Programming Language—Common Lisp

12. Numbers

12.1 Number Concepts

12.1.1 Numeric Operations

Common Lisp provides a large variety of operations related to *numbers*. This section provides an overview of those operations by grouping them into categories that emphasize some of the relationships among them.

Figure 12–1 shows *operators* relating to arithmetic operations.

*****	**1+**	**gcd**
+	**1-**	**incf**
-	**conjugate**	**lcm**
/	**decf**	

Figure 12–1. Operators relating to Arithmetic.

Figure 12–2 shows *defined names* relating to exponential, logarithmic, and trigonometric operations.

abs	**cos**	**signum**
acos	**cosh**	**sin**
acosh	**exp**	**sinh**
asin	**expt**	**sqrt**
asinh	**isqrt**	**tan**
atan	**log**	**tanh**
atanh	**phase**	
cis	**pi**	

Figure 12–2. Defined names relating to Exponentials, Logarithms, and Trigonometry.

Figure 12–3 shows *operators* relating to numeric comparison and predication.

/=	**>=**	**oddp**
<	**evenp**	**plusp**
<=	**max**	**zerop**
=	**min**	
>	**minusp**	

Figure 12–3. Operators for numeric comparison and predication.

Figure 12–4 shows *defined names* relating to numeric type manipulation and coercion.

ceiling	**float-radix**	**rational**
complex	**float-sign**	**rationalize**
decode-float	**floor**	**realpart**
denominator	**fround**	**rem**
fceiling	**ftruncate**	**round**
ffloor	**imagpart**	**scale-float**
float	**integer-decode-float**	**truncate**
float-digits	**mod**	
float-precision	**numerator**	

Figure 12–4. Defined names relating to numeric type manipulation and coercion.

12.1.1.1 Associativity and Commutativity in Numeric Operations

For functions that are mathematically associative (and possibly commutative), a *conforming implementation* may process the *arguments* in any manner consistent with associative (and possibly commutative) rearrangement. This does not affect the order in which the *argument forms* are *evaluated*; for a discussion of evaluation order, see Section 3.1.2.1.2.3 (Function Forms). What is unspecified is only the order in which the *parameter values* are processed. This implies that *implementations* may differ in which automatic *coercions* are applied; see Section 12.1.1.2 (Contagion in Numeric Operations).

A *conforming program* can control the order of processing explicitly by separating the operations into separate (possibly nested) *function forms*, or by writing explicit calls to *functions* that perform coercions.

12.1.1.1.1 Examples of Associativity and Commutativity in Numeric Operations

Consider the following expression, in which we assume that `1.0` and `1.0e-15` both denote *single floats*:

```
(+ 1/3 2/3 1.0d0 1.0 1.0e-15)
```

One *conforming implementation* might process the *arguments* from left to right, first adding `1/3` and `2/3` to get `1`, then converting that to a *double float* for combination with `1.0d0`, then successively converting and adding `1.0` and `1.0e-15`.

Another *conforming implementation* might process the *arguments* from right to left, first performing a *single float* addition of `1.0` and `1.0e-15` (perhaps losing accuracy in the process), then converting the sum to a *double float* and adding `1.0d0`, then converting `2/3` to a *double float* and adding it, and then converting `1/3` and adding that.

A third *conforming implementation* might first scan all the *arguments*, process all the *rationals* first to keep that part of the computation exact, then find an *argument* of the largest floating-point format among all the *arguments* and add that, and then add in all other *arguments*, converting each in turn (all in a perhaps misguided attempt to make the computation as accurate as possible).

In any case, all three strategies are legitimate.

A *conforming program* could control the order by writing, for example,

```
(+ (+ 1/3 2/3) (+ 1.0d0 1.0e-15) 1.0)
```

12.1.1.2 Contagion in Numeric Operations

For information about the contagion rules for implicit coercions of *arguments* in numeric operations, see Section 12.1.4.4 (Rule of Float Precision Contagion), Section 12.1.4.1 (Rule of Float and Rational Contagion), and Section 12.1.5.2 (Rule of Complex Contagion).

12.1.1.3 Viewing Integers as Bits and Bytes

12.1.1.3.1 Logical Operations on Integers

Logical operations require *integers* as arguments; an error of *type* **type-error** should be signaled if an argument is supplied that is not an *integer*. *Integer* arguments to logical operations are treated as if they were represented in two's-complement notation.

Figure 12–5 shows *defined names* relating to logical operations on numbers.

ash	**boole-ior**	**logbitp**
boole	**boole-nand**	**logcount**
boole-1	**boole-nor**	**logeqv**
boole-2	**boole-orc1**	**logior**
boole-and	**boole-orc2**	**lognand**
boole-andc1	**boole-set**	**lognor**
boole-andc2	**boole-xor**	**lognot**
boole-c1	**integer-length**	**logorc1**
boole-c2	**logand**	**logorc2**
boole-clr	**logandc1**	**logtest**
boole-eqv	**logandc2**	**logxor**

Figure 12–5. Defined names relating to logical operations on numbers.

12.1.1.3.2 Byte Operations on Integers

The byte-manipulation *functions* use *objects* called *byte specifiers* to designate the size and position of a specific *byte* within an *integer*. The representation of a *byte specifier* is *implementation-dependent*; it might or might not be a *number*. The *function* **byte** will construct a *byte specifier*, which various other byte-manipulation *functions* will accept.

Figure 12–6 shows *defined names* relating to manipulating *bytes* of *numbers*.

byte	**deposit-field**	**ldb-test**
byte-position	**dpb**	**mask-field**
byte-size	**ldb**	

Figure 12–6. Defined names relating to byte manipulation.

12.1.2 Implementation-Dependent Numeric Constants

Figure 12–7 shows *defined names* relating to *implementation-dependent* details about *numbers*.

double-float-epsilon	**most-negative-fixnum**
double-float-negative-epsilon	**most-negative-long-float**
least-negative-double-float	**most-negative-short-float**
least-negative-long-float	**most-negative-single-float**
least-negative-short-float	**most-positive-double-float**
least-negative-single-float	**most-positive-fixnum**
least-positive-double-float	**most-positive-long-float**
least-positive-long-float	**most-positive-short-float**
least-positive-short-float	**most-positive-single-float**
least-positive-single-float	**short-float-epsilon**
long-float-epsilon	**short-float-negative-epsilon**
long-float-negative-epsilon	**single-float-epsilon**
most-negative-double-float	**single-float-negative-epsilon**

Figure 12–7. Defined names relating to implementation-dependent details about numbers.

12.1.3 Rational Computations

The rules in this section apply to *rational* computations.

