *lambda-list* [[*declaration\** / *documentation*]] *form\**

**Arguments:**

*lambda-list*—an [ordinary lambda list](#).

*declaration*—a [declare expression](#); not evaluated.

*documentation*—a [string](#); not evaluated.

*form*—a [form](#).

**Description:**

A [lambda expression](#) is a [list](#) that can be used in place of a [function name](#) in certain contexts to denote a [function](#) by directly describing its behavior rather than indirectly by referring to the name of an [established function](#).

*Documentation* is attached to the denoted *function* (if any is actually created) as a [documentation string](#).

**See Also:**

[function](#), [documentation](#), [Section 3.1.3 \(Lambda Expressions\)](#), [Section 3.1.2.1.2.4 \(Lambda Forms\)](#), [Section 3.4.11 \(Syntactic Interaction of Documentation Strings and Declarations\)](#)

**Notes:**

The *lambda form*

```
((lambda lambda-list . body) . arguments)
```

is semantically equivalent to the *function form*

```
(funcall #'(lambda lambda-list . body) . arguments)
```

## **Special Operator LOAD-TIME-VALUE**

**Syntax:**

**load-time-value** *form* &optional *read-only-p* => *object*

**Arguments and Values:**

*form*—a *form*; evaluated as described below.

*read-only-p*—a *boolean*; not evaluated.

*object*—the *primary value* resulting from evaluating *form*.

**Description:**

**load-time-value** provides a mechanism for delaying evaluation of *form* until the expression is in the run-time environment; see [Section 3.2 \(Compilation\)](#).

*Read-only-p* designates whether the result can be considered a *constant object*. If **t**, the result is a read-only quantity that can, if appropriate to the *implementation*, be copied into read-only space and/or *coalesced* with *similar constant objects* from other *programs*. If **nil** (the default), the result must be neither copied nor coalesced; it must be considered to be potentially modifiable data.

If a **load-time-value** expression is processed by *compile-file*, the compiler performs its normal semantic processing (such as macro expansion and translation into machine code) on *form*, but arranges for the execution of *form* to occur at load time in a *null lexical environment*, with the result of this *evaluation* then being treated as a *literal object* at run time. It is guaranteed that the evaluation of *form* will take place only once when the *file* is *loaded*, but the order of evaluation with respect to the evaluation of *top level forms* in the file is *implementation-dependent*.

If a **load-time-value** expression appears within a function compiled with *compile*, the *form* is evaluated at compile time in a *null lexical environment*. The result of this compile-time evaluation is treated as a *literal object* in the compiled code.

If a **load-time-value** expression is processed by *eval*, *form* is evaluated in a *null lexical environment*, and one value is returned. Implementations that implicitly compile (or partially compile) expressions processed by *eval* might evaluate *form* only once, at the time this compilation is performed.

If the *same list* (**load-time-value form**) is evaluated or compiled more than once, it is *implementation-dependent* whether *form* is evaluated only once or is evaluated more than once. This can

## CLHS: Declaration DYNAMIC-EXTENT

happen both when an expression being evaluated or compiled shares substructure, and when the *same form* is processed by `eval` or `compile` multiple times. Since a `load-time-value` expression can be referenced in more than one place and can be evaluated multiple times by `eval`, it is *implementation-dependent* whether each execution returns a fresh *object* or returns the same *object* as some other execution. Users must use caution when destructively modifying the resulting *object*.

If two lists (`load-time-value form`) that are the *same* under `equal` but are not *identical* are evaluated or compiled, their values always come from distinct evaluations of `form`. Their *values* may not be coalesced unless `read-only-p` is `t`.

### Examples:

```
;;; The function INCR1 always returns the same value, even in different images.  
;;; The function INCR2 always returns the same value in a given image,  
;;; but the value it returns might vary from image to image.  
(defun incr1 (x) (+ x #.(random 17)))  
(defun incr2 (x) (+ x (load-time-value (random 17))))  
  
;;; The function FOO1-REF references the nth element of the first of  
;;; the *FOO-ARRAYS* that is available at load time. It is permissible for  
;;; that array to be modified (e.g., by SET-FOO1-REF); FOO1-REF will see the  
;;; updated values.  
(defvar *foo-arrays* (list (make-array 7) (make-array 8)))  
(defun fool-ref (n) (aref (load-time-value (first *my-arrays*) nil) n))  
(defun set-fool-ref (n val)  
  (setf (aref (load-time-value (first *my-arrays*) nil) n) val))  
  
;;; The function BAR1-REF references the nth element of the first of  
;;; the *BAR-ARRAYS* that is available at load time. The programmer has  
;;; promised that the array will be treated as read-only, so the system  
;;; can copy or coalesce the array.  
(defvar *bar-arrays* (list (make-array 7) (make-array 8)))  
(defun bar1-ref (n) (aref (load-time-value (first *my-arrays*) t) n))  
  
;;; This use of LOAD-TIME-VALUE permits the indicated vector to be coalesced  
;;; even though NIL was specified, because the object was already read-only  
;;; when it was written as a literal vector rather than created by a constructor.  
;;; User programs must treat the vector v as read-only.  
(defun baz-ref (n)  
  (let ((v (load-time-value #(A B C) nil)))  
    (values (svref v n) v)))  
  
;;; This use of LOAD-TIME-VALUE permits the indicated vector to be coalesced  
;;; even though NIL was specified in the outer situation because T was specified  
;;; in the inner situation. User programs must treat the vector v as read-only.  
(defun baz-ref (n)  
  (let ((v (load-time-value (load-time-value (vector 1 2 3) t) nil)))  
    (values (svref v n) v)))
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[compile-file](#), [compile](#), [eval](#), [Section 3.2.2.2 \(Minimal Compilation\)](#), [Section 3.2 \(Compilation\)](#)

**Notes:**

**load-time-value** must appear outside of quoted structure in a "for *evaluation*" position. In situations which would appear to call for use of **load-time-value** within a quoted structure, the **backquote reader macro** is probably called for; see [Section 2.4.6 \(Backquote\)](#).

Specifying **nil** for *read-only-p* is not a way to force an object to become modifiable if it has already been made read-only. It is only a way to say that, for an object that is modifiable, this operation is not intended to make that object read-only.

## **Special Operator LET, LET\***

**Syntax:**

**let** ({*var* / (*var* [*init-form*])}\*) *declaration\** *form\** => *result\**

**let\*** ({*var* / (*var* [*init-form*])}\*) *declaration\** *form\** => *result\**

**Arguments and Values:**

*var*—a symbol.

*init-form*—a form.

*declaration*—a declare expression; not evaluated.

*form*—a form.

*results*—the values returned by the forms.

**Description:**

**let** and **let\*** create new variable bindings and execute a series of forms that use these bindings. **let** performs the bindings in parallel and **let\*** does them sequentially.

The form

```
(let ((var1 init-form-1)
      (var2 init-form-2)
      ...
      (varm init-form-m))
  declaration1
  declaration2
  ...
  declarationp
  form1
  form2
  ...
  formn)
```

first evaluates the expressions *init-form-1*, *init-form-2*, and so on, in that order, saving the resulting values. Then all of the variables *varj* are bound to the corresponding values; each binding is lexical unless there is a

## CLHS: Declaration DYNAMIC-EXTENT

**special** declaration to the contrary. The expressions *form<sub>k</sub>* are then evaluated in order; the values of all but the last are discarded (that is, the body of a **let** is an *implicit progn*).

**let\*** is similar to **let**, but the *bindings* of variables are performed sequentially rather than in parallel. The expression for the *init-form* of a *var* can refer to *vars* previously bound in the **let\***.

The form

```
(let* ((var1 init-form-1)
      (var2 init-form-2)
      ...
      (varm init-form-m))
  declaration1
  declaration2
  ...
  declarationp
  form1
  form2
  ...
  formn)
```

first evaluates the expression *init-form-1*, then binds the variable *var1* to that value; then it evaluates *init-form-2* and binds *var2*, and so on. The expressions *form<sub>j</sub>* are then evaluated in order; the values of all but the last are discarded (that is, the body of **let\*** is an implicit **progn**).

For both **let** and **let\***, if there is not an *init-form* associated with a *var*, *var* is initialized to **nil**.

The special form **let** has the property that the *scope* of the name binding does not include any initial value form. For **let\***, a variable's *scope* also includes the remaining initial value forms for subsequent variable bindings.

### Examples:

```
(setq a 'top) => TOP
(defun dummy-function () a) => DUMMY-FUNCTION
(let ((a 'inside) (b a))
  (format nil "~S ~S ~S" a b (dummy-function))) => "INSIDE TOP TOP"
(let* ((a 'inside) (b a))
  (format nil "~S ~S ~S" a b (dummy-function))) => "INSIDE INSIDE TOP"
(let ((a 'inside) (b a))
  (declare (special a))
  (format nil "~S ~S ~S" a b (dummy-function))) => "INSIDE TOP INSIDE"
```

The code

```
(let (x)
  (declare (integer x))
  (setq x (gcd y z))
  ...)
```

is incorrect; although *x* is indeed set before it is used, and is set to a value of the declared type *integer*, nevertheless *x* initially takes on the value **nil** in violation of the type declaration.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

### progv

**Notes:** None.

## ***Special Operator LOCALLY***

**Syntax:**

**locally** *declaration\** *form\**  $\Rightarrow$  *result\**

**Arguments and Values:**

*Declaration*—a declare expression; not evaluated.

*forms*—an implicit progn.

*results*—the values of the *forms*.

**Description:**

Sequentially evaluates a body of *forms* in a lexical environment where the given *declarations* have effect.

**Examples:**

```
(defun sample-function (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))
        ;; this next y is regarded as special
        y))))
=> SAMPLE-FUNCTION
(sample-function nil) => (T NIL)
(setq x '(1 2 3) y '(4 . 5)) => (4 . 5)

;; The following declarations are not notably useful in specific.
;; They just offer a sample of valid declaration syntax using LOCALLY.
(locally (declare (inline floor) (notinline car cdr))
  (declare (optimize space))
  (floor (car x) (cdr y))) => 0, 1

;; This example shows a definition of a function that has a particular set
;; of OPTIMIZE settings made locally to that definition.
(locally (declare (optimize (safety 3) (space 3) (speed 0)))
  (defun frob (w x y &optional (z (foo x y)))
    (mumble x y z w)))
=> FROB

;; This is like the previous example, except that the optimize settings
;; remain in effect for subsequent definitions in the same compilation unit.
(declaim (optimize (safety 3) (space 3) (speed 0)))
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(defun frob (w x y &optional (z (foo x y)))
  (mumble x y z w))
=> FROB
```

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**declare**

**Notes:**

The **special** declaration may be used with **locally** to affect references to, rather than *bindings* of, *variables*.

If a **locally form** is a **top level form**, the body *forms* are also processed as **top level forms**. See [Section 3.2.3 \(File Compilation\)](#).

## **Special Operator MULTIPLE-VALUE-PROG1**

**Syntax:**

**multiple-value-prog1** *first-form* *form\** => *first-form-results*

**Arguments and Values:**

*first-form*—*a form*; evaluated as described below.

*form*—*a form*; evaluated as described below.

*first-form-results*—*the values* resulting from the *evaluation* of *first-form*.

**Description:**

**multiple-value-prog1** evaluates *first-form* and saves all the values produced by that *form*. It then evaluates each *form* from left to right, discarding their values.

**Examples:**

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

**Side Effects:** None.

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**prog1**

**Notes:** None.

## ***Special Operator MULTIPLE-VALUE-CALL***

**Syntax:**

**multiple-value-call** *function-form* *form\**  $\Rightarrow$  *result\**

**Arguments and Values:**

*function-form*—a form; evaluated to produce *function*.

*function*—a function designator resulting from the evaluation of *function-form*.

*form*—a form.

*results*—the values returned by the *function*.

**Description:**

Applies *function* to a list of the objects collected from groups of multiple values[2].

**multiple-value-call** first evaluates the *function-form* to obtain *function*, and then evaluates each *form*. All the values of each *form* are gathered together (not just one value from each) and given as arguments to the *function*.

**Examples:**

```
(multiple-value-call #'list 1 '/ (values 2 3) '/ (values) '/ (floor 2.5))
=> (1 / 2 3 / / 2 0.5)
(+ (floor 5 3) (floor 19 4)) == (+ 1 4)
=> 5
(multiple-value-call #'+ (floor 5 3) (floor 19 4)) == (+ 1 2 4 3)
=> 10
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**multiple-value-list, multiple-value-bind**

**Notes:** None.

## **Special Operator PROGN**

### Syntax:

**progn** *form\**  $\Rightarrow$  *result\**

### Arguments and Values:

*forms*—an implicit progn.

*results*—the values of the forms.

### Description:

**progn** evaluates *forms*, in the order in which they are given.

The values of each *form* but the last are discarded.

If **progn** appears as a top level form, then all forms within that **progn** are considered by the compiler to be top level forms.

### 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) 'here)
    (progn (setq a t) 'there)) => HERE
a => NIL
```

**Affected By:** None.

**Exceptional Situations:** None.

### See Also:

[prog1](#), [prog2](#), [Section 3.1 \(Evaluation\)](#)

### Notes:

Many places in Common Lisp involve syntax that uses implicit progns. That is, part of their syntax allows many forms to be written that are to be evaluated sequentially, discarding the results of all forms but the last and returning the results of the last form. Such places include, but are not limited to, the following: the body of a lambda expression; the bodies of various control and conditional forms (e.g., case, catch, progn, and when).

## **Special Operator PROGV**

### Syntax:

**progv** *symbols values form\** => *result\**

#### Arguments and Values:

*symbols*—a *list* of *symbols*; evaluated.

*values*—a *list* of *objects*; evaluated.

*forms*—an *implicit progn*.

*results*—the *values* returned by the *forms*.

#### Description:

**progv** creates new dynamic variable *bindings* and executes each *form* using those *bindings*. Each *form* is evaluated in order.

**progv** allows *binding* one or more dynamic variables whose names may be determined at run time. Each *form* is evaluated in order with the dynamic variables whose names are in *symbols* bound to corresponding *values*. If too few *values* are supplied, the remaining *symbols* are bound and then made to have no value. If too many *values* are supplied, the excess values are ignored. The *bindings* of the dynamic variables are undone on exit from **progv**.

#### Examples:

```
(setq *x* 1) => 1
(progv '(*x*) '(2) *x*) => 2
*x* => 1
```

Assuming *\*x\** is not globally special,

```
(let ((*x* 3))
  (progv '(*x*) '(4)
    (list *x* (symbol-value '*x*)))) => (3 4)
```

**Affected By:** None.

**Exceptional Situations:** None.

#### See Also:

[let](#), [Section 3.1 \(Evaluation\)](#)

#### Notes:

Among other things, **progv** is useful when writing interpreters for languages embedded in Lisp; it provides a handle on the mechanism for *binding dynamic variables*.

## Special Operator QUOTE

#### Syntax:

**quote** *object* => *object*

**Arguments and Values:**

*object*—an object; not evaluated.

**Description:**

The quote special operator just returns *object*.

The consequences are undefined if literal objects (including quoted objects) are destructively modified.

**Examples:**

```
(setq a 1) => 1
(quote (setq a 3)) => (SETQ A 3)
a => 1
'a => A
"a => (QUOTE A)
'"a => (QUOTE (QUOTE A))
(setq a 43) => 43
(list a (cons a 3)) => (43 (43 . 3))
(list (quote a) (quote (cons a 3))) => (A (CONS A 3))
1 => 1
'1 => 1
"foo" => "foo"
'"foo" => "foo"
(car '(a b)) => A
'(car '(a b)) => (CAR (QUOTE (A B)))
#(car '(a b)) => #(CAR (QUOTE (A B)))
'#{car '(a b)} => #(CAR (QUOTE (A B)))
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**Notes:**

The textual notation '*object*' is equivalent to (quote *object*); see [Section 3.2.1 \(Compiler Terminology\)](#).

Some objects, called self-evaluating objects, do not require quotation by quote. However, symbols and lists are used to represent parts of programs, and so would not be useable as constant data in a program without quote. Since quote suppresses the evaluation of these objects, they become data rather than program.

## Special Operator RETURN-FROM

**Syntax:**

**return-from** *name* [*result*] =>

**Arguments and Values:**

*name*—a block tag; not evaluated.

*result*—a form; evaluated. The default is nil.

### Description:

Returns control and multiple values[2] from a lexically enclosing block.

A block form named *name* must lexically enclose the occurrence of return-from; any values yielded by the evaluation of *result* are immediately returned from the innermost such lexically enclosing block.

The transfer of control initiated by return-from is performed as described in Section 5.2 (Transfer of Control to an Exit Point).

### Examples:

```
(block alpha (return-from alpha 1) => NIL
(block alpha (return-from alpha 1) 2) => 1
(block alpha (return-from alpha (values 1 2)) 3) => 1, 2
(let ((a 0))
  (dotimes (i 10) (incf a) (when (oddp i) (return)))
  a) => 2
(defun temp (x)
  (if x (return-from temp 'dummy)
  44) => TEMP
(temp nil) => 44
(temp t) => DUMMY
(block out
  (flet ((exit (n) (return-from out n)))
    (block out (exit 1)))
  2) => 1
(block nil
  (unwind-protect (return-from nil 1)
    (return-from nil 2)))
=> 2
(dolist (flag '(nil t))
  (block nil
    (let ((x 5))
      (declare (special x))
      (unwind-protect (return-from nil)
        (print x))))
    (print 'here)))
>> 5
>> HERE
>> 5
>> HERE
=> NIL
(dolist (flag '(nil t))
  (block nil
    (let ((x 5))
      (declare (special x))
      (unwind-protect
        (if flag (return-from nil))
        (print x))))
    (print 'here)))
>> 5
>> HERE
>> 5
```

```
>> HERE
=> NIL
```

The following has undefined consequences because the block form exits normally before the return-from form is attempted.

```
(funcall (block nil #'(lambda () (return-from nil)))) is an error.
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[block](#), [return](#), [Section 3.1 \(Evaluation\)](#)

**Notes:** None.

## **Special Form SETQ**

**Syntax:**

**setq** {pair}\* => result

pair ::= var form

**Pronunciation:**

[set,kyoo]

**Arguments and Values:**

*var*—a symbol naming a variable other than a constant variable.

*form*—a form.

*result*—the primary value of the last *form*, or nil if no *pairs* were supplied.

**Description:**

Assigns values to variables.

(**setq** *var1 form1 var2 form2 . . .*) is the simple variable assignment statement of Lisp. First *form1* is evaluated and the result is stored in the variable *var1*, then *form2* is evaluated and the result stored in *var2*, and so forth. **setq** may be used for assignment of both lexical and dynamic variables.

If any *var* refers to a binding made by **symbol-macrolet**, then that *var* is treated as if **setf** (not **setq**) had been used.

**Examples:**

## CLHS: Declaration DYNAMIC-EXTENT

```
; ; A simple use of SETQ to establish values for variables.  
(setq a 1 b 2 c 3) => 3  
a => 1  
b => 2  
c => 3  
  
; ; Use of SETQ to update values by sequential assignment.  
(setq a (1+ b) b (1+ a) c (+ a b)) => 7  
a => 3  
b => 4  
c => 7  
  
; ; This illustrates the use of SETQ on a symbol macro.  
(let ((x (list 10 20 30)))  
  (symbol-macrolet ((y (car x)) (z (cadr x)))  
    (setq y (1+ z) z (1+ y))  
    (list x y z)))  
=> ((21 22 30) 21 22)
```

### Side Effects:

The primary value of each *form* is assigned to the corresponding *var*.

**Affected By:** None.

**Exceptional Situations:** None.

### See Also:

psetq, set, setf

**Notes:** None.

## Special Operator SYMBOL-MACROLET

### Syntax:

**symbol-macrolet** ((*symbol expansion*)\*) *declaration\** *form\**\*

=> *result\**\*

### Arguments and Values:

*symbol*---a symbol.

*expansion*---a form.

*declaration*---a declare expression; not evaluated.

*forms*---an implicit progn.

*results*---the values returned by the *forms*.

### Description:

Special Operator SYMBOL-MACROLET

**symbol-macrolet** provides a mechanism for affecting the *macro expansion* environment for *symbols*.

**symbol-macrolet** lexically establishes expansion functions for each of the *symbol macros* named by *symbols*. The only guaranteed property of an expansion *function* for a *symbol macro* is that when it is applied to the *form* and the *environment* it returns the correct expansion. (In particular, it is *implementation-dependent* whether the expansion is conceptually stored in the expansion function, the *environment*, or both.)

Each reference to *symbol* as a variable within the lexical *scope* of **symbol-macrolet** is expanded by the normal macro expansion process; see [Section 3.1.2.1.1 \(Symbols as Forms\)](#). The expansion of a symbol macro is subject to further macro expansion in the same lexical environment as the symbol macro invocation, exactly analogous to normal *macros*.

Exactly the same *declarations* are allowed as for **let** with one exception: **symbol-macrolet** signals an error if a **special** declaration names one of the *symbols* being defined by **symbol-macrolet**.

When the *forms* of the **symbol-macrolet** form are expanded, any use of **setq** to set the value of one of the specified variables is treated as if it were a **setf**. **psetq** of a *symbol* defined as a symbol macro is treated as if it were a **psetf**, and **multiple-value-setq** is treated as if it were a **setf** of *values*.

The use of **symbol-macrolet** can be shadowed by **let**. In other words, **symbol-macrolet** only substitutes for occurrences of *symbol* that would be in the *scope* of a lexical binding of *symbol* surrounding the *forms*.

### Examples:

```
;;; The following is equivalent to
;;;   (list 'foo (let ((x 'bar)) x)),
;;; not
;;;   (list 'foo (let (('foo 'bar)) 'foo))
(symbol-macrolet ((x 'foo))
  (list x (let ((x 'bar)) x)))
=> (foo bar)
NOT=> (foo foo)

(symbol-macrolet ((x '(%(foo x))))
  (list x))
=> ((FOO X))
```

**Affected By:** None.

### Exceptional Situations:

If an attempt is made to bind a *symbol* that is defined as a *global variable*, an error of *type program-error* is signaled.

If *declaration* contains a **special** declaration that names one of the *symbols* being bound by **symbol-macrolet**, an error of *type program-error* is signaled.

### See Also:

[with-slots, macroexpand](#)

### Notes:

The special form **symbol-macrolet** is the basic mechanism that is used to implement **with-slots**.

If a **symbol-macrolet form** is a **top level form**, the **forms** are also processed as **top level forms**. See [Section 3.2.3 \(File Compilation\)](#).

## Special Operator TAGBODY

### Syntax:

**tagbody** {tag / statement}\* => **nil**

### Arguments and Values:

*tag*—a **go tag**; not evaluated.

*statement*—a **compound form**; evaluated as described below.

### Description:

Executes zero or more *statements* in a **lexical environment** that provides for control transfers to labels indicated by the *tags*.

The *statements* in a **tagbody** are *evaluated* in order from left to right, and their *values* are discarded. If at any time there are no remaining *statements*, **tagbody** returns **nil**. However, if (*go tag*) is *evaluated*, control jumps to the part of the body labeled with the *tag*. (Tags are compared with **eql**.)

A *tag* established by **tagbody** has **lexical scope** and has **dynamic extent**. Once **tagbody** has been exited, it is no longer valid to **go** to a *tag* in its body. It is permissible for **go** to jump to a **tagbody** that is not the innermost **tagbody** containing that **go**; the *tags* established by a **tagbody** only shadow other *tags* of like name.

The determination of which elements of the body are *tags* and which are *statements* is made prior to any **macro expansion** of that element. If a *statement* is a **macro form** and its **macro expansion** is an **atom**, that **atom** is treated as a *statement*, not a *tag*.

### 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
(defun f1 (flag)
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(let ((n 1))
  (tagbody
    (setq n (f2 flag #'(lambda () (go out))))
    out
    (prinl n)))
=> F1
(defun f2 (flag escape)
  (if flag (funcall escape) 2))
=> F2
(f1 nil)
>> 2
=> NIL
(f1 t)
>> 1
=> NIL
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

**go**

**Notes:**

The macros in the next figure have implicit tagbodies.

<u>do</u>	<u>do-external-symbols</u>	<u>dotimes</u>
<u>do*</u>	<u>do-symbols</u>	<u>prog</u>
<u>do-all-symbols</u>	<u>dolist</u>	<u>prog*</u>

**Figure 5–10. Macros that have implicit tagbodies.**

## Special Operator THE

**Syntax:**

**the** *value-type form => result\**

**Arguments and Values:**

*value-type*—a type specifier; not evaluated.

*form*—a form; evaluated.

*results*—the values resulting from the evaluation of *form*. These values must conform to the type supplied by *value-type*; see below.

**Description:**

**the** specifies that the values[1a] returned by *form* are of the types specified by *value-type*. The consequences are undefined if any *result* is not of the declared type.

## CLHS: Declaration DYNAMIC-EXTENT

It is permissible for *form* to *yield* a different number of *values* than are specified by *value-type*, provided that the values for which *types* are declared are indeed of those *types*. Missing values are treated as *nil* for the purposes of checking their *types*.

Regardless of number of *values* declared by *value-type*, the number of *values* returned by the the special form is the same as the number of *values* returned by *form*.

### Examples:

```
(the symbol (car (list (gensym)))) => #:G9876
(the fixnum (+ 5 7)) => 12
(the (values) (truncate 3.2 2)) => 1, 1.2
(the integer (truncate 3.2 2)) => 1, 1.2
(the (values integer) (truncate 3.2 2)) => 1, 1.2
(the (values integer float) (truncate 3.2 2)) => 1, 1.2
(the (values integer float symbol) (truncate 3.2 2)) => 1, 1.2
(the (values integer float symbol t null list)
     (truncate 3.2 2)) => 1, 1.2
(let ((i 100))
  (declare (fixnum i))
  (the fixnum (1+ i))) => 101
(let* ((x (list 'a 'b 'c))
       (y 5))
  (setf (the fixnum (car x)) y)
  x) => (5 B C)
```

**Affected By:** None.

### Exceptional Situations:

The consequences are undefined if the *values yielded* by the *form* are not of the *type* specified by *value-type*.

### See Also:

#### values

### Notes:

The **values type specifier** can be used to indicate the types of *multiple values*:

```
(the (values integer integer) (floor x y))
(the (values string t)
     (gethash the-key the-string-table))
```

**setf** can 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.

## Special Operator THROW

### Syntax:

**throw tag result-form =>**

### Arguments and Values:

*tag*—a catch tag; evaluated.

*result-form*—a form; evaluated as described below.

### Description:

**throw** causes a non-local control transfer to a catch whose tag is eq to *tag*.

*Tag* is evaluated first to produce an object called the throw tag; then *result-form* is evaluated, and its results are saved. If the *result-form* produces multiple values, then all the values are saved. The most recent outstanding catch whose *tag* is eq to the throw tag is exited; the saved results are returned as the value or values of catch.

The transfer of control initiated by **throw** is performed as described in [Section 5.2 \(Transfer of Control to an Exit Point\)](#).

### Examples:

```
(catch 'result
  (setq i 0 j 0)
  (loop (incf j 3) (incf i)
    (if (= i 3) (throw 'result (values i j))))) => 3, 9

(catch nil
  (unwind-protect (throw nil 1)
    (throw nil 2))) => 2
```

The consequences of the following are undefined because the catch of b is passed over by the first **throw**, hence portable programs must assume that its dynamic extent is terminated. The binding of the catch tag is not yet disestablished and therefore it is the target of the second **throw**.

```
(catch 'a
  (catch 'b
    (unwind-protect (throw 'a 1)
      (throw 'b 2))))
```

The following prints "The inner catch returns :SECOND-THROW" and then returns :outer-catch.

```
(catch 'foo
  (format t "The inner catch returns ~s.~%""
  (catch 'foo
    (unwind-protect (throw 'foo :first-throw)
      (throw 'foo :second-throw)))
  :outer-catch)
>> The inner catch returns :SECOND-THROW
=> :OUTER-CATCH
```

**Affected By:** None.

### Exceptional Situations:

If there is no outstanding catch tag that matches the throw tag, no unwinding of the stack is performed, and an error of type control-error is signaled. When the error is signaled, the dynamic environment is that which was in force at the point of the throw.

#### See Also:

block, catch, return-from, unwind-protect, Section 3.1 (Evaluation)

#### Notes:

catch and throw are normally used when the exit point must have dynamic scope (e.g., the throw is not lexically enclosed by the catch), while block and return are used when lexical scope is sufficient.

## Special Operator UNWIND-PROTECT

#### Syntax:

unwind-protect *protected-form* *cleanup-form\** => *result\**

#### Arguments and Values:

*protected-form*—a form.

*cleanup-form*—a form.

*results*—the values of the *protected-form*.

#### Description:

unwind-protect evaluates *protected-form* and guarantees that *cleanup-forms* are executed before unwind-protect exits, whether it terminates normally or is aborted by a control transfer of some kind. unwind-protect is intended to be used to make sure that certain side effects take place after the evaluation of *protected-form*.

If a non-local exit occurs during execution of *cleanup-forms*, no special action is taken. The *cleanup-forms* of unwind-protect are not protected by that unwind-protect.

unwind-protect protects against all attempts to exit from *protected-form*, including go, handler-case, ignore-errors, restart-case, return-from, throw, and with-simple-restart.

Undoing of handler and restart bindings during an exit happens in parallel with the undoing of the bindings of dynamic variables and catch tags, in the reverse order in which they were established. The effect of this is that *cleanup-form* sees the same handler and restart bindings, as well as dynamic variable bindings and catch tags, as were visible when the unwind-protect was entered.

#### Examples:

```
(tagbody
  (let ((x 3))
    (unwind-protect
      (if (numberp x) (go out))
```

## CLHS: Declaration DYNAMIC-EXTENT

```

        (print x)))
out
....)

```

When `go` is executed, the call to `print` is executed first, and then the transfer of control to the tag `out` is completed.

```

(defun dummy-function (x)
  (setq state 'running)
  (unless (numberp x) (throw 'abort 'not-a-number))
  (setq state (1+ x))) => DUMMY-FUNCTION
(catch 'abort (dummy-function 1)) => 2
state => 2
(catch 'abort (dummy-function 'trash)) => NOT-A-NUMBER
state => RUNNING
(catch 'abort (unwind-protect (dummy-function 'trash)
                               (setq state 'aborted))) => NOT-A-NUMBER
state => ABORTED

```

The following code is not correct:

```

(unwind-protect
  (progn (incf *access-count*)
         (perform-access))
  (decf *access-count*))

```

If an exit occurs before completion of `incf`, the `decf form` is executed anyway, resulting in an incorrect value for `*access-count*`. The correct way to code this is as follows:

```

(let ((old-count *access-count*))
  (unwind-protect
    (progn (incf *access-count*)
           (perform-access))
    (setq *access-count* old-count)))

;; The following returns 2.
(block nil
  (unwind-protect (return 1)
    (return 2)))

;; The following has undefined consequences.
(block a
  (block b
    (unwind-protect (return-from a 1)
      (return-from b 2)))))

;; The following returns 2.
(catch nil
  (unwind-protect (throw nil 1)
    (throw nil 2)))

;; The following has undefined consequences because the catch of B is
;; passed over by the first THROW, hence portable programs must assume
;; its dynamic extent is terminated. The binding of the catch tag is not
;; yet disestablished and therefore it is the target of the second throw.
(catch 'a
  (catch 'b
    (unwind-protect (throw 'a 1)
      (throw 'b 2))))

```

## CLHS: Declaration DYNAMIC-EXTENT

```
;;; The following prints "The inner catch returns :SECOND-THROW"
;;; and then returns :OUTER-CATCH.
(catch 'foo
  (format t "The inner catch returns ~s.~%""
    (catch 'foo
      (unwind-protect (throw 'foo :first-throw)
        (throw 'foo :second-throw))))
  :outer-catch)

;;; The following returns 10. The inner CATCH of A is passed over, but
;;; because that CATCH is disestablished before the THROW to A is executed,
;;; it isn't seen.
(catch 'a
  (catch 'b
    (unwind-protect (1+ (catch 'a (throw 'b 1)))
      (throw 'a 10)))

;;; The following has undefined consequences because the extent of
;;; the (CATCH 'BAR ...) exit ends when the (THROW 'FOO ...)
;;; commences.
(catch 'foo
  (catch 'bar
    (unwind-protect (throw 'foo 3)
      (throw 'bar 4)
      (print 'xxx)))))

;;; The following returns 4; XXX is not printed.
;;; The (THROW 'FOO ...) has no effect on the scope of the BAR
;;; catch tag or the extent of the (CATCH 'BAR ...) exit.
(catch 'bar
  (catch 'foo
    (unwind-protect (throw 'foo 3)
      (throw 'bar 4)
      (print 'xxx)))))