12.1.3.1 Rule of Unbounded Rational Precision

Rational computations cannot overflow in the usual sense (though there may not be enough storage to represent a result), since *integers* and *ratios* may in principle be of any magnitude.

12.1.3.2 Rule of Canonical Representation for Rationals

If any computation produces a result that is a mathematical ratio of two integers such that the denominator evenly divides the numerator, then the result is converted to the equivalent *integer*.

If the denominator does not evenly divide the numerator, the canonical representation of a *rational* number is as the *ratio* that numerator and that denominator, where the greatest common divisor of the numerator and denominator is one, and where the denominator is positive and greater than one.

When used as input (in the default syntax), the notation `-0` always denotes the *integer* `0`. A *conforming implementation* must not have a representation of "minus zero" for *integers* that is distinct from its representation of zero for *integers*. However, such a distinction is possible for *floats*; see the *type* **float**.

12.1.3.3 Rule of Float Substitutability

When the arguments to an irrational mathematical *function* are all *rational* and the true mathematical result is also (mathematically) rational, then unless otherwise noted an implementation is free to return either an accurate *rational* result or a *single float* approximation. If the arguments are all *rational* but the result cannot be expressed as a *rational* number, then a *single float* approximation is always returned.

If the arguments to an irrational mathematical *function* are all of type `(or rational (complex rational))` and the true mathematical result is (mathematically) a complex number with rational real and imaginary parts, then unless otherwise noted an implementation is free to return either an accurate result of type `(or rational (complex rational))` or a *single float* (permissible only if the imaginary part of the true mathematical result is zero) or `(complex single-float)`. If the arguments are all of type `(or rational (complex rational))` but the result cannot be expressed as a *rational* or *complex rational*, then the returned value will be of *type* **single-float** (permissible only if the imaginary part of the true mathematical result is zero) or `(complex single-float)`.

Float substitutability applies neither to the rational *functions* **+**, **-**, *****, and **/** nor to the related *operators* **1+**, **1-**, **incf**, **decf**, and **conjugate**. For rational *functions*, if all arguments are *rational*, then the result is *rational*; if all arguments are of type `(or rational (complex rational))`, then the result is of type `(or rational (complex rational))`.

Function	Sample Results
abs	`(abs #c(3 4))` → `5` *or* `5.0`
acos	`(acos 1)` → `0` *or* `0.0`
acosh	`(acosh 1)` → `0` *or* `0.0`
asin	`(asin 0)` → `0` *or* `0.0`
asinh	`(asinh 0)` → `0` *or* `0.0`
atan	`(atan 0)` → `0` *or* `0.0`
atanh	`(atanh 0)` → `0` *or* `0.0`
cis	`(cis 0)` → `1` *or* `#c(1.0 0.0)`
cos	`(cos 0)` → `1` *or* `1.0`
cosh	`(cosh 0)` → `1` *or* `1.0`
exp	`(exp 0)` → `1` *or* `1.0`
expt	`(expt 8 1/3)` → `2` *or* `2.0`
log	`(log 1)` → `0` *or* `0.0`
	`(log 8 2)` → `3` *or* `3.0`
phase	`(phase 7)` → `0` *or* `0.0`
signum	`(signum #c(3 4))` → `#c(3/5 4/5)` *or* `#c(0.6 0.8)`
sin	`(sin 0)` → `0` *or* `0.0`
sinh	`(sinh 0)` → `0` *or* `0.0`
sqrt	`(sqrt 4)` → `2` *or* `2.0`
	`(sqrt 9/16)` → `3/4` *or* `0.75`
tan	`(tan 0)` → `0` *or* `0.0`
tanh	`(tanh 0)` → `0` *or* `0.0`

Figure 12–8. Functions Affected by Rule of Float Substitutability

12.1.4 Floating-point Computations

The following rules apply to floating point computations.

12.1.4.1 Rule of Float and Rational Contagion

When *rationals* and *floats* are combined by a numerical function, the *rational* is first converted to a *float* of the same format. For *functions* such as + that take more than two arguments, it is permitted that part of the operation be carried out exactly using *rationals* and the rest be done using floating-point arithmetic.

When *rationals* and *floats* are compared by a numerical function, the *function* **rational** is effectively called to convert the *float* to a *rational* and then an exact comparison is performed. In the case of *complex* numbers, the real and imaginary parts are effectively handled individually.

12.1.4.1.1 Examples of Rule of Float and Rational Contagion

```
;;;; Combining rationals with floats.
;;; This example assumes an implementation in which
;;; (float-radix 0.5) is 2 (as in IEEE) or 16 (as in IBM/360),
;;; or else some other implementation in which 1/2 has an exact
;;;  representation in floating point.
(+ 1/2 0.5) → 1.0
(- 1/2 0.5d0) → 0.0d0
(+ 0.5 -0.5 1/2) → 0.5

;;;; Comparing rationals with floats.
;;; This example assumes an implementation in which the default float
;;; format is IEEE single-float, IEEE double-float, or some other format
;;; in which 5/7 is rounded upwards by FLOAT.
(< 5/7 (float 5/7)) → true
(< 5/7 (rational (float 5/7))) → true
(< (float 5/7) (float 5/7)) → false
```

12.1.4.2 Rule of Float Approximation

Computations with *floats* are only approximate, although they are described as if the results were mathematically accurate. Two mathematically identical expressions may be computationally different because of errors inherent in the floating-point approximation process. The precision of a *float* is not necessarily correlated with the accuracy of that number. For instance, 3.142857142857142857 is a more precise approximation to π than 3.14159, but the latter is more accurate. The precision refers to the number of bits retained in the representation. When an operation combines a *short float* with a *long float*, the result will be a *long float*. Common Lisp functions assume that the accuracy of arguments to them does not exceed their precision. Therefore when two *small floats* are combined, the result is a *small float*. Common Lisp functions never convert automatically from a larger size to a smaller one.

12.1.4.3 Rule of Float Underflow and Overflow

An error of *type* **floating-point-overflow** or **floating-point-underflow** should be signaled if a floating-point computation causes exponent overflow or underflow, respectively.

12.1.4.4 Rule of Float Precision Contagion

The result of a numerical function is a *float* of the largest format among all the floating-point arguments to the *function*.

12.1.5 Complex Computations

The following rules apply to *complex* computations:

12.1.5.1 Rule of Complex Substitutability

Except during the execution of irrational and transcendental *functions*, no numerical *function* ever *yields* a *complex* unless one or more of its *arguments* is a *complex*.

12.1.5.2 Rule of Complex Contagion

When a *real* and a *complex* are both part of a computation, the *real* is first converted to a *complex* by providing an imaginary part of `0`.

12.1.5.3 Rule of Canonical Representation for Complex Rationals

If the result of any computation would be a *complex* number whose real part is of *type* **rational** and whose imaginary part is zero, the result is converted to the *rational* which is the real part. This rule does not apply to *complex* numbers whose parts are *floats*. For example, `#C(5 0)` and `5` are not *different objects* in Common Lisp(they are always the *same* under **eql**); `#C(5.0 0.0)` and `5.0` are always *different objects* in Common Lisp (they are never the *same* under **eql**, although they are the *same* under **equalp** and =).

12.1.5.3.1 Examples of Rule of Canonical Representation for Complex Rationals

```
#c(1.0 1.0) → #C(1.0 1.0)
#c(0.0 0.0) → #C(0.0 0.0)
#c(1.0 1) → #C(1.0 1.0)
#c(0.0 0) → #C(0.0 0.0)
#c(1 1) → #C(1 1)
#c(0 0) → 0
(typep #c(1 1) '(complex (eql 1))) → true
(typep #c(0 0) '(complex (eql 0))) → false
```

12.1.5.4 Principal Values and Branch Cuts

Many of the irrational and transcendental functions are multiply defined in the complex domain; for example, there are in general an infinite number of complex values for the logarithm function. In each such case, a *principal value* must be chosen for the function to return. In general, such values cannot be chosen so as to make the range continuous; lines in the domain called branch cuts must be defined, which in turn define the discontinuities in the range. Common Lisp defines the branch cuts, *principal values*, and boundary conditions for the complex functions following "Principal Values and Branch Cuts in Complex APL." The branch cut rules that apply to each function are located with the description of that function.

Figure 12–9 lists the identities that are obeyed throughout the applicable portion of the complex domain, even on the branch cuts:

sin i z = i sinh z	sinh i z = i sin z	arctan i z = i arctanh z
cos i z = cosh z	cosh i z = cos z	arcsinh i z = i arcsin z
tan i z = i tanh z	arcsin i z = i arcsinh z	arctanh i z = i arctan z

Figure 12–9. Trigonometric Identities for Complex Domain

The quadrant numbers referred to in the discussions of branch cuts are as illustrated in Figure 12–10.

```
                          Positive
                       Imaginary Axis
                             :
                          II : I
Negative Real Axis  ..................  Positive Real Axis
                         III : IV
                             :
                          Negative
                       Imaginary Axis
```

Figure 12–10. Quadrant Numbering for Branch Cuts

12.1.6 Interval Designators

The *compound type specifier* form of the numeric *type specifiers* permit the user to specify an interval on the real number line which describe a *subtype* of the *type* which would be described by the corresponding *atomic type specifier*. A *subtype* of some *type* T is specified using an ordered pair of *objects* called *interval designators* for *type* T.

The first of the two *interval designators* for *type* T can be any of the following:

a number N of *type* T

This denotes a lower inclusive bound of N. That is, *elements* of the *subtype* of T will be greater than or equal to N.

a *singleton list* whose *element* is a number M of *type* T

This denotes a lower exclusive bound of M. That is, *elements* of the *subtype* of T will be greater than M.

the symbol *

This denotes the absence of a lower bound on the interval.

The second of the two *interval designators* for *type* T can be any of the following:

a number N of *type* T

This denotes an upper inclusive bound of N. That is, *elements* of the *subtype* of T will be less than or equal to N.

a *singleton list* whose *element* is a number M of *type* T

This denotes an upper exclusive bound of M. That is, *elements* of the *subtype* of T will be less than M.

the symbol *

This denotes the absence of an upper bound on the interval.

12.1.7 Random-State Operations

Figure 12–11 lists some *defined names* that are applicable to *random states*.

random-state	**random**
make-random-state	**random-state-p**

Figure 12–11. Random-state defined names

number *System Class*

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.

complex *System Class*

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:

Section 12.1.5.3 (Rule of Canonical Representation for Complex Rationals), Section 2.3.2 (Constructing Numbers from Tokens), Section 22.1.3.1.4 (Printing Complexes)

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)`.

real *System Class*

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 **<**, **>**, **<=**, and **>=** 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*.

float *System Class*

Class Precedence List:

float, **real**, **number**, **t**

Description:

A *float* is a mathematical rational (but *not* a Common Lisp *rational*) of the form $s \cdot f \cdot b^{e-p}$, 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)`.

short-float, single-float, double-float, long-float *Type*

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 $p \log_2 b$) 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 Figure 12–12. 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*.

rational *System Class*

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*.

ratio *System Class*

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)

integer *System Class*

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**.

signed-byte *Type*

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**.

unsigned-byte *Type*

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 `(mod m)` for $m = 2^s$, or to `(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**.

mod *Type Specifier*

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*.

bit *Type*

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)`.

fixnum *Type*

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)`.

bignum *Type*

Supertypes:

bignum, **integer**, **rational**, **real**, **number**, **t**

Description:

The *type* **bignum** is defined to be exactly `(and integer (not fixnum))`.

=, /=, <, >, <=, >= *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 <, >, <=, >=: a *real*; for =, /=: a *number*.

generalized-boolean—a *generalized boolean*.

Description:

=, /=, <, >, <=, and >= perform arithmetic comparisons on their arguments as follows:

=

The value of = is *true* if all *numbers* are the same in value; otherwise it is *false*. Two *complexes* are considered equal by = if their real and imaginary parts are equal according to =.

/=

The value of /= is *true* if no two *numbers* are the same in value; otherwise it is *false*.

`<`

The value of `<` is *true* if the *numbers* are in monotonically increasing order; otherwise it is *false*.

`>`

The value of `>` is *true* if the *numbers* are in monotonically decreasing order; otherwise it is *false*.

`<=`

The value of `<=` is *true* if the *numbers* are in monotonically nondecreasing order; otherwise it is *false*.

`>=`

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 Figure 12–13.