;;; The following prints 5.
(block nil
  (let ((x 5))
    (declare (special x))
    (unwind-protect (return)
      (print x))))
```

**Affected By:** None.

**Exceptional Situations:** None.

**See Also:**

[catch](#), [go](#), [handler-case](#), [restart-case](#), [return](#), [return-from](#), [throw](#), [Section 3.1 \(Evaluation\)](#)

**Notes:** None.

## Type Specifier AND

### Compound Type Specifier Kind:

Combining.

### Compound Type Specifier Syntax:

**and** *typespec*\*

### Compound Type Specifier Arguments:

*typespec*—a *type specifier*.

### Compound Type Specifier Description:

This denotes the set of all *objects* of the *type* determined by the intersection of the *typespecs*.

\* is not permitted as an argument.

The *type specifiers* (and) and **t** are equivalent. The symbol **and** is not valid as a *type specifier*, and, specifically, it is not an abbreviation for (and).

## System Class ARRAY

### Class Precedence List:

**array.t**

### Description:

An *array* contains *objects* arranged according to a Cartesian coordinate system. An *array* provides mappings from a set of *fixnums* {i0,i1,...,ir-1} to corresponding *elements* of the *array*, where 0 <=ij < dj, r is the rank of the array, and dj is the size of *dimension* j of the array.

When an *array* is created, the program requesting its creation may declare that all *elements* are of a particular *type*, called the *expressed array element type*. The implementation is permitted to *upgrade* this type in order to produce the *actual array element type*, which is the *element type* for the *array* if the *array* is actually *specialized*. See the *function upgraded-array-element-type*.

### Compound Type Specifier Kind:

Specializing.

### Compound Type Specifier Syntax:

**array** [{*element-type* | \*} [*dimension-spec*]]

*dimension-spec*::= rank | **\*** | ({*dimension* | **\***}\*)

**Compound Type Specifier Arguments:**

*dimension*—a valid array dimension.

*element-type*—a type specifier.

*rank*—a non-negative fixnum.

**Compound Type Specifier Description:**

This denotes the set of arrays whose element type, rank, and dimensions match any given element-type, rank, and dimensions. Specifically:

If element-type is the symbol `*`, arrays are not excluded on the basis of their element type. Otherwise, only those arrays are included whose actual array element type is the result of upgrading element-type; see [Section 15.1.2.1 \(Array Upgrading\)](#).

If the dimension-spec is a rank, the set includes only those arrays having that rank. If the dimension-spec is a list of dimensions, the set includes only those arrays having a rank given by the length of the dimensions, and having the indicated dimensions; in this case, `*` matches any value for the corresponding dimension. If the dimension-spec is the symbol `*`, the set is not restricted on the basis of rank or dimension.

**See Also:**

[\\*print-array\\*](#), [aref](#), [make-array](#), [vector](#), [Section 2.4.8.12 \(Sharpsign A\)](#), [Section 22.1.3.8 \(Printing Other Arrays\)](#)

**Notes:**

Note that the type (`array t`) is a proper subtype of the type (`array *`). The reason is that the type (`array t`) is the set of arrays that can hold any object (the elements are of type t, which includes all objects). On the other hand, the type (`array *`) is the set of all arrays whatsoever, including for example arrays that can hold only characters. The type (`array character`) is not a subtype of the type (`array t`); the two sets are disjoint because the type (`array character`) is not the set of all arrays that can hold characters, but rather the set of arrays that are specialized to hold precisely characters and no other objects.

**Type ATOM****Supertypes:**

atom, t

**Description:**

It is equivalent to (`not cons`).

**Type BOOLEAN****Supertypes:**

**boolean, symbol, t****Description:**

The *type boolean* contains the *symbols t* and *nil*, which represent true and false, respectively.

**See Also:**

**t (constant variable), nil (constant variable), if, not, complement**

**Notes:**

Conditional operations, such as *if*, permit the use of *generalized booleans*, not just *booleans*; any *non-nil* value, not just *t*, counts as true for a *generalized boolean*. However, as a matter of convention, the *symbol t* is considered the canonical value to use even for a *generalized boolean* when no better choice presents itself.

**Type BASE-CHAR****Supertypes:****base-char, character, t****Description:**

The *type base-char* is defined as the *upgraded array element type* of *standard-char*. An *implementation* can support additional *subtypes* of *type character* (besides the ones listed in this standard) that might or might not be *supertypes* of *type base-char*. In addition, an *implementation* can define *base-char* to be the *same type* as *character*.

*Base characters* are distinguished in the following respects:

1. *The type standard-char is a subrepertoire of the type base-char.*
2. *The selection of base characters that are not standard characters is implementation defined.*
3. *Only objects of the type base-char can be elements of a base string.*
4. *No upper bound is specified for the number of characters in the base-char repertoire; the size of that repertoire is implementation-defined. The lower bound is 96, the number of standard characters.*

Whether a character is a *base character* depends on the way that an *implementation* represents *strings*, and not any other properties of the *implementation* or the host operating system. For example, one implementation might encode all *strings* as characters having 16-bit encodings, and another might have two kinds of *strings*: those with characters having 8-bit encodings and those with characters having 16-bit encodings. In the first *implementation*, the *type base-char* is equivalent to the *type character*: there is only one kind of *string*. In the second *implementation*, the *base characters* might be those *characters* that could be stored in a *string* of *characters* having 8-bit encodings. In such an implementation, the *type base-char* is a *proper subtype* of the *type character*.

The *type standard-char* is a *subtype* of *type base-char*.

## Type BASE-STRING

Supertypes:

base-string, string, vector, array, sequence, t

Description:

The type base-string is equivalent to (`vector base-char`). The base string representation is the most efficient string representation that can hold an arbitrary sequence of standard characters.

Compound Type Specifier Kind:

Abbreviating.

Compound Type Specifier Syntax:

`base-string [size]`

Compound Type Specifier Arguments:

size—a non-negative fixnum, or the symbol \*.

Compound Type Specifier Description:

This is equivalent to the type (`vector base-char size`); that is, the set of base strings of size size.

## Type BIGNUM

Supertypes:

bignum, integer, rational, real, number, t

Description:

The type bignum is defined to be exactly (and integer (not fixnum)).

## Type BIT

Supertypes:

bit, unsigned-byte, signed-byte, integer, rational, real, number, t

Description:

The type bit is equivalent to the type (`integer 0 1`) and (`unsigned-byte 1`).

**System Class BROADCAST-STREAM****Class Precedence List:****broadcast-stream, stream, t****Description:**

A *broadcast stream* is an *output stream* which has associated with it a set of zero or more *output streams* such that any output sent to the *broadcast stream* gets passed on as output to each of the associated *output streams*. (If a *broadcast stream* has no *component streams*, then all output to the *broadcast stream* is discarded.)

The set of operations that may be performed on a *broadcast stream* is the intersection of those for its associated *output streams*.

Some output operations (e.g., *fresh-line*) return *values* based on the state of the *stream* at the time of the operation. Since these *values* might differ for each of the *component streams*, it is necessary to describe their return value specifically:

- \* *stream-element-type* returns the value from the last component stream, or *t* if there are no component streams.
- \* *fresh-line* returns the value from the last component stream, or *nil* if there are no component streams.
- \* The functions *file-length*, *file-position*, *file-string-length*, and *stream-external-format* return the value from the last component stream; if there are no component streams, *file-length* and *file-position* return 0, *file-string-length* returns 1, and *stream-external-format* returns :default.
- \* The functions *streamp* and *output-stream-p* always return *true* for broadcast streams.
- \* The functions *open-stream-p* tests whether the *broadcast stream* is *open*[2], not whether its component streams are *open*.
- \* The functions *input-stream-p* and *interactive-stream-p* return an implementation-defined generalized boolean value.
- \* For the input operations *clear-input*, *listen*, *peek-char*, *read-byte*, *read-char-no-hang*, *read-char*, *read-line*, and *unread-char*, the consequences are undefined if the indicated operation is performed. However, an implementation is permitted to define such a behavior as an implementation-dependent extension.

For any output operations not having their return values explicitly specified above or elsewhere in this document, it is defined that the *values* returned by such an operation are the *values* resulting from performing the operation on the last of its *component streams*; the *values* resulting from performing the operation on all preceding *streams* are discarded. If there are no *component streams*, the value is implementation-dependent.

**See Also:****broadcast-stream-streams, make-broadcast-stream****System Class BIT-VECTOR****Class Precedence List:****bit-vector, vector, array, sequence, t**

**Description:**

A *bit vector* is a *vector* the *element type* of which is *bit*.

The *type bit-vector* is a *subtype* of *type vector*, for *bit-vector* means (vector bit).

**Compound Type Specifier Kind:**

Abbreviating.

**Compound Type Specifier Syntax:**

**bit-vector** [*size*]

**Compound Type Specifier Arguments:**

*size*—a non-negative *fixnum*, or the *symbol* \*.

**Compound Type Specifier Description:**

This denotes the same *type* as the *type* (array bit (*size*)); that is, the set of *bit vectors* of size *size*.

**See Also:****System Class BUILT-IN-CLASS****Class Precedence List:**

[built-in-class](#), [class](#), [standard-object](#), [t](#)

**Description:**

A *built-in class* is a *class* whose *instances* have restricted capabilities or special representations. Attempting to use *defclass* to define *subclasses* of a *built-in class* signals an error of *type error*. Calling *make-instance* to create an *instance* of a *built-in class* signals an error of *type error*. Calling *slot-value* on an *instance* of a *built-in class* signals an error of *type error*. Redefining a *built-in class* or using *change-class* to change the *class* of an *instance* to or from a *built-in class* signals an error of *type error*. However, *built-in classes* can be used as *parameter specializers* in *methods*.

**System Class CHARACTER****Class Precedence List:**

[character](#), [t](#)

**Description:**

A *character* is an *object* that represents a unitary token in an aggregate quantity of text; see [Section 13.1 \(Character Concepts\)](#).

The types **base-char** and **extended-char** form an *exhaustive partition* of the type **character**.

See Also:

## **System Class CLASS**

Class Precedence List:

class, standard-object, **t**

Description:

The type **class** represents objects that determine the structure and behavior of their instances. Associated with an object of type **class** is information describing its place in the directed acyclic graph of classes, its slots, and its options.

## **Type COMPILED-FUNCTION**

Supertypes:

compiled-function, function, **t**

Description:

Any function may be considered by an implementation to be a a compiled function if it contains no references to macros that must be expanded at run time, and it contains no unresolved references to load time values. See Section 3.2.2 (Compilation Semantics).

Functions whose definitions appear lexically within a file that has been compiled with compile-file and then loaded with load are of type **compiled-function**. Functions produced by the compile function are of type **compiled-function**. Other functions might also be of type **compiled-function**.

## **System Class COMPLEX**

Class Precedence List:

complex, number, **t**

Description:

The type **complex** includes all mathematical complex numbers other than those included in the type **rational**. Complexes are expressed in Cartesian form with a real part and an imaginary part, each of which is a real. The real part and imaginary part are either both rational or both of the same float type. The imaginary part can be a float zero, but can never be a rational zero, for such a number is always represented by Common Lisp as a rational rather than a complex.

Compound Type Specifier Kind:

Specializing.

**Compound Type Specifier Syntax:**

**complex** [*typespec* /\*]

**Compound Type Specifier Arguments:**

*typespec*—a type specifier that denotes a subtype of type real.

**Compound Type Specifier Description:**

Every element of this type is a complex whose real part and imaginary part are each of type (upgraded-complex-part-type *typespec*). This type encompasses those complexes that can result by giving numbers of type *typespec* to complex.

(complex type-specifier) refers to all complexes that can result from giving numbers of type *type-specifier* to the function complex, plus all other complexes of the same specialized representation.

**See Also:****Notes:**

The input syntax for a complex with real part r and imaginary part i is #C(r i). For further details, see [Section 2.4 \(Standard Macro Characters\)](#).

For every float, n, there is a complex which represents the same mathematical number and which can be obtained by (COERCE n 'COMPLEX).

**System Class CONCATENATED-STREAM****Class Precedence List:**

concatenated-stream, stream, t

**Description:**

A concatenated stream is an input stream which is a composite stream of zero or more other input streams, such that the sequence of data which can be read from the concatenated stream is the same as the concatenation of the sequences of data which could be read from each of the constituent streams.

Input from a concatenated stream is taken from the first of the associated input streams until it reaches end of file[1]; then that stream is discarded, and subsequent input is taken from the next input stream, and so on. An end of file on the associated input streams is always managed invisibly by the concatenated stream—the only time a client of a concatenated stream sees an end of file is when an attempt is made to obtain data from the concatenated stream but it has no remaining input streams from which to obtain such data.

**See Also:**

concatenated-stream-streams, make-concatenated-stream

**System Class CONS****Class Precedence List:****cons, list, sequence, t****Description:**

A cons is a compound object having two components, called the car and cdr. These form a dotted pair. Each component can be any object.

**Compound Type Specifier Kind:**

Specializing.

**Compound Type Specifier Syntax:****cons [car-typespec [cdr-typespec]]****Compound Type Specifier Arguments:**

car-typespec—a type specifier, or the symbol \*. The default is the symbol \*.

cdr-typespec—a type specifier, or the symbol \*. The default is the symbol \*.

**Compound Type Specifier Description:**

This denotes the set of conses whose car is constrained to be of type car-typespec and whose cdr is constrained to be of type cdr-typespec. (If either car-typespec or cdr-typespec is \*, it is as if the type t had been denoted.)

**See Also:****System Class ECHO-STREAM****Class Precedence List:****echo-stream, stream, t****Description:**

An echo stream is a bidirectional stream that gets its input from an associated input stream and sends its output to an associated output stream.

All input taken from the input stream is echoed to the output stream. Whether the input is echoed immediately after it is encountered, or after it has been read from the input stream is implementation-dependent.

**See Also:****echo-stream-input-stream, echo-stream-output-stream, make-echo-stream**

## Type Specifier EQL

**Compound Type Specifier Kind:**

Combining.

**Compound Type Specifier Syntax:**

**eql** *object*

**Compound Type Specifier Arguments:**

*object*—an *object*.

**Compound Type Specifier Description:**

Represents the *type* of all *x* for which (**eql** *object* *x*) is true.

The argument *object* is required. The *object* can be \*, but if so it denotes itself (the symbol \*) and does not represent an unspecified value. The symbol **eql** is not valid as an atomic type specifier.

## Type EXTENDED-CHAR

**Supertypes:**

**extended-char**, **character**, **t**

**Description:**

The *type extended-char* is equivalent to the *type* (*and character* (not *base-char*)).

**Notes:**

The *type extended-char* might have no *elements*[4] in *implementations* in which all *characters* are of *type base-char*.

## System Class FILE-STREAM

**Class Precedence List:**

**file-stream**, **stream**, **t**

**Description:**

An *object* of *type file-stream* is a *stream* the direct source or sink of which is a *file*. Such a *stream* is created explicitly by **open** and **with-open-file**, and implicitly by *functions* such as **load** that process *files*.

**See Also:**

**load**, **open**, **with-open-file**

## Type FIXNUM

**Supertypes:**

fixnum, integer, rational, real, number, t

**Description:**

A fixnum is an integer whose value is between most-negative-fixnum and most-positive-fixnum inclusive. Exactly which integers are fixnums is implementation-defined. The type fixnum is required to be a supertype of (signed-byte 16).

## System Class FLOAT

**Class Precedence List:**

float, real, number, t

**Description:**

A float is a mathematical rational (but *not* a Common Lisp rational) of the form  $s*f*b^{e-p}$ , where s is +1 or -1, the *sign*; b is an integer greater than 1, the *base* or *radix* of the representation; p is a positive integer, the *precision* (in base-b digits) of the float; f is a positive integer between  $b^{p-1}$  and  $b^{p-1}$  (inclusive), the *significand*; and e is an integer, the *exponent*. The value of p and the range of e depends on the implementation and on the type of float within that implementation. In addition, there is a floating-point zero; depending on the implementation, there can also be a "minus zero". If there is no minus zero, then 0.0 and -0.0 are both interpreted as simply a floating-point zero. (= 0.0 -0.0) is always true. If there is a minus zero, (eql -0.0 0.0) is false, otherwise it is true.

The types short-float, single-float, double-float, and long-float are subtypes of type float. Any two of them must be either disjoint types or the same type; if the same type, then any other types between them in the above ordering must also be the same type. For example, if the type single-float and the type long-float are the same type, then the type double-float must be the same type also.

**Compound Type Specifier Kind:**

Abbreviating.

**Compound Type Specifier Syntax:**

**float** [*lower-limit* [*upper-limit*]]

**Compound Type Specifier Arguments:**

*lower-limit*, *upper-limit*—interval designators for type float. The defaults for each of *lower-limit* and *upper-limit* is the symbol \*.

**Compound Type Specifier Description:**

This denotes the floats on the interval described by *lower-limit* and *upper-limit*.

**See Also:**

[Figure 2–9, Section 2.3.2 \(Constructing Numbers from Tokens\)](#), [Section 22.1.3.1.3 \(Printing Floats\)](#)

**Notes:**

Note that all mathematical integers are representable not only as Common Lisp *reals*, but also as *complex floats*. For example, possible representations of the mathematical number 1 include the *integer* 1, the *float* 1.0, or the *complex* #C(1.0 0.0).

## System Class FUNCTION

**Class Precedence List:**

### **function.t**

**Description:**

A *function* is an *object* that represents code to be executed when an appropriate number of arguments is supplied. A *function* is produced by the *function special form*, the *function coerce*, or the *function compile*. A *function* can be directly invoked by using it as the first argument to *funcall*, *apply*, or *multiple-value-call*.

**Compound Type Specifier Kind:**

Specializing.

**Compound Type Specifier Syntax:**

**function** [*arg-typespec* [*value-typespec*]]

```
arg-typespec ::= ( typespec*
                    [&optional typespec*]
                    [&rest typespec]
                    [&key (keyword typespec)*] )
```

**Compound Type Specifier Arguments:**

*typespec*—*a type specifier*.

*value-typespec*—*a type specifier*.

**Compound Type Specifier Description:**

The list form of the *function type-specifier* can be used only for declaration and not for discrimination. Every element of this *type* is a *function* that accepts arguments of the types specified by the *arg-types* and returns values that are members of the *types* specified by *value-type*. The &optional, &rest, &key, and &allow-other-keys markers can appear in the list of argument types. The *type specifier* provided with &rest is the *type* of each actual argument, not the *type* of the corresponding variable.

The &key parameters should be supplied as lists of the form (*keyword type*). The *keyword* must be a valid keyword-name symbol as must be supplied in the actual arguments of a call. This is usually a *symbol* in the

## CLHS: Declaration DYNAMIC-EXTENT

KEYWORD package but can be any *symbol*. When &key is given in a **function type specifier lambda list**, the *keyword parameters* given are exhaustive unless &allow-other-keys is also present. &allow-other-keys is an indication that other keyword arguments might actually be supplied and, if supplied, can be used. For example, the *type* of the **function make-list** could be declared as follows:

```
(function ((integer 0) &key (:initial-element t)) list)
```

The *value-type* can be a **values type specifier** in order to indicate the *types of multiple values*.

Consider a declaration of the following form:

```
(ftype (function (arg0-type arg1-type ...) val-type) f))
```

Any *form* (f arg0 arg1 ...) within the scope of that declaration is equivalent to the following:

```
(the val-type (f (the arg0-type arg0) (the arg1-type arg1) ...))
```

That is, the consequences are undefined if any of the arguments are not of the specified *types* or the result is not of the specified *type*. In particular, if any argument is not of the correct *type*, the result is not guaranteed to be of the specified *type*.

Thus, an **ftype** declaration for a *function* describes *calls* to the *function*, not the actual definition of the *function*.

Consider a declaration of the following form:

```
(type (function (arg0-type arg1-type ...) val-type) fn-valued-variable)
```

This declaration has the interpretation that, within the scope of the declaration, the consequences are unspecified if the value of fn-valued-variable is called with arguments not of the specified *types*; the value resulting from a valid call will be of type val-type.

As with variable type declarations, nested declarations imply intersections of *types*, as follows:

\* Consider the following two declarations of **ftype**:

```
(ftype (function (arg0-type1 arg1-type1 ...) val-type1) f))
```

and

```
(ftype (function (arg0-type2 arg1-type2 ...) val-type2) f))
```

If both these declarations are in effect, then within the shared scope of the declarations, calls to f can be treated as if f were declared as follows:

```
(ftype (function ((and arg0-type1 arg0-type2) (and arg1-type1 arg1-type2 ...) ...) ...)  
      (and val-type1 val-type2))  
f))
```

It is permitted to ignore one or all of the **ftype** declarations in force.

\* If two (or more) type declarations are in effect for a variable, and they are both function declarations, the declarations combine similarly.

**System Class GENERIC–FUNCTION****Class Precedence List:****generic-function, function, t****Description:**

A generic function is a function whose behavior depends on the classes or identities of the arguments supplied to it. A generic function object contains a set of methods, a lambda list, a method combination type, and other information. The methods define the class-specific behavior and operations of the generic function; a method is said to specialize a generic function. When invoked, a generic function executes a subset of its methods based on the classes or identities of its arguments.

A generic function can be used in the same ways that an ordinary function can be used; specifically, a generic function can be used as an argument to funcall and apply, and can be given a global or a local name.

**System Class HASH–TABLE****Class Precedence List:****hash-table, t****Description:**

Hash tables provide a way of mapping any object (a key) to an associated object (a value).

**See Also:****Notes:**

The intent is that this mapping be implemented by a hashing mechanism, such as that described in Section 6.4 "Hashing" of The Art of Computer Programming, Volume 3 (pp506–549). In spite of this intent, no conforming implementation is required to use any particular technique to implement the mapping.

**System Class INTEGER****Class Precedence List:****integer, rational, real, number, t****Description:**

An integer is a mathematical integer. There is no limit on the magnitude of an integer.

The types fixnum and bignum form an exhaustive partition of type integer.

**Compound Type Specifier Kind:**

Abbreviating.

**Compound Type Specifier Syntax:**

**integer** [*lower-limit* [*upper-limit*]]

**Compound Type Specifier Arguments:**

*lower-limit*, *upper-limit*—*interval designators* for **type integer**. The defaults for each of *lower-limit* and *upper-limit* is the symbol \*.

**Compound Type Specifier Description:**

This denotes the integers on the interval described by *lower-limit* and *upper-limit*.

**See Also:**

[Figure 2–9, Section 2.3.2 \(Constructing Numbers from Tokens\)](#), [Section 22.1.3.1.1 \(Printing Integers\)](#)

**Notes:**

The **type** (`integer lower upper`), where *lower* and *upper* are **most-negative-fixnum** and **most-positive-fixnum**, respectively, is also called fixnum.

The **type** (`integer 0 1`) is also called bit. The **type** (`integer 0 *`) is also called unsigned-byte.

**Type KEYWORD****Supertypes:**

keyword, symbol, t

**Description:**

The **type keyword** includes all symbols interned the KEYWORD package.

*Interning a symbol* in the KEYWORD package has three automatic effects:

1. It causes the symbol to become bound to itself.
2. It causes the symbol to become an external symbol of the KEYWORD package.
3. It causes the symbol to become a constant variable.

**See Also:**

keywordp

**System Class LIST****Class Precedence List:**

list, sequence, t

**Description:**

A list is a chain of conses in which the car of each cons is an element of the list, and the cdr of each cons is either the next link in the chain or a terminating atom.

A proper list is a chain of conses terminated by the empty list, ( ), which is itself a proper list. A dotted list is a list which has a terminating atom that is not the empty list. A circular list is a chain of conses that has no termination because some cons in the chain is the cdr of a later cons.

Dotted lists and circular lists are also lists, but usually the unqualified term "list" within this specification means proper list. Nevertheless, the type list unambiguously includes dotted lists and circular lists.

For each element of a list there is a cons. The empty list has no elements and is not a cons.

The types cons and null form an exhaustive partition of the type list.

**See Also:****System Class LOGICAL-PATHNAME****Class Precedence List:****logical-pathname pathname t****Description:**

A pathname that uses a namestring syntax that is implementation-independent, and that has component values that are implementation-independent. Logical pathnames do not refer directly to filenames.

**See Also:****Type Specifier MEMBER****Compound Type Specifier Kind:**

Combining.

**Compound Type Specifier Syntax:**

**member** *object*\*

**Compound Type Specifier Arguments:**

*object*---an object.

**Compound Type Specifier Description:**

This denotes the set containing the named *objects*. An object is of this type if and only if it is eql to one of the specified *objects*.

## CLHS: Declaration DYNAMIC-EXTENT

The type specifiers (member) and nil are equivalent. \* can be among the *objects*, but if so it denotes itself (the symbol \*) and does not represent an unspecified value. The symbol member is not valid as a type specifier; and, specifically, it is not an abbreviation for either (member) or (member \*).

**See Also:**

the type eql

## System Class METHOD-COMBINATION

**Class Precedence List:**

method-combination, t

**Description:**

Every method combination object is an indirect instance of the class method-combination. A method combination object represents the information about the method combination being used by a generic function. A method combination object contains information about both the type of method combination and the arguments being used with that type.

## System Class METHOD

**Class Precedence List:**

method, t

**Description:**

A method is an object that represents a modular part of the behavior of a generic function.

A method contains code to implement the method's behavior, a sequence of parameter specializers that specify when the given method is applicable, and a sequence of qualifiers that is used by the method combination facility to distinguish among methods. Each required parameter of each method has an associated parameter specializer, and the method will be invoked only on arguments that satisfy its parameter specializers.

The method combination facility controls the selection of methods, the order in which they are run, and the values that are returned by the generic function. The object system offers a default method combination type and provides a facility for declaring new types of method combination.

**See Also:**

## Type Specifier MOD

**Compound Type Specifier Kind:**

Abbreviating.

**Compound Type Specifier Syntax:**

**mod** *n***Compound Type Specifier Arguments:***n*—a positive *integer*.**Compound Type Specifier Description:**

This denotes the set of non-negative *integers* less than *n*. This is equivalent to (*integer 0 (n)*) or to (*integer 0 m*), where *m=n-1*.

The argument is required, and cannot be *\**.

The symbol **mod** is not valid as a *type specifier*.

**Type NIL****Supertypes:**all *types***Description:**

The *type nil* contains no *objects* and so is also called the *empty type*. The *type nil* is a *subtype* of every *type*. No *object* is of *type nil*.

**Notes:**

The *type* containing the *object nil* is the *type null*, not the *type nil*.

**Type Specifier NOT****Compound Type Specifier Kind:**

Combining.

**Compound Type Specifier Syntax:****not** *typespec***Compound Type Specifier Arguments:***typespec*—a *type specifier*.**Compound Type Specifier Description:**

This denotes the set of all *objects* that are not of the *type typespec*.

The argument is required, and cannot be *\**.

The symbol **not** is not valid as a *type specifier*.

## **System Class NULL**

**Class Precedence List:**

**null, symbol, list, sequence, t**

**Description:**

The only *object* of *type null* is **nil**, which represents the *empty list* and can also be notated ( ).

**See Also:**

## **System Class NUMBER**

**Class Precedence List:**

**number, t**

**Description:**

The *type number* contains *objects* which represent mathematical numbers. The *types real* and *complex* are *disjoint subtypes* of *number*.

The *function =* tests for numerical equality. The *function eql*, when its arguments are both *numbers*, tests that they have both the same *type* and numerical value. Two *numbers* that are the *same* under *eql* or *=* are not necessarily the *same* under *eq*.

**Notes:**

Common Lisp differs from mathematics on some naming issues. In mathematics, the set of real numbers is traditionally described as a subset of the complex numbers, but in Common Lisp, the *type real* and the *type complex* are disjoint. The Common Lisp type which includes all mathematical complex numbers is called *number*. The reasons for these differences include historical precedent, compatibility with most other popular computer languages, and various issues of time and space efficiency.

## **Type Specifier OR**

**Compound Type Specifier Kind:**

Combining.

**Compound Type Specifier Syntax:**

**or typespec\***

**Compound Type Specifier Arguments:**

*typespec*—a *type specifier*.

**Compound Type Specifier Description:**

This denotes the set of all *objects* of the *type* determined by the union of the *typespecs*. For example, the *type list* by definition is the same as (or null cons). Also, the value returned by *position* is an *object* of *type* (or null (integer 0 \*)); i.e., either *nil* or a non-negative *integer*.

\* is not permitted as an argument.

The *type specifiers* (or) and *nil* are equivalent. The symbol *or* is not valid as a *type specifier*; and, specifically, it is not an abbreviation for (or).

**System Class PACKAGE****Class Precedence List:**

**package.t**

**Description:**

A *package* is a *namespace* that maps *symbol names* to *symbols*; see [Section 11.1 \(Package Concepts\)](#).

**See Also:****System Class PATHNAME****Class Precedence List:**

**pathname.t**

**Description:**

A *pathname* is a structured *object* which represents a *filename*.

There are two kinds of *pathnames*—*physical pathnames* and *logical pathnames*.

**System Class RATIO****Class Precedence List:**

**ratio.rational.real.number.t**

**Description:**

A *ratio* is a *number* representing the mathematical ratio of two non-zero integers, the numerator and denominator, whose greatest common divisor is one, and of which the denominator is positive and greater than one.

**See Also:**

[Figure 2–9, Section 2.3.2 \(Constructing Numbers from Tokens\)](#), [Section 22.1.3.1.2 \(Printing Ratios\)](#)

**System Class RATIONAL****Class Precedence List:****rational, real, number, t****Description:**

The canonical representation of a *rational* is as an *integer* if its value is integral, and otherwise as a *ratio*.

The *types integer* and *ratio* are *disjoint subtypes* of *type rational*.

**Compound Type Specifier Kind:**

Abbreviating.

**Compound Type Specifier Syntax:****rational [lower-limit [upper-limit]]****Compound Type Specifier Arguments:**

*lower-limit*, *upper-limit*—*interval designators* for *type rational*. The defaults for each of *lower-limit* and *upper-limit* is the *symbol* \*.

**Compound Type Specifier Description:**

This denotes the *rationals* on the interval described by *lower-limit* and *upper-limit*.

**System Class READTABLE****Class Precedence List:****readtable, t****Description:**

A *readtable* maps *characters* into *syntax types* for the *Lisp reader*; see [Section 2 \(Syntax\)](#). A *readtable* also contains associations between *macro characters* and their *reader macro functions*, and records information about the case conversion rules to be used by the *Lisp reader* when parsing *symbols*.

Each *simple character* must be representable in the *readtable*. It is *implementation-defined* whether *non-simple characters* can have syntax descriptions in the *readtable*.

**See Also:****System Class REAL****Class Precedence List:**

**real, number, t****Description:**

The type real includes all numbers that represent mathematical real numbers, though there are mathematical real numbers (e.g., irrational numbers) that do not have an exact representation in Common Lisp. Only *reals* can be ordered using the  $\leq$ ,  $\geq$ ,  $\leq\equiv$ , and  $\geq\equiv$  functions.

The types rational and float are disjoint subtypes of type real.

**Compound Type Specifier Kind:**

Abbreviating.

**Compound Type Specifier Syntax:**

**real** [*lower-limit* [*upper-limit*]]

**Compound Type Specifier Arguments:**

*lower-limit*, *upper-limit*—interval designators for type real. The defaults for each of *lower-limit* and *upper-limit* is the symbol \*.

**Compound Type Specifier Description:**

This denotes the *reals* on the interval described by *lower-limit* and *upper-limit*.

**System Class RANDOM-STATE****Class Precedence List:****random-state, t****Description:**

A random state object contains state information used by the pseudo-random number generator. The nature of a random state object is implementation-dependent. It can be printed out and successfully read back in by the same implementation, but might not function correctly as a random state in another implementation.

Implementations are required to provide a read syntax for objects of type random-state, but the specific nature of that syntax is implementation-dependent.

**See Also:**

**\*random-state\***, **random**, Section 22.1.3.10 (Printing Random States)

**System Class RESTART****Class Precedence List:**

**restart\_t****Description:**

An object of type **restart** represents a function that can be called to perform some form of recovery action, usually a transfer of control to an outer point in the running program.

An implementation is free to implement a restart in whatever manner is most convenient; a restart has only dynamic extent relative to the scope of the binding form which establishes it.

**Type Specifier SATISFIES****Compound Type Specifier Kind:**

Predicating.

**Compound Type Specifier Syntax:**

**satisfies** *predicate-name*

**Compound Type Specifier Arguments:**

*predicate-name*—a symbol.

**Compound Type Specifier Description:**

This denotes the set of all objects that satisfy the predicate *predicate-name*, which must be a symbol whose global function definition is a one-argument predicate. A name is required for *predicate-name*; lambda expressions are not allowed. For example, the type specifier (*and integer* (satisfies evenp)) denotes the set of all even integers. The form (*typep x* ' (satisfies p)) is equivalent to (*if (p x) t nil*).

The argument is required. The symbol\* can be the argument, but it denotes itself (the symbol\*), and does not represent an unspecified value.

The symbol **satisfies** is not valid as a type specifier.

**System Class SEQUENCE****Class Precedence List:****sequence\_t****Description:**

Sequences are ordered collections of objects, called the elements of the sequence.

The types vector and the type list are disjoint subtypes of type sequence, but are not necessarily an exhaustive partition of sequence.

When viewing a vector as a sequence, only the active elements of that vector are considered elements of the sequence; that is, sequence operations respect the fill pointer when given sequences represented as vectors.

## Type SIGNED-BYTE

**Supertypes:**

signed-byte, integer, rational, real, number, t

**Description:**

The atomic type specifier **signed-byte** denotes the same type as is denoted by the type specifier **integer**; however, the list forms of these two type specifiers have different semantics.

**Compound Type Specifier Kind:**

Abbreviating.

**Compound Type Specifier Syntax:**

**signed-byte** [*s* / \*]

**Compound Type Specifier Arguments:**

*s*—a positive integer.

**Compound Type Specifier Description:**

This denotes the set of integers that can be represented in two's-complement form in a byte of *s* bits. This is equivalent to (integer - $2^s-1$   $2^s-1-1$ ). The type **signed-byte** or the type (signed-byte \*) is the same as the type integer.

## Type SHORT-FLOAT, SINGLE-FLOAT, DOUBLE-FLOAT, LONG-FLOAT

**Supertypes:**

short-float: short-float, float, real, number, t

single-float: single-float, float, real, number, t

double-float: double-float, float, real, number, t

long-float: long-float, float, real, number, t

**Description:**

For the four defined subtypes of type float, it is true that intermediate between the type short-float and the

type long-float are the type single-float and the type double-float. The precise definition of these categories is implementation-defined. The precision (measured in "bits", computed as  $p \log 2b$ ) and the exponent size (also measured in "bits," computed as  $\log 2(n+1)$ , where  $n$  is the maximum exponent value) is recommended to be at least as great as the values in the next figure. Each of the defined subtypes of type float might or might not have a minus zero.

Format	Minimum Precision	Minimum Exponent Size
--------	-------------------	-----------------------

Short	13 bits	5 bits	Single	24 bits	8 bits	Double	50 bits	8 bits	Long	50 bits	8 bits
-------	---------	--------	--------	---------	--------	--------	---------	--------	------	---------	--------

**Figure 12–12. Recommended Minimum Floating–Point Precision and Exponent Size**

There can be fewer than four internal representations for floats. If there are fewer distinct representations, the following rules apply:

- If there is only one, it is the type single-float. In this representation, an object is simultaneously of types single-float, double-float, short-float, and long-float.
- Two internal representations can be arranged in either of the following ways:
  - ◆ Two types are provided: single-float and short-float. An object is simultaneously of types single-float, double-float, and long-float.
  - ◆ Two types are provided: single-float and double-float. An object is simultaneously of types single-float and short-float, or double-float and long-float.
- Three internal representations can be arranged in either of the following ways:
  - ◆ Three types are provided: short-float, single-float, and double-float. An object can simultaneously be of type double-float and long-float.
  - ◆ Three types are provided: single-float, double-float, and long-float. An object can simultaneously be of types single-float and short-float.

### Compound Type Specifier Kind:

Abbreviating.

### Compound Type Specifier Syntax:

**short-float** [*short-lower-limit* [*short-upper-limit*]]

**single-float** [*single-lower-limit* [*single-upper-limit*]]

**double-float** [*double-lower-limit* [*double-upper-limit*]]

**long-float** [*long-lower-limit* [*long-upper-limit*]]

### Compound Type Specifier Arguments:

*short-lower-limit*, *short-upper-limit*—interval designators for type short-float. The defaults for each of *lower-limit* and *upper-limit* is the symbol \*.

*single-lower-limit*, *single-upper-limit*---interval designators for type single-float. The defaults for each of *lower-limit* and *upper-limit* is the symbol \*.

*double-lower-limit*, *double-upper-limit*---interval designators for type double-float. The defaults for each of *lower-limit* and *upper-limit* is the symbol \*.

*long-lower-limit*, *long-upper-limit*---interval designators for type long-float. The defaults for each of *lower-limit* and *upper-limit* is the symbol \*.

### Compound Type Specifier Description:

Each of these denotes the set of floats of the indicated type that are on the interval specified by the interval designators.

## Type SIMPLE-ARRAY

### Supertypes:

simple-array, array, t

### Description:

The type of an array that is not displaced to another array, has no fill pointer, and is not expressly adjustable is a subtype of type simple-array. The concept of a simple array exists to allow the implementation to use a specialized representation and to allow the user to declare that certain values will always be simple arrays.

The types simple-vector, simple-string, and simple-bit-vector are disjoint subtypes of type simple-array, for they respectively mean (simple-array t (\*)), the union of all (simple-array c (\*)) for any c being a subtype of type character, and (simple-array bit (\*)).

### Compound Type Specifier Kind:

Specializing.

### Compound Type Specifier Syntax:

simple-array [{element-type / \*} [dimension-spec]]

dimension-spec::= rank | \* | ({dimension | \*}\*)

### Compound Type Specifier Arguments:

*dimension*---a valid array dimension.

*element-type*---a type specifier.

*rank*---a non-negative fixnum.

### Compound Type Specifier Description:

## CLHS: Declaration DYNAMIC-EXTENT

This compound type specifier is treated exactly as the corresponding compound type specifier for type array would be treated, except that the set is further constrained to include only simple arrays.

**Notes:**

It is implementation-dependent whether displaced arrays, vectors with fill pointers, or arrays that are actually adjustable are simple arrays.

(simple-array \*) refers to all simple arrays regardless of element type, (simple-array type-specifier) refers only to those simple arrays that can result from giving type-specifier as the :element-type argument to make-array.

## Type SIMPLE-BASE-STRING

**Supertypes:**

simple-base-string, base-string, simple-string, string, vector, simple-array, array, sequence, t

**Description:**

The type simple-base-string is equivalent to (simple-array base-char (\*)).

**Compound Type Specifier Kind:**

Abbreviating.

**Compound Type Specifier Syntax:**

simple-base-string [size]

**Compound Type Specifier Arguments:**

size—a non-negative fixnum, or the symbol \*.

**Compound Type Specifier Description:**

This is equivalent to the type (simple-array base-char (size)); that is, the set of simple base strings of size size.

## Type SIMPLE-BIT-VECTOR

**Supertypes:**

simple-bit-vector, bit-vector, vector, simple-array, array, sequence, t

**Description:**

The type of a bit vector that is not displaced to another array, has no fill pointer, and is not expressly adjustable is a subtype of type simple-bit-vector.

**Compound Type Specifier Kind:**

Abbreviating.

**Compound Type Specifier Syntax:**

**simple-bit-vector** [*size*]

**Compound Type Specifier Arguments:**

*size*—a non-negative *fixnum*, or the *symbol* \*. The default is the *symbol* \*.

**Compound Type Specifier Description:**

This denotes the same type as the *type* (*simple-array bit* (*size*)); that is, the set of *simple bit vectors* of size *size*.

**Type SIMPLE-STRING****Supertypes:**

simple-string, string, vector, simple-array, array, sequence, t

**Description:**

A *simple string* is a specialized one-dimensional *simple array* whose *elements* are of *type character* or a *subtype* of *type character*. When used as a *type specifier* for object creation, simple-string means (*simple-array character* (*size*)).

**Compound Type Specifier Kind:**

Abbreviating.

**Compound Type Specifier Syntax:**

**simple-string** [*size*]

**Compound Type Specifier Arguments:**

*size*—a non-negative *fixnum*, or the *symbol* \*.

**Compound Type Specifier Description:**

This denotes the union of all *types* (*simple-array c* (*size*)) for all *subtypes* *c* of *character*; that is, the set of *simple strings* of size *size*.

**Type SIMPLE-VECTOR****Supertypes:**

**simple–vector, vector, simple–array, array, sequence, t****Description:**

The *type* of a *vector* that is not displaced to another *array*, has no *fill pointer*, is not *expressly adjustable* and is able to hold elements of any *type* is a *subtype* of *type simple–vector*.

The *type simple–vector* is a *subtype* of *type vector*, and is a *subtype* of *type (vector t)*.

**Compound Type Specifier Kind:**

Specializing.

**Compound Type Specifier Syntax:**

**simple–vector [size]**

**Compound Type Specifier Arguments:**

*size*—a non-negative *fixnum*, or the *symbol* \*. The default is the *symbol* \*.

**Compound Type Specifier Description:**

This is the same as (*simple–array t (size)*).

**Type STANDARD–CHAR****Supertypes:**

**standard–char, base–char, character, t**

**Description:**

A fixed set of 96 *characters* required to be present in all *conforming implementations*. *Standard characters* are defined in [Section 2.1.3 \(Standard Characters\)](#).

Any *character* that is not *simple* is not a *standard character*.

**See Also:****System Class STANDARD–CLASS****Class Precedence List:**

**standard–class, class, standard–object, t**

**Description:**

The *class standard–class* is the default *class* of *classes* defined by *defclass*.

**System Class STANDARD-GENERIC-FUNCTION****Class Precedence List:****standard-generic-function, generic-function, function, t****Description:**

The class standard-generic-function is the default class of generic functions established by defmethod, ensure-generic-function, defgeneric, and defclass forms.

**System Class STANDARD-METHOD****Class Precedence List:****standard-method, method, standard-object, t****Description:****Class STANDARD-OBJECT****Class Precedence List:****standard-object, t****Description:**

The class standard-object is an instance of standard-class and is a superclass of every class that is an instance of standard-class except itself.

**System Class STRING-STREAM****Class Precedence List:****string-stream, stream, t****Description:**

A string stream is a stream which reads input from or writes output to an associated string.

The stream element type of a string stream is always a subtype of type character.

**See Also:**

**make-string-input-stream, make-string-output-stream, with-input-from-string, with-output-to-string**

**System Class STREAM****Class Precedence List:****stream\_t****Description:**

A stream is an object that can be used with an input or output function to identify an appropriate source or sink of characters or bytes for that operation.

For more complete information, see [Section 21.1 \(Stream Concepts\)](#).

**See Also:****System Class STRING****Class Precedence List:****string\_vector\_array\_sequence\_t****Description:**

A string is a specialized vector whose elements are of type character or a subtype of type character. When used as a type specifier for object creation, string means (vector character).

**Compound Type Specifier Kind:**

Abbreviating.

**Compound Type Specifier Syntax:****string [size]****Compound Type Specifier Arguments:**

size—a non-negative fixnum, or the symbol \*.

**Compound Type Specifier Description:**

This denotes the union of all types (array c (size)) for all subtypes c of character; that is, the set of strings of size size.

**See Also:****System Class STRUCTURE-CLASS****Class Precedence List:****structure-class\_class\_standard-object\_t**

**Description:**

All classes defined by means of defstruct are instances of the class structure-class.

## Class STRUCTURE-OBJECT

**Class Precedence List:**

### structure-object.t

**Description:**

The class structure-object is an instance of structure-class and is a superclass of every class that is an instance of structure-class except itself, and is a superclass of every class that is defined by defstruct.

**See Also:**

defstruct, Section 2.4.8.13 (Sharpsign S), Section 22.1.3.12 (Printing Structures)

## System Class SYMBOL

**Class Precedence List:**

### symbol.t

**Description:**

Symbols are used for their object identity to name various entities in Common Lisp, including (but not limited to) linguistic entities such as variables and functions.

Symbols can be collected together into packages. A symbol is said to be interned in a package if it is accessible in that package; the same symbol can be interned in more than one package. If a symbol is not interned in any package, it is called uninterned.

An interned symbol is uniquely identifiable by its name from any package in which it is accessible.

Symbols have the following attributes. For historical reasons, these are sometimes referred to as cells, although the actual internal representation of symbols and their attributes is implementation-dependent.

**Name**

The name of a symbol is a string used to identify the symbol. Every symbol has a name, and the consequences are undefined if that name is altered. The name is used as part of the external, printed representation of the symbol; see Section 2.1 (Character Syntax). The function symbol-name returns the name of a given symbol. A symbol may have any character in its name.

**Package**

The object in this cell is called the home package of the symbol. If the home package is nil, the symbol is sometimes said to have no home package.

When a symbol is first created, it has no home package. When it is first interned, the package in which it is initially interned becomes its home package. The home package of a symbol can be

accessed by using the [function symbol-package](#).

If a [symbol](#) is uninterned from the [package](#) which is its [home package](#), its [home package](#) is set to [nil](#). Depending on whether there is another [package](#) in which the [symbol](#) is interned, the symbol might or might not really be an [uninterned symbol](#). A [symbol](#) with no [home package](#) is therefore called [apparently uninterned](#).

The consequences are undefined if an attempt is made to alter the [home package](#) of a [symbol](#) external in the COMMON-LISP package or the KEYWORD package.

#### **Property list**

The [property list](#) of a [symbol](#) provides a mechanism for associating named attributes with that [symbol](#). The operations for adding and removing entries are destructive to the [property list](#). Common Lisp provides [operators](#) both for direct manipulation of [property list objects](#) (e.g., see [getf](#), [remf](#), and [symbol-plist](#)) and for implicit manipulation of a [symbol's property list](#) by reference to the [symbol](#) (e.g., see [get](#) and [remprop](#)). The [property list](#) associated with a [fresh symbol](#) is initially [null](#).

#### **Value**

If a symbol has a value attribute, it is said to be [bound](#), and that fact can be detected by the [function boundp](#). The [object](#) contained in the [value cell](#) of a [bound symbol](#) is the [value](#) of the [global variable](#) named by that [symbol](#), and can be accessed by the [function symbol-value](#). A [symbol](#) can be made to be [unbound](#) by the [function makunbound](#).

The consequences are undefined if an attempt is made to change the [value](#) of a [symbol](#) that names a [constant variable](#), or to make such a [symbol](#) be [unbound](#).

#### **Function**

If a symbol has a function attribute, it is said to be [fbound](#), and that fact can be detected by the [function fboundp](#). If the [symbol](#) is the [name](#) of a [function](#) in the [global environment](#), the [function cell](#) contains the [function](#), and can be accessed by the [function symbol-function](#). If the [symbol](#) is the [name](#) of either a [macro](#) in the [global environment](#) (see [macro-function](#)) or a [special operator](#) (see [special-operator-p](#)), the [symbol](#) is [fbound](#), and can be accessed by the [function symbol-function](#), but the [object](#) which the [function cell](#) contains is of [implementation-dependent type](#) and purpose. A [symbol](#) can be made to be [funbound](#) by the [function fmakunbound](#).

The consequences are undefined if an attempt is made to change the [functional value](#) of a [symbol](#) that names a [special form](#).

Operations on a [symbol's value cell](#) and [function cell](#) are sometimes described in terms of their effect on the [symbol](#) itself, but the user should keep in mind that there is an intimate relationship between the contents of those [cells](#) and the [global variable](#) or global [function](#) definition, respectively.

[Symbols](#) are used as identifiers for [lexical variables](#) and lexical [function](#) definitions, but in that role, only their [object](#) identity is significant. Common Lisp provides no operation on a [symbol](#) that can have any effect on a [lexical variable](#) or on a lexical [function](#) definition.

#### **See Also:**

## **System Class SYNONYM-STREAM**

#### **Class Precedence List:**

[synonym-stream](#), [stream](#), [t](#)

**Description:**

A stream that is an alias for another stream, which is the value of a dynamic variable whose name is the synonym stream symbol of the synonym stream.

Any operations on a synonym stream will be performed on the stream that is then the value of the dynamic variable named by the synonym stream symbol. If the value of the variable should change, or if the variable should be bound, then the stream will operate on the new value of the variable.

**See Also:**

[make-synonym-stream](#), [synonym-stream-symbol](#)

**System Class T****Class Precedence List:**

**t**

**Description:**

The set of all objects. The type t is a supertype of every type, including itself. Every object is of type t.

**System Class TWO-WAY-STREAM****Class Precedence List:**

[two-way-stream](#), [stream](#), [t](#)

**Description:**

A bidirectional composite stream that receives its input from an associated input stream and sends its output to an associated output stream.

**See Also:**

[make-two-way-stream](#), [two-way-stream-input-stream](#), [two-way-stream-output-stream](#)

**Type UNSIGNED-BYTE****Supertypes:**

[unsigned-byte](#), [signed-byte](#), [integer](#), [rational](#), [real](#), [number](#), [t](#)

**Description:**

The atomic type specifier unsigned-byte denotes the same type as is denoted by the type specifier (integer 0 \*).

**Compound Type Specifier Kind:**

Abbreviating.

### Compound Type Specifier Syntax:

**unsigned-byte** [*s* /\*]

### Compound Type Specifier Arguments:

*s*—a positive integer.

### Compound Type Specifier Description:

This denotes the set of non-negative integers that can be represented in a byte of size *s* (bits). This is equivalent to  $(\text{mod } m)$  for  $m=2^s$ , or to  $(\text{integer } 0 \ n)$  for  $n=2^s-1$ . The *type unsigned-byte* or the type *(unsigned-byte \* )* is the same as the type *(integer 0 \*)*, the set of non-negative integers.

### Notes:

The *type* (*unsigned-byte 1*) is also called bit.

## Type Specifier VALUES

### Compound Type Specifier Kind:

Specializing.

### Compound Type Specifier Syntax:

**values** *value-typespec*

*value-typespec*::= *typespec\** [&optional *typespec\**] [&rest *typespec*] [&allow-other-keys]

### Compound Type Specifier Arguments:

*typespec*—a type specifier.

### Compound Type Specifier Description:

This type specifier can be used only as the *value-type* in a function type specifier or a the special form. It is used to specify individual types when multiple values are involved. The &optional and &rest markers can appear in the *value-type* list; they indicate the parameter list of a function that, when given to multiple-value-call along with the values, would correctly receive those values.

The symbol \* may not be among the *value-types*.

The symbol values is not valid as a type specifier; and, specifically, it is not an abbreviation for *(values)*.

## System Class VECTOR

### Class Precedence List:

**vector, array, sequence, t****Description:**

Any one-dimensional array is a vector.

The type vector is a subtype of type array; for all types x, (vector x) is the same as (array x (\*)).

The type (vector t), the type string, and the type bit-vector are disjoint subtypes of type vector.

**Compound Type Specifier Kind:**

Specializing.

**Compound Type Specifier Syntax:**

**vector** [{element-type / \*} [{size / \*}]]

**Compound Type Specifier Arguments:**

size—a non-negative fixnum.

element-type—a type specifier.

**Compound Type Specifier Description:**

This denotes the set of specialized vectors whose element type and dimension match the specified values. Specifically:

If element-type is the symbol \*, vectors are not excluded on the basis of their element type. Otherwise, only those vectors are included whose actual array element type is the result of upgrading element-type; see Section 15.1.2.1 (Array Upgrading).

If a size is specified, the set includes only those vectors whose only dimension is size. If the symbol \* is specified instead of a size, the set is not restricted on the basis of dimension.

**See Also:****Notes:**

The type (vector e s) is equivalent to the type (array e (s)).

The type (vector bit) has the name bit-vector.

The union of all types (vector C), where C is any subtype of character, has the name string.

(vector \*) refers to all vectors regardless of element type, (vector type-specifier) refers only to those vectors that can result from giving type-specifier as the :element-type argument to make-array.

**Variable –****Value Type:**a *form*.**Initial Value:***implementation-dependent*.**Description:**The *value* of \_ is the *form* that is currently being evaluated by the *Lisp read–eval–print loop*.**Examples:**

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

**Affected By:***Lisp read–eval–print loop*.**See Also:** $\pm$  (*variable*),  $*$  (*variable*),  $/_$  (*variable*), Section 25.1.1 (Top level loop)**Notes:** None.**Variable \*, \*\*, \*\*\*****Value Type:**an *object*.**Initial Value:***implementation-dependent*.**Description:**The *variables* \*, \*\*, and \*\*\* are maintained by the *Lisp read–eval–print loop* to save the values of results that are printed each time through the loop.The *value* of \* is the most recent *primary value* that was printed, the *value* of \*\* is the previous value of \*, and the *value* of \*\*\* is the previous value of \*\*.If several values are produced, \* contains the first value only; \* contains nil if zero values are produced.

## CLHS: Declaration DYNAMIC-EXTENT

The values of \*, \*\*, and \*\*\* are updated immediately prior to printing the return value of a top-level form by the Lisp read–eval–print loop. If the evaluation of such a form is aborted prior to its normal return, the values of \*, \*\*, and \*\*\* are not updated.

### Examples:

```
(values 'a1 'a2) => A1, A2
'b => B
(values 'c1 'c2 'c3) => C1, C2, C3
(list * ** *** ) => (C1 B A1)

(defun cube-root (x) (expt x 1/3)) => CUBE-ROOT
(compile *) => CUBE-ROOT
(setq a (cube-root 27.0)) => 3.0
(* * 9.0) => 27.0
```

### Affected By:

Lisp read–eval–print loop.

### See Also:

= (variable), + (variable), / (variable), Section 25.1.1 (Top level loop)

### Notes:

```
*    == (car /)
**   == (car //)
***  == (car ///)
```

## Constant Variable ARRAY–DIMENSION–LIMIT

### Constant Value:

A positive fixnum, the exact magnitude of which is implementation–dependent, but which is not less than 1024.

### Description:

The upper exclusive bound on each individual dimension of an array.

### Examples:

None.

### See Also:

make–array

### Notes:

None.

## Constant Variable ARRAY–RANK–LIMIT

### Constant Value:

## CLHS: Declaration DYNAMIC-EXTENT

A positive *fixnum*, the exact magnitude of which is *implementation-dependent*, but which is not less than 8.

### Description:

The upper exclusive bound on the *rank* of an *array*.

**Examples:** None.

### See Also:

[make-array](#)

**Notes:** None.

## **Constant Variable ARRAY-TOTAL-SIZE-LIMIT**

### Constant Value:

A positive *fixnum*, the exact magnitude of which is *implementation-dependent*, but which is not less than 1024.

### Description:

The upper exclusive bound on the *array total size* of an *array*.

The actual limit on the *array total size* imposed by the *implementation* might vary according the *element type* of the *array*; in this case, the value of **array-total-size-limit** will be the smallest of these possible limits.

**Examples:** None.

### See Also:

[make-array](#), [array-element-type](#)

**Notes:** None.

**Constant Variable BOOLE-1, BOOLE-2, BOOLE-AND, BOOLE-ANDC1, BOOLE-ANDC2, BOOLE-C1, BOOLE-C2, BOOLE-CLR, BOOLE-EQV, BOOLE-IOR, BOOLE-NAND, BOOLE-NOR, BOOLE-ORC1, BOOLE-ORC2, BOOLE-SET, BOOLE-XOR**

**Constant Value:**

The identity and nature of the *values* of each of these *variables* is *implementation-dependent*, except that it must be *distinct* from each of the *values* of the others, and it must be a valid first *argument* to the *function boole*.

**Description:**

Each of these *constants* has a *value* which is one of the sixteen possible *bit-wise logical operation specifiers*.

**Examples:**

```
(boole boole-ior 1 16) => 17
(boole boole-and -2 5) => 4
(boole boole-eqv 17 15) => -31
```

**See Also:**

**boole**

**Notes:** None.

**Variable \*BREAK-ON-SIGNALS\***

**Value Type:**

a *type specifier*.

**Initial Value:**

**nil**.

**Description:**

When (*typep condition \*break-on-signals\**) returns *true*, calls to *signal*, and to other *operators* such as *error* that implicitly call *signal*, enter the debugger prior to *signaling the condition*.

The *continue restart* can be used to continue with the normal *signaling* process when a break occurs process due to **\*break-on-signals\***.

**Examples:**

```
*break-on-signals* => NIL
(ignore-errors (error 'simple-error :format-control "Fooey!"))
=> NIL, #<SIMPLE-ERROR 32207172>
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(let ((*break-on-signals* 'error))
  (ignore-errors (error 'simple-error :format-control "Fooey!")))
>> Break: Fooey!
>> BREAK entered because of *BREAK-ON-SIGNALS*.
>> To continue, type :CONTINUE followed by an option number:
>> 1: Continue to signal.
>> 2: Top level.
>> Debug> :CONTINUE 1
>> Continue to signal.
=> NIL, #<SIMPLE-ERROR 32212257>

(let ((*break-on-signals* 'error))
  (error 'simple-error :format-control "Fooey!"))
>> Break: Fooey!
>> BREAK entered because of *BREAK-ON-SIGNALS*.
>> To continue, type :CONTINUE followed by an option number:
>> 1: Continue to signal.
>> 2: Top level.
>> Debug> :CONTINUE 1
>> Continue to signal.
>> Error: Fooey!
>> To continue, type :CONTINUE followed by an option number:
>> 1: Top level.
>> Debug> :CONTINUE 1
>> Top level.
```

**Affected By:** None.

**See Also:**

[\*\*break\*\*](#), [\*\*signal\*\*](#), [\*\*warn\*\*](#), [\*\*error\*\*](#), [\*\*typep\*\*](#), Section 9.1 (Condition System Concepts)

**Notes:**

**\*break-on-signals\*** is intended primarily for use in debugging code that does signaling. When setting **\*break-on-signals\***, the user is encouraged to choose the most restrictive specification that suffices. Setting **\*break-on-signals\*** effectively violates the modular handling of [\*condition\*](#) signaling. In practice, the complete effect of setting **\*break-on-signals\*** might be unpredictable in some cases since the user might not be aware of the variety or number of calls to [\*\*signal\*\*](#) that are used in code called only incidentally.

**\*break-on-signals\*** enables an early entry to the debugger but such an entry does not preclude an additional entry to the debugger in the case of operations such as [\*\*error\*\*](#) and [\*\*cerror\*\*](#).

## **Constant Variable CALL-ARGUMENTS-LIMIT**

**Constant Value:**

An integer not smaller than 50 and at least as great as the [\*value\*](#) of [\*\*lambda-parameters-limit\*\*](#), the exact magnitude of which is [\*implementation-dependent\*](#).

**Description:**

The upper exclusive bound on the number of [\*arguments\*](#) that may be passed to a [\*function\*](#).

**Examples:** None.

**See Also:**

[lambda-parameters-limit](#), [multiple-values-limit](#)

**Notes:** None.

**Constant Variable CHAR-CODE-LIMIT****Constant Value:**

A non-negative integer, the exact magnitude of which is implementation-dependent, but which is not less than 96 (the number of standard characters).

**Description:**

The upper exclusive bound on the value returned by the function char-code.

**See Also:**

[char-code](#)

**Notes:**

The value of char-code-limit might be larger than the actual number of characters supported by the implementation.

**Variable \*COMPILE-FILE-PATHNAME\*, \*COMPILE-FILE-TRUENAME\*****Value Type:**

The value of \*compile-file-pathname\* must always be a pathname or nil. The value of \*compile-file-trueename\* must always be a physical pathname or nil.

**Initial Value:**

nil.

**Description:**

During a call to compile-file, \*compile-file-pathname\* is bound to the pathname denoted by the first argument to compile-file, merged against the defaults; that is, it is bound to (pathname (merge-pathnames input-file)). During the same time interval, \*compile-file-trueename\* is bound to the trueename of the file being compiled.

At other times, the value of these variables is nil.

If a break loop is entered while compile-file is ongoing, it is implementation-dependent whether these variables retain the values they had just prior to entering the break loop or whether they are bound to nil.

## CLHS: Declaration DYNAMIC-EXTENT

The consequences are unspecified if an attempt is made to *assign* or *bind* either of these *variables*.

**Examples:** None.

**Affected By:**

The *file system*.

**See Also:**

[compile-file](#)

**Notes:** None.

## **Variable \*COMPILE-PRINT\*, \*COMPILE-VERBOSE\***

**Value Type:**

a *generalized boolean*.

**Initial Value:**

*implementation-dependent*.

**Description:**

The *value* of **\*compile-print\*** is the default value of the :print *argument* to [compile-file](#). The *value* of **\*compile-verbose\*** is the default value of the :verbose *argument* to [compile-file](#).

**Examples:** None.

**Affected By:** None.

**See Also:**

[compile-file](#)

**Notes:** None.

## **Variable \*DEBUG-IO\*, \*ERROR-OUTPUT\*, \*QUERY-IO\*, \*STANDARD-INPUT\*, \*STANDARD-OUTPUT\*, \*TRACE-OUTPUT\***

**Value Type:**

For **\*standard-input\***: an *input stream*

For **\*error-output\***, **\*standard-output\***, and **\*trace-output\***: an *output stream*.

For **\*debug-io\***, **\*query-io\***: a *bidirectional stream*.

#### Initial Value:

*implementation-dependent*, but it must be an *open stream* that is not a *generalized synonym stream* to an *I/O customization variables* but that might be a *generalized synonym stream* to the value of some *I/O customization variable*. The initial value might also be a *generalized synonym stream* to either the *symbol \*terminal-io\** or to the *stream* that is its *value*.

#### Description:

These *variables* are collectively called the *standardized I/O customization variables*. They can be *bound* or *assigned* in order to change the default destinations for input and/or output used by various *standardized operators* and facilities.

The *value* of **\*debug-io\***, called *debug I/O*, is a *stream* to be used for interactive debugging purposes.

The *value* of **\*error-output\***, called *error output*, is a *stream* to which warnings and non-interactive error messages should be sent.

The *value* of **\*query-io\***, called *query I/O*, is a *bidirectional stream* to be used when asking questions of the user. The question should be output to this *stream*, and the answer read from it.

The *value* of **\*standard-input\***, called *standard input*, is a *stream* that is used by many *operators* as a default source of input when no specific *input stream* is explicitly supplied.

The *value* of **\*standard-output\***, called *standard output*, is a *stream* that is used by many *operators* as a default destination for output when no specific *output stream* is explicitly supplied.

The *value* of **\*trace-output\***, called *trace output*, is the *stream* on which traced functions (see **trace**) and the **time macro** print their output.

#### Examples:

```
(with-output-to-string (*error-output*)
  (warn "this string is sent to *error-output*"))
=> "Warning: this string is sent to *error-output*
" ;The exact format of this string is implementation-dependent.