`(= 3 3)` is *true*.	`(/= 3 3)` is *false*.
`(= 3 5)` is *false*.	`(/= 3 5)` is *true*.
`(= 3 3 3 3)` is *true*.	`(/= 3 3 3 3)` is *false*.
`(= 3 3 5 3)` is *false*.	`(/= 3 3 5 3)` is *false*.
`(= 3 6 5 2)` is *false*.	`(/= 3 6 5 2)` is *true*.
`(= 3 2 3)` is *false*.	`(/= 3 2 3)` is *false*.
`(< 3 5)` is *true*.	`(<= 3 5)` is *true*.
`(< 3 -5)` is *false*.	`(<= 3 -5)` is *false*.
`(< 3 3)` is *false*.	`(<= 3 3)` is *true*.
`(< 0 3 4 6 7)` is *true*.	`(<= 0 3 4 6 7)` is *true*.
`(< 0 3 4 4 6)` is *false*.	`(<= 0 3 4 4 6)` is *true*.
`(> 4 3)` is *true*.	`(>= 4 3)` is *true*.
`(> 4 3 2 1 0)` is *true*.	`(>= 4 3 2 1 0)` is *true*.
`(> 4 3 3 2 0)` is *false*.	`(>= 4 3 3 2 0)` is *true*.
`(> 4 3 1 2 0)` is *false*.	`(>= 4 3 1 2 0)` is *false*.
`(= 3)` is *true*.	`(/= 3)` is *true*.
`(< 3)` is *true*.	`(<= 3)` is *true*.
`(= 3.0 #c(3.0 0.0))` is *true*.	`(/= 3.0 #c(3.0 1.0))` is *true*.
`(= 3 3.0)` is *true*.	`(= 3.0s0 3.0d0)` is *true*.
`(= 0.0 -0.0)` is *true*.	`(= 5/2 2.5)` is *true*.
`(> 0.0 -0.0)` is *false*.	`(= 0 -0.0)` is *true*.
`(<= 0 x 9)` is *true* if `x` is between `0` and `9`, inclusive	
`(< 0.0 x 1.0)` is *true* if `x` is between `0.0` and `1.0`, exclusive	
`(< -1 j (length v))` is *true* if `j` is a *valid array index* for a *vector* `v`	

Figure 12–13. Uses of /=, =, <, >, <=, and >=

Exceptional Situations:

Might signal **type-error** if some *argument* is not a *real*. Might signal **arithmetic-error** if otherwise unable to fulfill its contract.

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.

max, min *Function*

Syntax:

max `&rest` *reals*$^+$ → *max-real*

max, min

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 =, 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
```

```
 (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
```

Exceptional Situations:

Should signal an error of *type* **type-error** if any *number* is not a *real*.

minusp, plusp *Function*

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
```

Exceptional Situations:

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

zerop *Function*

Syntax:

zerop *number* → *generalized-boolean*

Pronunciation:

[ˈzē(ˌ)rō(ˌ)pē]

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
(zerop #c(0 0.0)) → true
```

Exceptional Situations:

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

Notes:

```
(zerop number) ≡ (= number 0)
```

floor, ffloor, ceiling, fceiling, truncate, ftruncate, round, fround *Function*

Syntax:

floor *number* `&optional` *divisor* → *quotient, remainder*
ffloor *number* `&optional` *divisor* → *quotient, remainder*
ceiling *number* `&optional` *divisor* → *quotient, remainder*

floor, ffloor, ceiling, fceiling, truncate, ftruncate, ...

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

$$quotient \cdot divisor + remainder = number$$

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 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*+$\frac{1}{2}$), 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
 (truncate 1) → 1, 0
 (truncate .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 ~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
```

Notes:

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))`.

sin, cos, tan *Function*

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:

```
(sin 0) → 0.0
(cos 0.7853982) → 0.707107
(tan #c(0 1)) → #C(0.0 0.761594)
```

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)

asin, acos, atan *Function*

Syntax:

asin *number* $\rightarrow$ *radians*

acos *number* $\rightarrow$ *radians*

atan *number1* `&optional` *number2* $\rightarrow$ *radians*

Arguments and Values:

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 Figure 12–14.

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

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 $-\pi$ (exclusive) and π (inclusive) when minus zero is not supported. The range of the two-argument arc tangent when minus zero is supported includes $-\pi$.

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:

asin, acos, atan

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

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:

$$\texttt{arccos}\ z = \frac{\pi}{2} - \texttt{arcsin}\ z$$

or, which are equivalent,

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

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

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 π. 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 π 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:

$$\texttt{arctan}\ z = \frac{\texttt{log}\ (1 + iz) - \texttt{log}\ (1 - iz)}{2i}$$

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.

asin, acos, atan

For **atan**, the signs of *number1* (indicated as x) and *number2* (indicated as y) are used to derive quadrant information. Figure 12–15 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	$0 <$ result $< \pi/2$
	$y > 0$	$x = 0$	Positive y-axis	$\pi/2$
	$y > 0$	$x < 0$	Quadrant II	$\pi/2 <$ result $< \pi$
	$y = 0$	$x < 0$	Negative x-axis	π
*	$y = +0$	$x < 0$	Negative x-axis	$+\pi$
*	$y = -0$	$x < 0$	Negative x-axis	$-\pi$
	$y < 0$	$x < 0$	Quadrant III	$-\pi <$ result $< -\pi/2$
	$y < 0$	$x = 0$	Negative y-axis	$-\pi/2$
	$y < 0$	$x > 0$	Quadrant IV	$-\pi/2 <$ 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	$+\pi$
*	$y = -0$	$x = -0$	Origin	$-\pi$

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)
```

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.

pi *Constant Variable*

Value:

an *implementation-dependent long float*.

Description:

The best *long float* approximation to the mathematical constant π.

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
(cos pi) → -1.0L0

(defun sin-of-degrees (degrees)
  (let ((x (if (floatp degrees) degrees (float degrees pi))))
    (sin (* x (/ (float pi x) 180)))))
```

Notes:

An approximation to π 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**.

sinh, cosh, tanh, asinh, acosh, atanh *Function*

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 Figure 12–16.

Function	Definition
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	$\texttt{log}\ (x + \sqrt{1 + x^2})$
Hyperbolic arc cosine	$2\ \texttt{log}\ (\sqrt{(x+1)/2} + \sqrt{(x-1)/2})$
Hyperbolic arc tangent	$(\texttt{log}\ (1+x) - \texttt{log}\ (1-x))/2$

Figure 12–16. Mathematical definitions for hyperbolic functions

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

$$\texttt{arccosh}\ z = 2\ \texttt{log}\left(\sqrt{(z+1)/2} + \sqrt{(z-1)/2}\right).$$

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 π (inclusive). A number with real part zero is in the range if its imaginary part is between zero (inclusive) and π (inclusive).

sinh, cosh, tanh, asinh, acosh, atanh

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

$$\texttt{arcsinh}\ z = \texttt{log}\left(z + \sqrt{1 + z^2}\right).$$

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:

$$\texttt{arctanh}\ z = \frac{\texttt{log}\ (1 + z) - \texttt{log}\ (1 - z)}{2}.$$

Note that:

$$i\ \texttt{arctan}\ z = \texttt{arctanh}\ iz.$$

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 $-\pi i/2$ and $\pi i/2$ excluded.

Examples:

```
(sinh 0) → 0.0
(cosh (complex 0 -1)) → #C(0.540302 -0.0)
```

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:

* `&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)
```

Exceptional Situations:

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

See Also:

Section 12.1.1 (Numeric Operations), Section 12.1.3 (Rational Computations), Section 12.1.4 (Floating-point Computations), Section 12.1.5 (Complex Computations)

+ *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
```

Exceptional Situations:

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

See Also:

Section 12.1.1 (Numeric Operations), Section 12.1.3 (Rational Computations), Section 12.1.4 (Floating-point Computations), Section 12.1.5 (Complex Computations)

− *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
(eql (- 0.0) -0.0) → true
(- #c(100 45) #c(0 45)) → 100
(- 10 1 2 3 4) → 0
```

Exceptional Situations:

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

See Also:

Section 12.1.1 (Numeric Operations), Section 12.1.3 (Rational Computations), Section 12.1.4 (Floating-point Computations), Section 12.1.5 (Complex Computations)

/ *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.

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)
```

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**

1+, 1− *Function*

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
```

Exceptional Situations:

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.

abs *Function*

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
(eql (abs -0.0) -0.0) → true
```

See Also:

Section 12.1.3.3 (Rule of Float Substitutability)

Notes:

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.

evenp, oddp *Function*

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 10000000000000000000000) → false
(oddp -1) → true
```

Exceptional Situations:

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

Notes:

```
(evenp integer) ≡ (not (oddp integer))
(oddp integer)  ≡ (not (evenp integer))
```

exp, expt *Function*

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 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
```

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
```

gcd *Function*

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
(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
```

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)
```

incf, decf *Macro*

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.

See Also:

+, **-**, **1+**, **1-**, **setf**

---

lcm *Function*

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
```

Exceptional Situations:

Should signal **type-error** if any argument is not an *integer*.

See Also:

gcd

log *Function*

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

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 π (inclusive) if minus zero is not supported, or $-\pi$ (inclusive) and π (inclusive) if minus zero is supported.

The two-argument logarithm function is defined as

```
(log base number)
≡ (/ (log number) (log base))
```

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.

See Also:

exp, **expt**, Section 12.1.3.3 (Rule of Float Substitutability)

mod, rem *Function*

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
```

See Also:

floor, **truncate**

Notes:

The result of **mod** is either zero or a *real* with the same sign as *divisor*.

signum *Function*

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*.

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
```

See Also:

Section 12.1.3.3 (Rule of Float Substitutability)

Notes:

```
(signum x) ≡ (if (zerop x) x (/ x (abs x)))
```

sqrt, isqrt *Function*

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*.

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)
```

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.

---

random-state *System Class*

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)

make-random-state *Function*

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))
```

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.

random *Function*

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.

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.

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.

random-state-p *Function*

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
```

See Also:

make-random-state, ***random-state***

Notes:

```
(random-state-p object) ≡ (typep object 'random-state)
```

∗random-state∗ *Variable*

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))
      (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*.

numberp *Function*

Syntax:

numberp *object* → *generalized-boolean*

Arguments and Values:

object—an *object*.

generalized-boolean—a *generalized boolean*.

Description:

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
```

Notes:

(numberp *object*) ≡ (typep *object* 'number)

cis *Function*

Syntax:

cis *radians* → *number*

Arguments and Values:

radians—a *real*.

number—a *complex*.

Description:

cis returns the value of $e^{i \cdot \ radians}$, 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)
```

See Also:

Section 12.1.3.3 (Rule of Float Substitutability)

complex *Function*

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)
```

See Also:

realpart, **imagpart**, Section 2.4.8.11 (Sharpsign C)

complexp *Function*

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
```

See Also:

complex (*function* and *type*), **typep**

Notes:

```
(complexp object) ≡ (typep object 'complex)
```

conjugate *Function*

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
```

Notes:

For a *complex* number `z`,

```
(conjugate z) ≡ (complex (realpart z) (- (imagpart z)))
```

phase *Function*

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 $-\pi$ (exclusive) if minus zero is not supported, or $-\pi$ (inclusive) if minus zero is supported, to π (inclusive). The phase of a positive *real* number is zero; that of a negative *real* number is π. 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 $-\pi$ (exclusive) and π (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
```

Exceptional Situations:

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

See Also:

Section 12.1.3.3 (Rule of Float Substitutability)

realpart, imagpart *Function*

Syntax:

realpart *number* → *real*

imagpart *number* → *real*

Arguments and Values:

number—a *number*.

real—a *real*.

Description:

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
```

Exceptional Situations:

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

See Also:

complex

upgraded-complex-part-type *Function*

Syntax:

upgraded-complex-part-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 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*.

See Also:

complex (*function* and *type*)

Notes:

realp *Function*

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
```

Notes:

(realp *object*) ≡ (typep *object* 'real)

numerator, denominator *Function*

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
```

See Also:

/

Notes:

```
(gcd (numerator x) (denominator x)) → 1
```

rational, rationalize *Function*

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
```

Affected By:

The *implementation*.

Exceptional Situations:

Should signal an error of *type* **type-error** if *number* is not a *real*. Might signal **arithmetic-error**.

Notes:

It is always the case that

```
(float (rational x) x) ≡ x
```

and

```
(float (rationalize x) x) ≡ x
```

That is, rationalizing a *float* by either method and then converting it back to a *float* of the same format produces the original *number*.

rationalp *Function*

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
```

See Also:

rational

Notes:

```
(rationalp object) ≡ (typep object 'rational)
```

ash *Function*

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 `floor`($integer \cdot 2^{count}$). 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:

```
(ash 16 1) → 32
(ash 16 0) → 16
(ash 16 -1) → 8
(ash -100000000000000000000000000000000 -100) → -79
```

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**.

Notes:

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

integer-length *Function*

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
```

Exceptional Situations:

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

Notes:

This function could have been defined by:

```
(defun integer-length (integer)
  (ceiling (log (if (minusp integer)
                    (- integer)
                    (1+ 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)`.

integerp *Function*

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:

```
(integerp 1) → true
(integerp (expt 2 130)) → true
(integerp 6/5) → false
(integerp nil) → false
```

Notes:

```
(integerp object) ≡ (typep object 'integer)
```

parse-integer *Function*

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 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
```

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.

boole *Function*

Syntax:

boole *op integer-1 integer-2* → *result-integer*

Arguments and Values:

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

integer-1—an *integer*.

boole

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 Figure 12–17.

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

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)
```

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 Variable*

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

logand, logandc1, logandc2, logeqv, logior, lognand, lognor, lognot, logorc1, logorc2, logxor *Function*

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*

logior &rest *integers* → *result-integer*

lognand *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**, **lognand**, **lognor**, **lognot**, **logorc1**, **logorc2**, and **logxor** perform bit-wise logical operations on their *arguments*, that are treated as if they were binary.

Figure 12–18 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.

logand, logandc1, logandc2, logeqv, logior, lognand, ...