(with-input-from-string (*standard-input* "1001")
  (+ 990 (read))) => 1991

(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")))
       :done)
=> :DONE
out
=>
\"print and format t send things to\" *standard-output* now going to a string"
```

```
(defun fact (n) (if (< n 2) 1 (* n (fact (- n 1)))))
=> FACT
(trace fact)
=> (FACT)
;; Of course, the format of traced output is implementation-dependent.
(with-output-to-string (*trace-output*)
  (fact 3))
=> "
1 Enter FACT 3
| 2 Enter FACT 2
| | 3 Enter FACT 1
| | 3 Exit FACT 1
| 2 Exit FACT 2
1 Exit FACT 6"
```

**See Also:**

[\\*terminal-io\\*](#), [synonym-stream](#), [time](#), [trace](#), [Section 9 \(Conditions\)](#), [Section 23 \(Reader\)](#), [Section 22 \(Printer\)](#)

**Notes:**

The intent of the constraints on the initial *value* of the *I/O customization variables* is to ensure that it is always safe to *bind* or *assign* such a *variable* to the *value* of another *I/O customization variable*, without unduly restricting *implementation* flexibility.

It is common for an *implementation* to make the initial *values* of [\\*debug-io\\*](#) and [\\*query-io\\*](#) be the *same stream*, and to make the initial *values* of [\\*error-output\\*](#) and [\\*standard-output\\*](#) be the *same stream*.

The functions [y-or-n-p](#) and [yes-or-no-p](#) use *query I/O* for their input and output.

In the normal *Lisp read-eval-print loop*, input is read from *standard input*. Many input functions, including [read](#) and [read-char](#), take a *stream* argument that defaults to *standard input*.

In the normal *Lisp read-eval-print loop*, output is sent to *standard output*. Many output functions, including [print](#) and [write-char](#), take a *stream* argument that defaults to *standard output*.

A program that wants, for example, to divert output to a file should do so by *binding* [\\*standard-output\\*](#); that way error messages sent to [\\*error-output\\*](#) can still get to the user by going through [\\*terminal-io\\*](#) (if [\\*error-output\\*](#) is bound to [\\*terminal-io\\*](#)), which is usually what is desired.

**Variable \*DEBUGGER-HOOK\*****Value Type:**

a *designator* for a *function* of two *arguments* (a *condition* and the *value* of [\\*debugger-hook\\*](#) at the time the debugger was entered), or [nil](#).

**Initial Value:**

[nil](#).

**Description:**

When the value of **\*debugger-hook\*** is non-nil, it is called prior to normal entry into the debugger, either due to a call to **invoke-debugger** or due to automatic entry into the debugger from a call to **error** or **cerror** with a condition that is not handled. The function may either handle the condition (transfer control) or return normally (allowing the standard debugger to run). To minimize recursive errors while debugging, **\*debugger-hook\*** is bound to **nil** by **invoke-debugger** prior to calling the function.

**Examples:**

```
(defun one-of (choices &optional (prompt "Choice"))
  (let ((n (length choices)) (i))
    (do ((c choices (cdr c)) (i 1 (+ i 1)))
        ((null c))
      (format t "~&[~D] ~A~%" i (car c)))
    (do () ((typep i `(integer 1 ,n)))
      (format t "~&~A: " prompt)
      (setq i (read))
      (fresh-line))
    (nth (- i 1) choices)))

(defun my-debugger (condition me-or-my-encapsulation)
  (format t "~&Fooey: ~A" condition)
  (let ((restart (one-of (compute-restarts))))
    (if (not restart) (error "My debugger got an error."))
    (let ((*debugger-hook* me-or-my-encapsulation))
      (invoke-restart-interactively restart)))

  (let ((*debugger-hook* #'my-debugger))
    (+ 3 'a))
>> Fooey: The argument to +, A, is not a number.
>> [1] Supply a replacement for A.
>> [2] Return to Cloe Toplevel.
>> Choice: 1
>> Form to evaluate and use: (+ 5 'b)
>> Fooey: The argument to +, B, is not a number.
>> [1] Supply a replacement for B.
>> [2] Supply a replacement for A.
>> [3] Return to Cloe Toplevel.
>> Choice: 1
>> Form to evaluate and use: 1
=> 9
```

**Affected By:****invoke-debugger**

**See Also:** None.

**Notes:**

When evaluating code typed in by the user interactively, it is sometimes useful to have the hook function bind **\*debugger-hook\*** to the function that was its second argument so that recursive errors can be handled using the same interactive facility.

**Variable \*DEFAULT-PATHNAME-DEFAULTS\*****Value Type:**

a *pathname object*.

**Initial Value:**

An *implementation-dependent pathname*, typically in the working directory that was current when Common Lisp was started up.

**Description:**

a *pathname*, used as the default whenever a *function* needs a default *pathname* and one is not supplied.

**Examples:**

```
; ; This example illustrates a possible usage for a hypothetical Lisp running on a
; ; DEC TOPS-20 file system. Since pathname conventions vary between Lisp
; ; implementations and host file system types, it is not possible to provide a
; ; general-purpose, conforming example.
*default-pathname-defaults* => #P"PS:<FRED>"
(merge-pathnames (make-pathname :name "CALENDAR" ))
=> #P"PS:<FRED>CALENDAR"
(let ((*default-pathname-defaults* (pathname "<MARY>" )))
  (merge-pathnames (make-pathname :name "CALENDAR" )))
=> #P"<MARY>CALENDAR"
```

**Affected By:**

The *implementation*.

**See Also:** None.

**Notes:** None.

**Variable \*FEATURES\*****Value Type:**

a *proper list*.

**Initial Value:**

*implementation-dependent*.

**Description:**

The *value* of **\*features\*** is called the *features list*. It is a *list of symbols*, called *features*, that correspond to some aspect of the *implementation* or *environment*.

## CLHS: Declaration DYNAMIC-EXTENT

Most *features* have *implementation-dependent* meanings; The following meanings have been assigned to feature names:

:clt11

If present, indicates that the LISP package *purports to conform* to the 1984 specification *Common Lisp: The Language*. It is possible, but not required, for a *conforming implementation* to have this feature because this specification specifies that its *symbols* are to be in the COMMON-LISP package, not the LISP package.

:clt12

If present, indicates that the implementation *purports to conform* to *Common Lisp: The Language, Second Edition*. This feature must not be present in any *conforming implementation*, since conformance to that document is not compatible with conformance to this specification. The name, however, is reserved by this specification in order to help programs distinguish implementations which conform to that document from implementations which conform to this specification.

:ieee-floating-point

If present, indicates that the implementation *purports to conform* to the requirements of *IEEE Standard for Binary Floating-Point Arithmetic*.

:x3j13

If present, indicates that the implementation conforms to some particular working draft of this specification, or to some subset of features that approximates a belief about what this specification might turn out to contain. A *conforming implementation* might or might not contain such a feature. (This feature is intended primarily as a stopgap in order to provide implementors something to use prior to the availability of a draft standard, in order to discourage them from introducing the :draft-ansi-cl and :ansi-cl *features* prematurely.)

:draft-ansi-cl

If present, indicates that the *implementation purports to conform* to the first full draft of this specification, which went to public review in 1992. A *conforming implementation* which has the :draft-ansi-cl-2 or :ansi-cl *feature* is not permitted to retain the :draft-ansi-cl *feature* since incompatible changes were made subsequent to the first draft.

:draft-ansi-cl-2

If present, indicates that a second full draft of this specification has gone to public review, and that the *implementation purports to conform* to that specification. (If additional public review drafts are produced, this keyword will continue to refer to the second draft, and additional keywords will be added to identify conformance with such later drafts. As such, the meaning of this keyword can be relied upon not to change over time.) A *conforming implementation* which has the :ansi-cl *feature* is only permitted to retain the :draft-ansi-cl *feature* if the finally approved standard is not incompatible with the draft standard.

:ansi-cl

If present, indicates that this specification has been adopted by ANSI as an official standard, and that the *implementation purports to conform*.

:common-lisp

This feature must appear in \***features**\* for any implementation that has one or more of the features :x3j13, :draft-ansi-cl, or :ansi-cl. It is intended that it should also appear in implementations which have the features :clt11 or :clt12, but this specification cannot force such behavior. The intent is that this feature should identify the language family named "Common Lisp," rather than some specific dialect within that family.

**Examples:** None.

**Affected By:** None.

**See Also:****Notes:**

The value of **\*features\*** is used by the #+ and #- reader syntax.

Symbols in the features list may be in any package, but in practice they are generally in the KEYWORD package. This is because KEYWORD is the package used by default when reading[2] feature expressions in the #+ and #- reader macros. Code that needs to name a feature[2] in a package P (other than KEYWORD) can do so by making explicit use of a package prefix for P, but note that such code must also assure that the package P exists in order for the feature expression to be read[2]—even in cases where the feature expression is expected to fail.

It is generally considered wise for an implementation to include one or more features identifying the specific implementation, so that conditional expressions can be written which distinguish idiosyncrasies of one implementation from those of another. Since features are normally symbols in the KEYWORD package where name collisions might easily result, and since no uniquely defined mechanism is designated for deciding who has the right to use which symbol for what reason, a conservative strategy is to prefer names derived from one's own company or product name, since those names are often trademarked and are hence less likely to be used unwittingly by another implementation.

## Variable \*GENSYM-COUNTER\*

**Value Type:**

a non-negative integer.

**Initial Value:**

implementation-dependent.

**Description:**

A number which will be used in constructing the name of the next symbol generated by the function gensym.

**\*gensym-counter\*** can be either assigned or bound at any time, but its value must always be a non-negative integer.

**Examples:** None.**Affected By:**

gensym.

**See Also:**

gensym

**Notes:**

**Constant Variable INTERNAL-TIME-UNITS-PER-SECOND****Constant Value:**

A positive *integer*, the magnitude of which is *implementation-dependent*.

**Description:**

The number of *internal time units* in one second.

**Examples:** None.

**See Also:**

[get-internal-run-time](#), [get-internal-real-time](#)

**Notes:**

These units form the basis of the Internal Time format representation.

**Constant Variable LAMBDA-PARAMETERS-LIMIT****Constant Value:**

*implementation-dependent*, but not smaller than 50.

**Description:**

A positive *integer* that is the upper exclusive bound on the number of *parameter names* that can appear in a single *lambda list*.

**Examples:** None.

**See Also:**

[call-arguments-limit](#)

**Notes:**

Implementors are encouraged to make the *value* of **lambda-parameters-limit** as large as possible.

**Constant Variable LAMBDA-LIST-KEYWORDS****Constant Value:**

a *list*, the *elements* of which are *implementation-dependent*, but which must contain at least the *symbols* &allow-other-keys, &aux, &body, &environment, &key, &optional, &rest, and &whole.

**Description:**

## CLHS: Declaration DYNAMIC-EXTENT

A list of all the lambda list keywords used in the implementation, including the additional ones used only by macro definition forms.

**Examples:** None.

**See Also:**

defun, flet, defmacro, macrolet, Section 3.1.2 (The Evaluation Model)

**Notes:** None.

## Variable \*LOAD-PATHNAME\*, \*LOAD-TRUENAME\*

**Value Type:**

The value of \*load-pathname\* must always be a pathname or nil. The value of \*load-truename\* must always be a physical pathname or nil.

**Initial Value:**

nil.

**Description:**

During a call to load, \*load-pathname\* is bound to the pathname denoted by the the first argument to load, merged against the defaults; that is, it is bound to (pathname (merge-pathnames filespec)). During the same time interval, \*load-truename\* is bound to the truename of the file being loaded.

At other times, the value of these variables is nil.

If a break loop is entered while load is ongoing, it is implementation-dependent whether these variables retain the values they had just prior to entering the break loop or whether they are bound to nil.

The consequences are unspecified if an attempt is made to assign or bind either of these variables.

**Examples:** None.

**Affected By:**

The file system.

**See Also:**

load

**Notes:** None.

**Variable \*LOAD-PRINT\*, \*LOAD-VERBOSE\*****Value Type:**

a *generalized boolean*.

**Initial Value:**

The initial *value* of **\*load-print\*** is *false*. The initial *value* of **\*load-verbose\*** is *implementation-dependent*.

**Description:**

The *value* of **\*load-print\*** is the default value of the :print *argument* to **load**. The *value* of **\*load-verbose\*** is the default value of the :verbose *argument* to **load**.

**Examples:** None.

**Affected By:** None.

**See Also:**

**load**

**Notes:** None.

**Variable \*MACROEXPAND-HOOK\*****Value Type:**

a *designator* for a *function* of three *arguments*: a *macro function*, a *macro form*, and an *environment object*.

**Initial Value:**

a *designator* for a function that is equivalent to the *function funcall*, but that might have additional *implementation-dependent* side-effects.

**Description:**

Used as the expansion interface hook by **macroexpand-1** to control the *macro expansion* process. When a *macro form* is to be expanded, this *function* is called with three arguments: the *macro function*, the *macro form*, and the *environment* in which the *macro form* is to be expanded. The *environment object* has *dynamic extent*; the consequences are undefined if the *environment object* is referred to outside the *dynamic extent* of the macro expansion function.

**Examples:**

```
(defun hook (expander form env)
  (format t "Now expanding: ~S~%" form)
  (funcall expander form env)) => HOOK
(defmacro machook (x y) `(/ (+ ,x ,y) 2)) => MACHOOK
(macroexpand '(machook 1 2)) => (/ (+ 1 2) 2), true
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(let ((*macroexpand-hook* #'hook)) (macroexpand '(machook 1 2)))
>> Now expanding (MACHOOK 1 2)
=> (/ (+ 1 2) 2), true
```

**Affected By:** None.

**See Also:**

[macroexpand](#), [macroexpand-1](#), [funcall](#), Section 3.1 (Evaluation)

**Notes:**

The net effect of the chosen initial value is to just invoke the *macro function*, giving it the *macro form* and *environment* as its two arguments.

Users or user programs can *assign* this *variable* to customize or trace the *macro expansion* mechanism. Note, however, that this *variable* is a global resource, potentially shared by multiple *programs*; as such, if any two *programs* depend for their correctness on the setting of this *variable*, those *programs* may not be able to run in the same *Lisp image*. For this reason, it is frequently best to confine its uses to debugging situations.

Users who put their own function into **\*macroexpand-hook\*** should consider saving the previous value of the hook, and calling that value from their own.

## Variable \*MODULES\*

**Value Type:**

a *list* of *strings*.

**Initial Value:**

*implementation-dependent*.

**Description:**

The *value* of **\*modules\*** is a list of names of the modules that have been loaded into the current *Lisp image*.

**Examples:** None.

**Affected By:**

[provide](#)

**See Also:**

[provide](#), [require](#)

**Notes:**

The variable **\*modules\*** is deprecated.

**Constant Variable MOST-POSITIVE-SHORT-FLOAT,  
LEAST-POSITIVE-SHORT-FLOAT,  
LEAST-POSITIVE-NORMALIZED-SHORT-FLOAT,  
MOST-POSITIVE-DOUBLE-FLOAT, LEAST-POSITIVE-DOUBLE-FLOAT,  
LEAST-POSITIVE-NORMALIZED-DOUBLE-FLOAT,  
MOST-POSITIVE-LONG-FLOAT, LEAST-POSITIVE-LONG-FLOAT,  
LEAST-POSITIVE-NORMALIZED-LONG-FLOAT,  
MOST-POSITIVE-SINGLE-FLOAT, LEAST-POSITIVE-SINGLE-FLOAT,  
LEAST-POSITIVE-NORMALIZED-SINGLE-FLOAT,  
MOST-NEGATIVE-SHORT-FLOAT, LEAST-NEGATIVE-SHORT-FLOAT,  
LEAST-NEGATIVE-NORMALIZED-SHORT-FLOAT,  
MOST-NEGATIVE-SINGLE-FLOAT, LEAST-NEGATIVE-SINGLE-FLOAT,  
LEAST-NEGATIVE-NORMALIZED-SINGLE-FLOAT,  
MOST-NEGATIVE-DOUBLE-FLOAT, LEAST-NEGATIVE-DOUBLE-FLOAT,  
LEAST-NEGATIVE-NORMALIZED-DOUBLE-FLOAT,  
MOST-NEGATIVE-LONG-FLOAT, LEAST-NEGATIVE-LONG-FLOAT,  
LEAST-NEGATIVE-NORMALIZED-LONG-FLOAT**

**Constant Value:**

*implementation-dependent.*

**Description:**

These *constant variables* provide a way for programs to examine the *implementation-defined* limits for the various float formats.

Of these *variables*, each which has "-normalized" in its *name* must have a *value* which is a *normalized float*, and each which does not have "-normalized" in its name may have a *value* which is either a *normalized float* or a *denormalized float*, as appropriate.

Of these *variables*, each which has "short-float" in its name must have a *value* which is a *short float*, each which has "single-float" in its name must have a *value* which is a *single float*, each which has "double-float" in its name must have a *value* which is a *double float*, and each which has "long-float" in its name must have a *value* which is a *long float*.

- **most-positive-short-float, most-positive-single-float, most-positive-double-float,  
most-positive-long-float**

Each of these *constant variables* has as its *value* the positive *float* of the largest magnitude (closest in value to, but not equal to, positive infinity) for the float format implied by its name.

- **least-positive-short-float, least-positive-normalized-short-float, least-positive-single-float,  
least-positive-normalized-single-float, least-positive-double-float,  
least-positive-normalized-double-float, least-positive-long-float,  
least-positive-normalized-long-float**

Each of these *constant variables* has as its *value* the smallest positive (nonzero) *float* for the float

format implied by its name.

- least-negative-short-float, least-negative-normalized-short-float,  
least-negative-single-float, least-negative-normalized-single-float,  
least-negative-double-float, least-negative-normalized-double-float,  
least-negative-long-float, least-negative-normalized-long-float

Each of these constant variables has as its value the negative (nonzero) float of the smallest magnitude for the float format implied by its name. (If an implementation supports minus zero as a different object from positive zero, this value must not be minus zero.)

- most-negative-short-float, most-negative-single-float, most-negative-double-float,  
most-negative-long-float

Each of these constant variables has as its value the negative float of the largest magnitude (closest in value to, but not equal to, negative infinity) for the float format implied by its name.

**Examples:** None.

**See Also:** None.

**Notes:**

## **Constant Variable MOST-POSITIVE-FIXNUM, MOST-NEGATIVE-FIXNUM**

**Constant Value:**

implementation-dependent.

**Description:**

most-positive-fixnum is that fixnum closest in value to positive infinity provided by the implementation, and greater than or equal to both  $2^{15} - 1$  and array-dimension-limit.

most-negative-fixnum is that fixnum closest in value to negative infinity provided by the implementation, and less than or equal to  $-2^{15}$ .

**Examples:** None.

**See Also:** None.

**Notes:** None.

## **Constant Variable MULTIPLE-VALUES-LIMIT**

**Constant Value:**

An integer not smaller than 20, the exact magnitude of which is implementation-dependent.

**Description:**

## CLHS: Declaration DYNAMIC-EXTENT

The upper exclusive bound on the number of *values* that may be returned from a *function*, bound or assigned by *multiple-value-bind* or *multiple-value-setq*, or passed as a first argument to *nth-value*. (If these individual limits might differ, the minimum value is used.)

**Examples:** None.

**See Also:**

[lambda-parameters-limit](#), [call-arguments-limit](#)

**Notes:**

Implementors are encouraged to make this limit as large as possible.

## Constant Variable NIL

**Constant Value:**

nil.

**Description:**

nil represents both *boolean* (and *generalized boolean*) *false* and the *empty list*.

**Examples:**

```
nil => NIL
```

**See Also:**

t

**Notes:** None.

## Constant Variable PI

**Value:**

an *implementation-dependent long float*.

**Description:**

The best *long float* approximation to the mathematical constant <PI>.

**Examples:**

```
; In each of the following computations, the precision depends
; on the implementation. Also, if 'long float' is treated by
; the implementation as equivalent to some other float format
; (e.g., 'double float') the exponent marker might be the marker
; for that equivalent (e.g., 'D' instead of 'L').
pi => 3.141592653589793L0
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(cos pi) => -1.0L0

(defun sin-of-degrees (degrees)
  (let ((x (if (floatp degrees) degrees (float degrees pi))))
    (sin (* x (/ (float pi x) 180)))))
```

**See Also:** None.

### Notes:

An approximation to <PI> in some other precision can be obtained by writing (`float pi x`), where `x` is a *float* of the desired precision, or by writing (`coerce pi type`), where `type` is the desired type, such as `short-float`.

## Variable \*PACKAGE\*

### Value Type:

a *package object*.

### Initial Value:

the COMMON-LISP-USER package.

### Description:

Whatever *package object* is currently the *value* of **\*package\*** is referred to as the *current package*.

### Examples:

```
(in-package "COMMON-LISP-USER") => #<PACKAGE "COMMON-LISP-USER">
*package* => #<PACKAGE "COMMON-LISP-USER">
(make-package "SAMPLE-PACKAGE" :use '("COMMON-LISP"))
=> #<PACKAGE "SAMPLE-PACKAGE">
(list
  (symbol-package
    (let ((*package* (find-package 'sample-package)))
      (setq *some-symbol* (read-from-string "just-testing"))))
  *package*)
=> (#<PACKAGE "SAMPLE-PACKAGE"> #<PACKAGE "COMMON-LISP-USER">)
(list (symbol-package (read-from-string "just-testing"))
  *package*)
=> (#<PACKAGE "COMMON-LISP-USER"> #<PACKAGE "COMMON-LISP-USER">)
(eq 'foo (intern "FOO")) => true
(eq 'foo (let ((*package* (find-package 'sample-package)))
           (intern "FOO")))
=> false
```

### Affected By:

**load, compile-file, in-package**

### See Also:

Variable \*PACKAGE\*

742

**compile-file, in-package, load, package****Notes:** None.**Variable +, ++, +++****Value Type:**an *object*.**Initial Value:***implementation-dependent*.**Description:**

The *variables* `+`, `++`, and `+++` are maintained by the *Lisp read–eval–print loop* to save *forms* that were recently *evaluated*.

The *value* of `+` is the last *form* that was *evaluated*, the *value* of `++` is the previous value of `+`, and the *value* of `+++` is the previous value of `++`.

**Examples:**

```
(+ 0 1) => 1
(- 4 2) => 2
(/ 9 3) => 3
(list + ++ +++) => ((/ 9 3) (- 4 2) (+ 0 1))
(setq a 1 b 2 c 3 d (list a b c)) => (1 2 3)
(setq a 4 b 5 c 6 d (list a b c)) => (4 5 6)
(list a b c) => (4 5 6)
(eval +++) => (1 2 3)
#.`(,@++ d) => (1 2 3 (1 2 3))
```

**Affected By:***Lisp read–eval–print loop*.**See Also:**`=` (*variable*), `*` (*variable*), `/` (*variable*), Section 25.1.1 (Top level loop)**Notes:** None.**Variable \*PRINT-ARRAY\*****Value Type:**a *generalized boolean*.

**Initial Value:**

*implementation-dependent.*

**Description:**

Controls the format in which *arrays* are printed. If it is *false*, the contents of *arrays* other than *strings* are never printed. Instead, *arrays* are printed in a concise form using #< that gives enough information for the user to be able to identify the *array*, but does not include the entire *array* contents. If it is *true*, non-*string arrays* are printed using #( . . . ), #\*, or #nA syntax.

**Examples:** None.

**Affected By:**

The *implementation*.

**See Also:**

**Notes:** None.

**Variable \*PRINT-BASE\*, \*PRINT-RADIX\*****Value Type:**

\*print-base\*---a *radix*. \*print-radix\*---a *generalized boolean*.

**Initial Value:**

The initial *value* of \*print-base\* is 10. The initial *value* of \*print-radix\* is *false*.

**Description:**

\*print-base\* and \*print-radix\* control the printing of *rationals*. The *value* of \*print-base\* is called the *current output base*.

The *value* of \*print-base\* is the *radix* in which the printer will print *rationals*. For radices above 10, letters of the alphabet are used to represent digits above 9.

If the *value* of \*print-radix\* is *true*, the printer will print a radix specifier to indicate the *radix* in which it is printing a *rational* number. The radix specifier is always printed using lowercase letters. If \*print-base\* is 2, 8, or 16, then the radix specifier used is #b, #o, or #x, respectively. For *integers*, base ten is indicated by a trailing decimal point instead of a leading radix specifier; for *ratios*, #10r is used.

**Examples:**

```
(let ((*print-base* 24.) (*print-radix* t))
  (print 23.))
>> #24rN
=> 23
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(setq *print-base* 10) => 10
(setq *print-radix* nil) => NIL
(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
  (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)))
  >> #b1010  #b1/1010
  >> #3r101  #3r1/101
  >> #o12   #o1/12
  >> 10.   #10r1/10
  >> #xA   #x1/A
=> NIL
```

### Affected By:

Might be bound by format, and write, write-to-string.

### See Also:

format, write, write-to-string

Notes: None.

## Variable \*PRINT-CASE\*

### Value Type:

One of the symbols :upcase, :downcase, or :capitalize.

### Initial Value:

The symbol :upcase.

### Description:

The value of \*print-case\* controls the case (upper, lower, or mixed) in which to print any uppercase characters in the names of symbols when vertical-bar syntax is not used.

\*print-case\* has an effect at all times when the value of \*print-escape\* is false. \*print-case\* also has an effect when the value of \*print-escape\* is true unless inside an escape context (i.e., unless between vertical-bars or after a slash).

### Examples:

## CLHS: Declaration DYNAMIC-EXTENT

```
(defun test-print-case ()
  (dolist (*print-case* '(:upcase :downcase :capitalize))
    (format t "~&~S ~S~%" 'this-and-that '|And-something-else|)))
=> TEST-PC
;; Although the choice of which characters to escape is specified by
;; *PRINT-CASE*, the choice of how to escape those characters
;; (i.e., whether single escapes or multiple escapes are used)
;; is implementation-dependent. The examples here show two of the
;; many valid ways in which escaping might appear.
(test-print-case) ;Implementation A
>> THIS-AND-THE |And-something-else|
>> this-and-the a\n\|d-\s\o\m\|e\|t\h\i\n\g-\e\lse
>> This-And-The A\n\|d-\s\o\m\|e\|t\h\i\n\g-\e\lse
=> NIL
(test-print-case) ;Implementation B
>> THIS-AND-THE |And-something-else|
>> this-and-the a|nd-something-el|se
>> This-And-The A|nd-something-el|se
=> NIL
```

**Affected By:** None.

**See Also:**

[write](#)

**Notes:**

**read** normally converts lowercase characters appearing in symbols to corresponding uppercase characters, so that internally print names normally contain only uppercase characters.

## Variable \*PRINT-CIRCLE\*

**Value Type:**

a generalized boolean.

**Initial Value:**

false.

**Description:**

Controls the attempt to detect circularity and sharing in an object being printed.

If false, the printing process merely proceeds by recursive descent without attempting to detect circularity and sharing.

If true, the printer will endeavor to detect cycles and sharing in the structure to be printed, and to use #n= and #n# syntax to indicate the circularities or shared components.

If true, a user-defined print-object method can print objects to the supplied stream using write, prin1, princ, or format and expect circularities and sharing to be detected and printed using the #n# syntax. If a user-defined print-object method prints to a stream other than the one that was supplied, then circularity

detection starts over for that stream.

Note that implementations should not use #n# notation when the Lisp reader would automatically assure sharing without it (e.g., as happens with interned symbols).

### Examples:

```
(let ((a (list 1 2 3)))
  (setf (cdddr a) a)
  (let ((*print-circle* t))
    (write a)
    :done))
>> #1=(1 2 3 . #1#)
=> :DONE
```

**Affected By:** None.

### See Also:

#### write

### Notes:

An attempt to print a circular structure with \*print-circle\* set to nil may lead to looping behavior and failure to terminate.

## Variable \*PRINT-ESCAPE\*

### Value Type:

a generalized boolean.

### Initial Value:

true.

### Description:

If false, escape characters and package prefixes are not output when an expression is printed.

If true, an attempt is made to print an expression in such a way that it can be read again to produce an equal expression. (This is only a guideline; not a requirement. See \*print-readably\*.)

For more specific details of how the value of \*print-escape\* affects the printing of certain types, see Section 22.1.3 (Default Print-Object Methods).

### Examples:

```
(let ((*print-escape* t)) (write #\a))
>> #\a
=> #\a
(let ((*print-escape* nil)) (write #\a))
>> a
```

```
=> #\a
```

**Affected By:****princ, prin1, format****See Also:****write, readable-case****Notes:**

**princ** effectively binds **\*print-escape\*** to *false*. **prin1** effectively binds **\*print-escape\*** to *true*.

**Variable \*PRINT-GENSYM\*****Value Type:**

a *generalized boolean*.

**Initial Value:**

*true*.

**Description:**

Controls whether the prefix "#:" is printed before *apparently uninterned symbols*. The prefix is printed before such *symbols* if and only if the *value* of **\*print-gensym\*** is *true*.

**Examples:**

```
(let ((*print-gensym* nil))
  (print (gensym)))
>> G6040
=> #:G6040
```

**Affected By:** None.**See Also:****write, \*print-escape\*****Notes:** None.**Variable \*PRINT-LEVEL\*, \*PRINT-LENGTH\*****Value Type:**

a non-negative *integer*, or **nil**.

**Initial Value:****nil**.**Description:**

**\*print-level\*** controls how many levels deep a nested *object* will print. If it is *false*, then no control is exercised. Otherwise, it is an *integer* indicating the maximum level to be printed. An *object* to be printed is at level 0; its components (as of a *list* or *vector*) are at level 1; and so on. If an *object* to be recursively printed has components and is at a level equal to or greater than the *value* of **\*print-level\***, then the *object* is printed as "#".

**\*print-length\*** controls how many elements at a given level are printed. If it is *false*, there is no limit to the number of components printed. Otherwise, it is an *integer* indicating the maximum number of *elements* of an *object* to be printed. If exceeded, the printer will print "..." in place of the other *elements*. In the case of a *dotted list*, if the *list* contains exactly as many *elements* as the *value* of **\*print-length\***, the terminating *atom* is printed rather than printing "..."

**\*print-level\*** and **\*print-length\*** affect the printing of any *object* printed with a list-like syntax. They do not affect the printing of *symbols*, *strings*, and *bit vectors*.

**Examples:**

```
(setq a '(1 (2 (3 (4 (5 (6))))))) => (1 (2 (3 (4 (5 (6))))))
(dotimes (i 8)
  (let ((*print-level* i))
    (format t "~&~D -- ~S~%" i a)))
>> 0 -- #
>> 1 -- (1 #)
>> 2 -- (1 (2 #))
>> 3 -- (1 (2 (3 #)))
>> 4 -- (1 (2 (3 (4 #))))
>> 5 -- (1 (2 (3 (4 (5 #)))))
>> 6 -- (1 (2 (3 (4 (5 (6))))))
>> 7 -- (1 (2 (3 (4 (5 (6))))))
=> NIL

(setq a '(1 2 3 4 5 6)) => (1 2 3 4 5 6)
(dotimes (i 7)
  (let ((*print-length* i))
    (format t "~&~D -- ~S~%" i a)))
>> 0 -- (...)
>> 1 -- (1 ...)
>> 2 -- (1 2 ...)
>> 3 -- (1 2 3 ...)
>> 4 -- (1 2 3 4 ...)
>> 5 -- (1 2 3 4 5 6)
>> 6 -- (1 2 3 4 5 6)
=> NIL

(dolist (level-length '((0 1) (1 1) (1 2) (1 3) (1 4)
                        (2 1) (2 2) (2 3) (3 2) (3 3) (3 4)))
  (let ((*print-level* (first level-length))
        (*print-length* (second level-length)))
    (format t "~&~D ~D -- ~S~%" 
            *print-level* *print-length*
            '(if (member x y) (+ (car x) 3) '(foo . #(a b c d "Baz"))))))
```

```

>> 0 1 -- #
>> 1 1 -- (IF ...)
>> 1 2 -- (IF # ...)
>> 1 3 -- (IF # # ...)
>> 1 4 -- (IF # # #)
>> 2 1 -- (IF ...)
>> 2 2 -- (IF (MEMBER X ...) ...)
>> 2 3 -- (IF (MEMBER X Y) (+ # 3) ...)
>> 3 2 -- (IF (MEMBER X ...) ...)
>> 3 3 -- (IF (MEMBER X Y) (+ (CAR X) 3) ...)
>> 3 4 -- (IF (MEMBER X Y) (+ (CAR X) 3) '(FOO . #(A B C D ...)))
=> NIL

```

**Affected By:** None.

**See Also:**

### write

**Notes:** None.

## **Variable \*PRINT-LINES\***

**Value Type:**

a non-negative integer, or nil.

**Initial Value:**

nil.

**Description:**

When the value of **\*print-lines\*** is other than nil, it is a limit on the number of output lines produced when something is pretty printed. If an attempt is made to go beyond that many lines, "... is printed at the end of the last line followed by all of the suffixes (closing delimiters) that are pending to be printed.

**Examples:**

```

(let ((*print-right-margin* 25) (*print-lines* 3))
  (pprint '(progn (setq a 1 b 2 c 3 d 4))))
>> (PROGN (SETQ A 1
                  B 2
                  C 3 ...))
=> <no values>

```

**See Also:** None.

**Notes:**

The "..." notation is intentionally different than the "..." notation used for level abbreviation, so that the two different situations can be visually distinguished.

## CLHS: Declaration DYNAMIC-EXTENT

This notation is used to increase the likelihood that the *Lisp reader* will signal an error if an attempt is later made to read the abbreviated output. Note however that if the truncation occurs in a *string*, as in "This string has been trunc...", the problem situation cannot be detected later and no such error will be signaled.

### **Variable \*PRINT-MISER-WIDTH\***

#### **Value Type:**

a non-negative *integer*, or **nil**.

#### **Initial Value:**

*implementation-dependent*

#### **Description:**

If it is not **nil**, the *pretty printer* switches to a compact style of output (called miser style) whenever the width available for printing a substructure is less than or equal to this many *ems*.

**Examples:** None.

**See Also:** None.

**Notes:** None.

### **Variable \*PRINT-PPRINT-DISPATCH\***

#### **Value Type:**

a *pprint dispatch table*.

#### **Initial Value:**

*implementation-dependent*, but the initial entries all use a special class of priorities that have the property that they are less than every priority that can be specified using **set-pprint-dispatch**, so that the initial contents of any entry can be overridden.

#### **Description:**

The *pprint dispatch table* which currently controls the *pretty printer*.

**Examples:** None.

**See Also:**

**\*print-pretty\***, Section 22.2.1.4 (Pretty Print Dispatch Tables)

**Notes:**

The intent is that the initial *value* of this *variable* should cause `traditional' *pretty printing of code*. In general, however, you can put a value in **\*print-pretty-dispatch\*** that makes pretty-printed output look exactly like non-pretty-printed output. Setting **\*print-pretty\*** to *true* just causes the functions contained in the *current pprint dispatch table* to have priority over normal **print-object** methods; it has no magic way of enforcing that those functions actually produce pretty output. For details, see [Section 22.2.1.4 \(Pretty Print Dispatch Tables\)](#).

## **Variable \*PRINT-PRETTY\***

### **Value Type:**

a *generalized boolean*.

### **Initial Value:**

*implementation-dependent*.

### **Description:**

Controls whether the *Lisp printer* calls the *pretty printer*.

If it is *false*, the *pretty printer* is not used and a minimum of *whitespace*[1] is output when printing an expression.

If it is *true*, the *pretty printer* is used, and the *Lisp printer* will endeavor to insert extra *whitespace*[1] where appropriate to make *expressions* more readable.

**\*print-pretty\*** has an effect even when the *value* of **\*print-escape\*** is *false*.

### **Examples:**