Function	Identity	Operation performed
logandc1	—	and complement of *integer-1* with *integer-2*
logandc2	—	and *integer-1* with complement of *integer-2*
logand	-1	and
logeqv	-1	equivalence (exclusive nor)
logior	0	inclusive or
lognand	—	complement of *integer-1* and *integer-2*
lognor	—	complement of *integer-1* or *integer-2*
lognot	—	complement
logorc1	—	or complement of *integer-1* with *integer-2*
logorc2	—	or *integer-1* with complement of *integer-2*
logxor	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:

```
 (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
```

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)
(logiorc2 n1 n2) ≡ (logior n1 (lognot n2))
(logbitp j (lognot x)) ≡ (not (logbitp j x))
```

logbitp *Function*

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^{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
```

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*.

Notes:

```
(logbitp k n) ≡ (ldb-test (byte 1 k) n)
```

logcount *Function*

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
```

Exceptional Situations:

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

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))
```

logtest *Function*

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
```

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*.

Notes:

```
(logtest x y) ≡ (not (zerop (logand x y)))
```

byte, byte-size, byte-position *Function*

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
```

See Also:

ldb, **dpb**

Notes:

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

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

```
(ldb (byte 0 3) #o7777) → 0
(dpb #o7777 (byte 0 3) 0) → 0
```

deposit-field *Function*

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
```

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**.

dpb *Function*

Syntax:

dpb *newbyte bytespec integer* → *result-integer*

Pronunciation:

[ˌ**d**ϵˈ**pib**] or [ˌ**d**ϵˈ**p**ϵ**b**] or [ˈ**dē**ˈ**pē**ˈ**bē**]

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
(dpb -2 (byte 2 10) 0) → 2048
(dpb 1 (byte 2 10) 2048) → 1024
```

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."

ldb *Accessor*

Syntax:

ldb *bytespec integer* → *byte*

(**setf** (**ldb** *bytespec place*) *new-byte*)

Pronunciation:

[ˈ**lidib**] or [ˈ**lidεb**] or [ˈ**el**ˈ**dē**ˈ**bē**]

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)
```

See Also:

byte, **byte-position**, **byte-size**, **dpb**

Notes:

```
(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."

ldb-test *Function*

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
```

See Also:

byte, **ldb**, **zerop**

Notes:

```
(ldb-test bytespec n) ≡
(not (zerop (ldb bytespec n))) ≡
(logtest (ldb bytespec -1) n)
```

mask-field *Accessor*

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
```

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))
```

most-positive-fixnum, most-negative-fixnum *Constant Variable*

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}.

decode-float, scale-float, float-radix, float-sign, float-digits, float-precision, integer-decode-float *Function*

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*.

decode-float, scale-float, float-radix, float-sign, ...

float-radix—an *integer*.

integer—a non-negative *integer*.

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

decode-float, scale-float, float-radix, float-sign, ...

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)
```

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
```

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*.

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)
                      (decode-float f)
   (* (scale-float signif expon) sign))
≡ f
```

float *Function*

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
```

See Also:

coerce

floatp *Function*

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
```

Notes:

```
(floatp object) ≡ (typep object 'float)
```

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 Variable*

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.

Notes:

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 Variable*

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* ϵ of the given format, such that the following expression is *true* when evaluated:

```
(not (= (float 1 ε) (+ (float 1 ε) ε)))
```

The value of each of the constants **short-float-negative-epsilon**, **single-float-negative-epsilon**, **double-float-negative-epsilon**, and **long-float-negative-epsilon** is the smallest positive *float* ϵ of the given format, such that the following expression is *true* when evaluated:

```
(not (= (float 1 ε) (- (float 1 ε) ε)))
```

arithmetic-error *Condition Type*

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**

arithmetic-error-operands, arithmetic-error-operation *Function*

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*.

See Also:

arithmetic-error, Chapter 9 (Conditions)

Notes:

division-by-zero *Condition Type*

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.

floating-point-invalid-operation *Condition Type*

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*.

floating-point-inexact *Condition Type*

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*.

floating-point-overflow *Condition Type*

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.

floating-point-underflow *Condition Type*

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.

Programming Language—Common Lisp

13. Characters

13.1 Character Concepts

13.1.1 Introduction to Characters

A ***character*** is an *object* that represents a unitary token (*e.g.*, a letter, a special symbol, or a "control character") in an aggregate quantity of text (*e.g.*, a *string* or a text *stream*).

Common Lisp allows an implementation to provide support for international language *characters* as well as *characters* used in specialized arenas (*e.g.*, mathematics).

The following figures contain lists of *defined names* applicable to *characters*.

Figure 13–1 lists some *defined names* relating to *character attributes* and *character predicates*.

alpha-char-p	**char-not-equal**	**char>**
alphanumericp	**char-not-greaterp**	**char>=**
both-case-p	**char-not-lessp**	**digit-char-p**
char-code-limit	**char/=**	**graphic-char-p**
char-equal	**char<**	**lower-case-p**
char-greaterp	**char<=**	**standard-char-p**
char-lessp	**char=**	**upper-case-p**

Figure 13–1. Character defined names – 1

Figure 13–2 lists some *character* construction and conversion *defined names*.

char-code	**char-name**	**code-char**
char-downcase	**char-upcase**	**digit-char**
char-int	**character**	**name-char**

Figure 13–2. Character defined names – 2

13.1.2 Introduction to Scripts and Repertoires

13.1.2.1 Character Scripts

A *script* is one of possibly several sets that form an *exhaustive partition* of the type **character**.

The number of such sets and boundaries between them is *implementation-defined*. Common Lisp does not require these sets to be *types*, but an *implementation* is permitted to define such *types* as an extension. Since no *character* from one *script* can ever be a member of another *script*, it is generally more useful to speak about *character repertoires*.

Although the term "*script*" is chosen for definitional compatibility with ISO terminology, no *conforming implementation* is required to use any particular *scripts* standardized by ISO or by any other standards organization.

Whether and how the *script* or *scripts* used by any given *implementation* are named is *implementation-dependent*.

13.1.2.2 Character Repertoires

A ***repertoire*** is a *type specifier* for a *subtype* of *type* **character**. This term is generally used when describing a collection of *characters* independent of their coding. *Characters* in *repertoires* are only identified by name, by *glyph*, or by character description.

A *repertoire* can contain *characters* from several *scripts*, and a *character* can appear in more than one *repertoire*.

For some examples of *repertoires*, see the coded character standards ISO 8859/1, ISO 8859/2, and ISO 6937/2. Note, however, that although the term "*repertoire*" is chosen for definitional compatibility with ISO terminology, no *conforming implementation* is required to use *repertoires* standardized by ISO or any other standards organization.