```
(setq *print-pretty* 'nil) => NIL
(progn (write '(let ((a 1) (b 2) (c 3)) (+ a b c))) nil)
>> (LET ((A 1) (B 2) (C 3)) (+ A B C))
=> NIL
(let ((*print-pretty* t))
  (progn (write '(let ((a 1) (b 2) (c 3)) (+ a b c))) nil))
>> (LET ((A 1)
>>       (B 2)
>>       (C 3))
>>     (+ A B C))
=> NIL
;; Note that the first two expressions printed by this next form
;; differ from the second two only in whether escape characters are printed.
;; In all four cases, extra whitespace is inserted by the pretty printer.
(flet ((test (x)
           (let ((*print-pretty* t))
             (print x)
             (format t "~%~S " x)
             (terpri) (princ x) (princ " ")
             (format t "~%~A " x))))
  (test #'(lambda () (list "a" #'c #'d))))
>> #'(LAMBDA ()
>>      (LIST "a" #'C #'D))
>> #'(LAMBDA ()
```

```
>>      (LIST "a" #'C #'D))
>> #'(LAMBDA ()
>>      (LIST a b 'C #'D))
>> #'(LAMBDA ()
>>      (LIST a b 'C #'D))
=> NIL
```

**Affected By:** None.

**See Also:**

### write

**Notes:** None.

## **Variable \*PRINT-READABLY\***

**Value Type:**

a *generalized boolean*.

**Initial Value:**

*false*.

**Description:**

If **\*print-readably\*** is *true*, some special rules for printing *objects* go into effect. Specifically, printing any *object* O1 produces a printed representation that, when seen by the *Lisp reader* while the *standard readable* is in effect, will produce an *object* O2 that is *similar* to O1. The printed representation produced might or might not be the same as the printed representation produced when **\*print-readably\*** is *false*. If printing an *object readable* is not possible, an error of *type print-not-readable* is signaled rather than using a syntax (e.g., the "#<" syntax) that would not be readable by the same *implementation*. If the *value* of some other *printer control variable* is such that these requirements would be violated, the *value* of that other *variable* is ignored.

Specifically, if **\*print-readably\*** is *true*, printing proceeds as if **\*print-escape\***, **\*print-array\***, and **\*print-gensym\*** were also *true*, and as if **\*print-length\***, **\*print-level\***, and **\*print-lines\*** were *false*.

If **\*print-readably\*** is *false*, the normal rules for printing and the normal interpretations of other *printer control variables* are in effect.

Individual *methods* for **print-object**, including user-defined *methods*, are responsible for implementing these requirements.

If **\*read-eval\*** is *false* and **\*print-readably\*** is *true*, any such method that would output a reference to the "# ." *reader macro* will either output something else or will signal an error (as described above).

**Examples:**

```
(let ((x (list "a" '\a (gensym) '((a (b (c))) d e f g)))
      (*print-escape* nil)
      (*print-gensym* nil))
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(*print-level* 3)
(*print-length* 3))
(write x)
(let ((*print-readably* t))
  (terpri)
  (write x)
  :done))
>> (a a G4581 ((A #) D E ...))
>> ("a" |a| #:G4581 ((A (B (C))) D E F G))
=> :DONE

;; This is setup code is shared between the examples
;; of three hypothetical implementations which follow.
(setq table (make-hash-table)) => #<HASH-TABLE EQL 0/120 32005763>
(setf (gethash table 1) 'one) => ONE
(setf (gethash table 2) 'two) => TWO

;; Implementation A
(let ((*print-readably* t)) (print table))
Error: Can't print #<HASH-TABLE EQL 0/120 32005763> readably.

;; Implementation B
;; No standardized #S notation for hash tables is defined,
;; but there might be an implementation-defined notation.
(let ((*print-readably* t)) (print table))
>> #S(HASH-TABLE :TEST EQL :SIZE 120 :CONTENTS (1 ONE 2 TWO))
=> #<HASH-TABLE EQL 0/120 32005763>

;; Implementation C
;; Note that #. notation can only be used if *READ-EVAL* is true.
;; If *READ-EVAL* were false, this same implementation might have to
;; signal an error unless it had yet another printing strategy to fall
;; back on.
(let ((*print-readably* t)) (print table))
>> #.(LET ((HASH-TABLE (MAKE-HASH-TABLE)))
     (SETF (GETHASH 1 HASH-TABLE) ONE)
     (SETF (GETHASH 2 HASH-TABLE) TWO)
     HASH-TABLE)
=> #<HASH-TABLE EQL 0/120 32005763>
```

**Affected By:** None.

**See Also:**

[write, print–unreadable–object](#)

**Notes:**

The rules for "similarity" imply that #A or #( syntax cannot be used for arrays of element type other than t. An implementation will have to use another syntax or signal an error of type print–not–readable.

## Variable \*PRINT–RIGHT–MARGIN\*

**Value Type:**

a non-negative integer, or nil.

**Initial Value:**nil.**Description:**

If it is non-nil, it specifies the right margin (as integer number of ems) to use when the pretty printer is making layout decisions.

If it is nil, the right margin is taken to be the maximum line length such that output can be displayed without wraparound or truncation. If this cannot be determined, an implementation-dependent value is used.

**Examples:** None.

**See Also:** None.

**Notes:**

This measure is in units of ems in order to be compatible with implementation-defined variable-width fonts while still not requiring the language to provide support for fonts.

**Variable \*READ-BASE\*****Value Type:**a radix.**Initial Value:**

10.

**Description:**

Controls the interpretation of tokens by read as being integers or ratios.

The value of \*read-base\*, called the current input base, is the radix in which integers and ratios are to be read by the Lisp reader. The parsing of other numeric types (e.g., floats) is not affected by this option.

The effect of \*read-base\* on the reading of any particular rational number can be locally overridden by explicit use of the #O, #X, #B, or #nR syntax or by a trailing decimal point.

**Examples:**

```
(dotimes (i 6)
  (let ((*read-base* (+ 10. i)))
    (let ((object (read-from-string "(\\DAD DAD |BEE| BEE 123. 123)")))
      (print (list *read-base* object)))))

>> (10 (DAD DAD BEE BEE 123 123))
>> (11 (DAD DAD BEE BEE 123 146))
>> (12 (DAD DAD BEE BEE 123 171))
>> (13 (DAD DAD BEE BEE 123 198))
>> (14 (DAD 2701 BEE BEE 123 227))
```

## CLHS: Declaration DYNAMIC-EXTENT

```
>> (15 (DAD 3088 BEE 2699 123 258))  
=> NIL
```

**Affected By:** None.

**See Also:** None.

**Notes:**

Altering the input radix can be useful when reading data files in special formats.

## Variable \*READ-DEFAULT-FLOAT-FORMAT\*

**Value Type:**

one of the atomic type specifiers **short-float**, **single-float**, **double-float**, or **long-float**, or else some other type specifier defined by the implementation to be acceptable.

**Initial Value:**

The symbol **single-float**.

**Description:**

Controls the floating-point format that is to be used when reading a floating-point number that has no exponent marker or that has e or E for an exponent marker. Other exponent markers explicitly prescribe the floating-point format to be used.

The printer uses **\*read-default-float-format\*** to guide the choice of exponent markers when printing floating-point numbers.

**Examples:**

```
(let ((*read-default-float-format* 'double-float))  
  (read-from-string "(1.0 1.0e0 1.0s0 1.0f0 1.0d0 1.0L0)"))  
=> (1.0 1.0 1.0 1.0 1.0) ;Implementation has float format F.  
=> (1.0 1.0 1.0s0 1.0 1.0 1.0) ;Implementation has float formats S and F.  
=> (1.0d0 1.0d0 1.0 1.0 1.0d0 1.0d0) ;Implementation has float formats F and D.  
=> (1.0d0 1.0d0 1.0s0 1.0 1.0d0 1.0d0) ;Implementation has float formats S, F, D.  
=> (1.0d0 1.0d0 1.0 1.0 1.0d0 1.0L0) ;Implementation has float formats F, D, L.  
=> (1.0d0 1.0d0 1.0s0 1.0 1.0d0 1.0L0) ;Implementation has formats S, F, D, L.
```

**Affected By:** None.

**See Also:** None.

**Notes:** None.

## Variable \*READ-EVAL\*

**Value Type:**

a *generalized boolean*.

**Initial Value:**

*true*.

**Description:**

If it is *true*, the # . *reader macro* has its normal effect. Otherwise, that *reader macro* signals an error of *type reader-error*.

**Examples:** None.

**Affected By:** None.

**See Also:**

**\*print-readably\***

**Notes:**

If **\*read-eval\*** is *false* and **\*print-readably\*** is *true*, any *method* for **print-object** that would output a reference to the # . *reader macro* either outputs something different or signals an error of *type print-not-readable*.

## Variable \*READ-SUPPRESS\*

**Value Type:**

a *generalized boolean*.

**Initial Value:**

*false*.

**Description:**

This variable is intended primarily to support the operation of the read-time conditional notations #+ and #-. It is important for the *reader macros* which implement these notations to be able to skip over the printed representation of an *expression* despite the possibility that the syntax of the skipped *expression* may not be entirely valid for the current implementation, since #+ and #- exist in order to allow the same program to be shared among several Lisp implementations (including dialects other than Common Lisp) despite small incompatibilities of syntax.

If it is *false*, the *Lisp reader* operates normally.

If the *value* of **\*read-suppress\*** is *true*, **read**, **read-preserving-whitespace**, **read-delimited-list**, and **read-from-string** all return a *primary value* of **nil** when they complete successfully; however, they continue to parse the representation of an *object* in the normal way, in order to skip over the *object*, and continue to indicate *end of file* in the normal way. Except as noted below, any *standardized reader macro*[2] that is defined to *read*[2] a following *object* or *token* will do so, but not signal an error if the *object* read is not of an

## CLHS: Declaration DYNAMIC-EXTENT

appropriate type or syntax. The standard syntax and its associated reader macros will not construct any new objects (e.g., when reading the representation of a symbol, no symbol will be constructed or interned).

### Extended tokens

All extended tokens are completely uninterpreted. Errors such as those that might otherwise be signaled due to detection of invalid potential numbers, invalid patterns of package markers, and invalid uses of the dot character are suppressed.

### Dispatching macro characters (including sharpsign)

Dispatching macro characters continue to parse an infix numerical argument, and invoke the dispatch function. The standardized sharpsign reader macros do not enforce any constraints on either the presence of or the value of the numerical argument.

#=

The #= notation is totally ignored. It does not read a following object. It produces no object, but is treated as whitespace[2].

##

The ## notation always produces nil.

No matter what the value of \*read-suppress\*, parentheses still continue to delimit and construct lists; the #( notation continues to delimit vectors; and comments, strings, and the single-quote and backquote notations continue to be interpreted properly. Such situations as ' ), #<, # ), and #<Space> continue to signal errors.

### Examples:

```
(let ((*read-suppress* t))
  (mapcar #'read-from-string
          '("#(foo bar baz)" "#P(:type :lisp)" "#c1.2"
            "#.(PRINT 'FOO)" "#3AHELLO" "#S(INTEGER)"
            "#*ABC" "#\GARBAGE" "#RALPHA" "#3R444"))
  => (NIL NIL NIL NIL NIL NIL NIL NIL NIL))
```

**Affected By:** None.

**See Also:**

[read](#), [Section 2 \(Syntax\)](#)

**Notes:**

Programmers and implementations that define additional macro characters are strongly encouraged to make them respect \*read-suppress\* just as standardized macro characters do. That is, when the value of \*read-suppress\* is true, they should ignore type errors when reading a following object and the functions that implement dispatching macro characters should tolerate nil as their infix parameter value even if a numeric value would ordinarily be required.

## Variable \*READTABLE\*

**Value Type:**

a readtable.

**Initial Value:**

## CLHS: Declaration DYNAMIC-EXTENT

A readtable that conforms to the description of Common Lisp syntax in [Section 2 \(Syntax\)](#).

### Description:

The value of **\*readtable\*** is called the current readtable. It controls the parsing behavior of the Lisp reader, and can also influence the Lisp printer (e.g., see the function readable-case).

### Examples:

```
(readtablep *readtable*) => true
(setq zvar 123) => 123
(set-syntax-from-char #\z #'(setq table2 (copy-readtable))) => T
zvar => 123
(setq *readtable* table2) => #<READTABLE>
zvar => VAR
(setq *readtable* (copy-readtable nil)) => #<READTABLE>
zvar => 123
```

### Affected By:

[compile-file](#), [load](#)

### See Also:

[compile-file](#), [load](#), [readtable](#), [Section 2.1.1.1 \(The Current Readtable\)](#)

**Notes:** None.

## Variable \*RANDOM-STATE\*

### Value Type:

a random state.

### Initial Value:

implementation-dependent.

### Description:

The current random state, which is used, for example, by the function random when a random state is not explicitly supplied.

### Examples:

```
(random-state-p *random-state*) => true
(setq snap-shot (make-random-state))
;; The series from any given point is random,
;; but if you backtrack to that point, you get the same series.
(list (loop for i from 1 to 10 collect (random))
      (let ((*random-state* snap-shot))
        (loop for i from 1 to 10 collect (random)))
      (loop for i from 1 to 10 collect (random)))
```

## CLHS: Declaration DYNAMIC-EXTENT

```
(let ((*random-state* snap-shot))
      (loop for i from 1 to 10 collect (random))))
=> ((19 16 44 19 96 15 76 96 13 61)
    (19 16 44 19 96 15 76 96 13 61)
    (16 67 0 43 70 79 58 5 63 50)
    (16 67 0 43 70 79 58 5 63 50))
```

### Affected By:

The implementation.

random.

### See Also:

make-random-state, random, random-state

### Notes:

Binding \*random-state\* to a different random state object correctly saves and restores the old random state object.

## Constant Variable SHORT–FLOAT–EPSILON, SHORT–FLOAT–NEGATIVE–EPSILON, SINGLE–FLOAT–EPSILON, SINGLE–FLOAT–NEGATIVE–EPSILON, DOUBLE–FLOAT–EPSILON, DOUBLE–FLOAT–NEGATIVE–EPSILON, LONG–FLOAT–EPSILON, LONG–FLOAT–NEGATIVE–EPSILON

### Constant Value:

implementation-dependent.

### Description:

The value of each of the constants short–float–epsilon, single–float–epsilon, double–float–epsilon, and long–float–epsilon is the smallest positive float <EPSILON> of the given format, such that the following expression is true when evaluated:

```
(not (= (float 1 <EPSILON>) (+ (float 1 <EPSILON>) <EPSILON>)))
```

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 float <EPSILON> of the given format, such that the following expression is true when evaluated:

Constant Variable SHORT–FLOAT–EPSILON, SHORT–FLOAT–NEGATIVE–EPSILON, SINGLE–FLOAT–

## CLHS: Declaration DYNAMIC-EXTENT

```
(not (= (float 1 <EPSILON>) (- (float 1 <EPSILON>) <EPSILON>))))
```

**Examples:** None.

**See Also:** None.

**Notes:** None.

## **Variable I, II, III**

**Value Type:**

a *proper list*.

**Initial Value:**

*implementation-dependent*.

**Description:**

The *variables* *I*, *II*, and *III* are maintained by the *Lisp read–eval–print loop* to save the values of results that were printed at the end of the loop.

The *value* of *I* is a *list* of the most recent *values* that were printed, the *value* of *II* is the previous value of *I*, and the *value* of *III* is the previous value of *II*.

The *values* of *I*, *II*, and *III* are updated immediately prior to printing the *return value* of a top-level *form* by the *Lisp read–eval–print loop*. If the *evaluation* of such a *form* is aborted prior to its normal return, the values of *I*, *II*, and *III* are not updated.

**Examples:**

```
(floor 22 7) => 3, 1  
(+ (* (car /) 7) (cadr /)) => 22
```

**Affected By:**

*Lisp read–eval–print loop*.

**See Also:**

= (variable), + (variable), \* (variable), Section 25.1.1 (Top level loop)

**Notes:** None.

## **Constant Variable T**

**Constant Value:**

Variable *I*, *II*, *III*

761

**t.**

**Description:**

The boolean representing true, and the canonical generalized boolean representing true. Although any object other than nil is considered true, t is generally used when there is no special reason to prefer one such object over another.

The symbol t is also sometimes used for other purposes as well. For example, as the name of a class, as a designator (e.g., a stream designator) or as a special symbol for some syntactic reason (e.g., in case and typecase to label the otherwise-clause).

**Examples:**

```
t => T
(eq t 't) => true
(find-class 't) => #<CLASS T 610703333>
(case 'a (a 1) (t 2)) => 1
(case 'b (a 1) (t 2)) => 2
(prin1 'hello t)
>> HELLO
=> HELLO
```

**See Also:**nil**Notes:** None.**Variable \*TERMINAL-IO\*****Value Type:**

a bidirectional stream.

**Initial Value:**

implementation-dependent, but it must be an open stream that is not a generalized synonym stream to an I/O customization variables but that might be a generalized synonym stream to the value of some I/O customization variable.

**Description:**

The value of \*terminal-io\*, called terminal I/O, is ordinarily a bidirectional stream that connects to the user's console. Typically, writing to this stream would cause the output to appear on a display screen, for example, and reading from the stream would accept input from a keyboard. It is intended that standard input functions such as read and read-char, when used with this stream, cause echoing of the input into the output side of the stream. The means by which this is accomplished are implementation-dependent.

The effect of changing the value of \*terminal-io\*, either by binding or assignment, is implementation-defined.

**Examples:**

```
(progn (prinl 'foo) (prinl 'bar *terminal-io*))  
>> FOOBAR  
=> BAR  
(with-output-to-string (*standard-output*)  
  (prinl 'foo)  
  (prinl 'bar *terminal-io*))  
>> BAR  
=> "FOO"
```

**Affected By:** None.

**See Also:**

\*debug-io\*, \*error-output\*, \*query-io\*, \*standard-input\*, \*standard-output\*, \*trace-output\*

**Notes:** None.