13.1.3 Character Attributes

Characters have only one *standardized attribute*: a *code*. A *character*'s *code* is a non-negative *integer*. This *code* is composed from a character *script* and a character label in an *implementation-dependent* way. See the *functions* **char-code** and **code-char**.

Additional, *implementation-defined attributes* of *characters* are also permitted so that, for example, two *characters* with the same *code* may differ in some other, *implementation-defined* way.

For any *implementation-defined attribute* there is a distinguished value called the ***null*** value for that *attribute*. A *character* for which each *implementation-defined attribute* has the null value for that *attribute* is called a *simple character*. If the *implementation* has no *implementation-defined attributes*, then all *characters* are *simple characters*.

13.1.4 Character Categories

There are several (overlapping) categories of *characters* that have no formally associated *type* but that are nevertheless useful to name. They include *graphic characters*, *alphabetic*$_1$ *characters*, *characters* with *case* (*uppercase* and *lowercase characters*), *numeric characters*, *alphanumeric characters*, and *digits* (in a given *radix*).

For each *implementation-defined attribute* of a *character*, the documentation for that *implementation* must specify whether *characters* that differ only in that *attribute* are permitted to differ in whether are not they are members of one of the aforementioned categories.

Note that these terms are defined independently of any special syntax which might have been enabled in the *current readtable*.

13.1.4.1 Graphic Characters

Characters that are classified as ***graphic***, or displayable, are each associated with a glyph, a visual representation of the *character*.

A *graphic character* is one that has a standard textual representation as a single *glyph*, such as `A` or `*` or `=`. *Space*, which effectively has a blank *glyph*, is defined to be a *graphic*.

Of the *standard characters*, *newline* is *non-graphic* and all others are *graphic*; see Section 2.1.3 (Standard Characters).

Characters that are not *graphic* are called ***non-graphic***. *Non-graphic characters* are sometimes informally called "formatting characters" or "control characters."

`#\Backspace`, `#\Tab`, `#\Rubout`, `#\Linefeed`, `#\Return`, and `#\Page`, if they are supported by the *implementation*, are *non-graphic*.

13.1.4.2 Alphabetic Characters

The *alphabetic*$_1$ *characters* are a subset of the *graphic characters*. Of the *standard characters*, only these are the *alphabetic*$_1$ *characters*:

`A B C D E F G H I J K L M N O P Q R S T U V W X Y Z`

`a b c d e f g h i j k l m n o p q r s t u v w x y z`

Any *implementation-defined character* that has *case* must be *alphabetic*$_1$. For each *implementation-defined graphic character* that has no *case*, it is *implementation-defined* whether that *character* is *alphabetic*$_1$.

13.1.4.3 Characters With Case

The *characters* with *case* are a subset of the *alphabetic*$_1$ *characters*. A *character* with *case* has the property of being either *uppercase* or *lowercase*. Every *character* with *case* is in one-to-one correspondence with some other *character* with the opposite *case*.

13.1.4.3.1 Uppercase Characters

An uppercase *character* is one that has a corresponding *lowercase character* that is *different* (and can be obtained using **char-downcase**).

Of the *standard characters*, only these are *uppercase characters*:

`A B C D E F G H I J K L M N O P Q R S T U V W X Y Z`

13.1.4.3.2 Lowercase Characters

A lowercase *character* is one that has a corresponding *uppercase character* that is *different* (and can be obtained using **char-upcase**).

Of the *standard characters*, only these are *lowercase characters*:

`a b c d e f g h i j k l m n o p q r s t u v w x y z`

13.1.4.3.3 Corresponding Characters in the Other Case

The *uppercase standard characters* `A` through `Z` mentioned above respectively correspond to the *lowercase standard characters* `a` through `z` mentioned above. For example, the *uppercase character* `E` corresponds to the *lowercase character* `e`, and vice versa.

13.1.4.3.4 Case of Implementation-Defined Characters

An *implementation* may define that other *implementation-defined graphic characters* have *case*. Such definitions must always be done in pairs—one *uppercase character* in one-to-one *correspondence* with one *lowercase character*.

13.1.4.4 Numeric Characters

The *numeric characters* are a subset of the *graphic characters*. Of the *standard characters*, only these are *numeric characters*:

`0 1 2 3 4 5 6 7 8 9`

For each *implementation-defined graphic character* that has no *case*, the *implementation* must define whether or not it is a *numeric character*.

13.1.4.5 Alphanumeric Characters

The set of *alphanumeric characters* is the union of the set of *alphabetic*$_1$ *characters* and the set of *numeric characters*.

13.1.4.6 Digits in a Radix

What qualifies as a *digit* depends on the *radix* (an *integer* between `2` and `36`, inclusive). The potential *digits* are:

```
0 1 2 3 4 5 6 7 8 9 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

Their respective weights are `0`, `1`, `2`, ... `35`. In any given radix n, only the first n potential *digits* are considered to be *digits*. For example, the digits in radix `2` are `0` and `1`, the digits in radix `10` are `0` through `9`, and the digits in radix `16` are `0` through `F`.

Case is not significant in *digits*; for example, in radix `16`, both `F` and `f` are *digits* with weight `15`.

13.1.5 Identity of Characters

Two *characters* that are **eql**, **char=**, or **char-equal** are not necessarily **eq**.

13.1.6 Ordering of Characters

The total ordering on *characters* is guaranteed to have the following properties:

- If two *characters* have the same *implementation-defined attributes*, then their ordering by **char<** is consistent with the numerical ordering by the predicate `<` on their code *attributes*.

- If two *characters* differ in any *attribute*, then they are not **char=**.

- The total ordering is not necessarily the same as the total ordering on the *integers* produced by applying **char-int** to the *characters*.

- While *alphabetic*$_1$ *standard characters* of a given *case* must obey a partial ordering, they need not be contiguous; it is permissible for *uppercase* and *lowercase characters* to be interleaved. Thus `(char<= #\a x #\z)` is not a valid way of determining whether or not `x` is a *lowercase character*.

Of the *standard characters*, those which are *alphanumeric* obey the following partial ordering:

```
A<B<C<D<E<F<G<H<I<J<K<L<M<N<O<P<Q<R<S<T<U<V<W<X<Y<Z
a<b<c<d<e<f<g<h<i<j<k<l<m<n<o<p<q<r<s<t<u<v<w<x<y<z
0<1<2<3<4<5<6<7<8<9
either 9<A or Z<0
either 9<a or z<0
```

This implies that, for *standard characters*, *alphabetic*$_1$ ordering holds within each *case* (*uppercase* and *lowercase*), and that the *numeric characters* as a group are not interleaved with *alphabetic characters*. However, the ordering or possible interleaving of *uppercase characters* and *lowercase characters* is *implementation-defined*.

13.1.7 Character Names

The following *character names* must be present in all *conforming implementations*:

`Newline`

The character that represents the division between lines. An implementation must translate between `#\Newline`, a single-character representation, and whatever external representation(s) may be used.

`Space`

The space or blank character.

The following names are *semi-standard*; if an *implementation* supports them, they should be used for the described *characters* and no others.

`Rubout`

The rubout or delete character.

`Page`

The form-feed or page-separator character.

`Tab`

The tabulate character.

`Backspace`

The backspace character.

`Return`

The carriage return character.

`Linefeed`

The line-feed character.

In some *implementations*, one or more of these *character names* might denote a *standard character*; for example, `#\Linefeed` and `#\Newline` might be the *same character* in some *implementations*.

13.1.8 Treatment of Newline during Input and Output

When the character `#\Newline` is written to an output file, the implementation must take the appropriate action to produce a line division. This might involve writing out a record or translating `#\Newline` to a CR/LF sequence. When reading, a corresponding reverse transformation must take place.

13.1.9 Character Encodings

A *character* is sometimes represented merely by its *code*, and sometimes by another *integer* value which is composed from the *code* and all *implementation-defined attributes* (in an *implementation-defined* way that might vary between *Lisp images* even in the same *implementation*). This *integer*, returned by the function **char-int**, is called the character's "encoding." There is no corresponding function from a character's encoding back to the *character*, since its primary intended uses include things like hashing where an inverse operation is not really called for.

13.1.10 Documentation of Implementation-Defined Scripts

An *implementation* must document the *character scripts* it supports. For each *character script* supported, the documentation must describe at least the following:

- Character labels, glyphs, and descriptions. Character labels must be uniquely named using only Latin capital letters A–Z, hyphen (-), and digits 0–9.
- Reader canonicalization. Any mechanisms by which **read** treats *different* characters as equivalent must be documented.
- The impact on **char-upcase**, **char-downcase**, and the case-sensitive *format directives*. In particular, for each *character* with *case*, whether it is *uppercase* or *lowercase*, and which *character* is its equivalent in the opposite case.
- The behavior of the case-insensitive *functions* **char-equal**, **char-not-equal**, **char-lessp**, **char-greaterp**, **char-not-greaterp**, and **char-not-lessp**.
- The behavior of any *character predicates*; in particular, the effects of **alpha-char-p**, **lower-case-p**, **upper-case-p**, **both-case-p**, **graphic-char-p**, and **alphanumericp**.
- The interaction with file I/O, in particular, the supported coded character sets (for example, ISO8859/1-1987) and external encoding schemes supported are documented.

character *System Class*

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:

Section 13.1 (Character Concepts), Section 2.4.8.1 (Sharpsign Backslash), Section 22.1.3.2 (Printing Characters)

base-char *Type*

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**.

standard-char *Type*

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:

Section 2.1.3 (Standard Characters)

extended-char *Type*

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**.

char=, char/=, char<, char>, char<=, char>=, char-equal, char-not-equal, char-lessp, char-greaterp, char-not-greaterp, char-not-lessp *Function*

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*
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
(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
```

Exceptional Situations:

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

See Also:

Section 2.1 (Character Syntax), Section 13.1.10 (Documentation of Implementation-Defined Scripts)

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

character *Function*

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:

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

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)
```

characterp *Function*

Syntax:

characterp *object* → *generalized-boolean*

Arguments and Values:

object—an *object*.

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
```

See Also:

character (*type* and *function*), **typep**

Notes:

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

alpha-char-p *Function*

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
;; This next example presupposes an implementation
;; in which #\α is a defined character.
(alpha-char-p #\α) → 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)

alphanumericp *Function*

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:

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))))
```

digit-char *Function*

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
```

See Also:

digit-char-p, **graphic-char-p**, Section 2.1 (Character Syntax)

Notes:

digit-char-p *Function*

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:

```
(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 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 readtable*.)

See Also:

alphanumericp

Notes:

Digits are *graphic characters*.

---

graphic-char-p *Function*

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
```

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)

standard-char-p *Function*

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:

```
(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
```

Exceptional Situations:

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

char-upcase, char-downcase *Function*

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
```

Exceptional Situations:

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**.

upper-case-p, lower-case-p, both-case-p *Function*

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
```

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)

char-code *Function*

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
```

Exceptional Situations:

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

See Also:

char-code-limit

char-int *Function*

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
```

See Also:

char-code

code-char *Function*

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.

See Also:

char-code

Notes:

char-code-limit *Constant Variable*

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*.

char-name

char-name *Function*

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"
```

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:

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).

name-char *Function*

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
```

Exceptional Situations:

Should signal an error of *type* **type-error** if *name* is not a *string designator*.

See Also:

char-name

Programming Language—Common Lisp

14. Conses

14.1 Cons Concepts

A ***cons*** is a compound data *object* having two components called the *car* and the *cdr*.

car	**cons**	**rplacd**
cdr	**rplaca**	

Figure 14–1. Some defined names relating to conses.

Depending on context, a group of connected *conses* can be viewed in a variety of different ways. A variety of operations is provided to support each of these various views.

14.1.1 Conses as Trees

A ***tree*** is a binary recursive data structure made up of *conses* and *atoms*: the *conses* are themselves also *trees* (sometimes called "subtrees" or "branches"), and the *atoms* are terminal nodes (sometimes called ***leaves***). Typically, the *leaves* represent data while the branches establish some relationship among that data.

caaaar	**caddar**	**cdar**	**nsubst**
caaadr	**cadddr**	**cddaar**	**nsubst-if**
caaar	**caddr**	**cddadr**	**nsubst-if-not**
caadar	**cadr**	**cddar**	**nthcdr**
caaddr	**cdaaar**	**cdddar**	**sublis**
caadr	**cdaadr**	**cddddr**	**subst**
caar	**cdaar**	**cdddr**	**subst-if**
cadaar	**cdadar**	**cddr**	**subst-if-not**
cadadr	**cdaddr**	**copy-tree**	**tree-equal**
cadar	**cdadr**	**nsublis**	

Figure 14–2. Some defined names relating to trees.

14.1.1.1 General Restrictions on Parameters that must be Trees

Except as explicitly stated otherwise, for any *standardized function* that takes a *parameter* that is required to be a *tree*, the consequences are undefined if that *tree* is circular.

14.1.2 Conses as Lists

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 *list* terminated by the *empty list*. The *empty list* is a *proper list*, but is not a *cons*.

An ***improper list*** is a *list* that is not a *proper list*; that is, it is a *circular list* or a *dotted list*.

A ***dotted list*** is a *list* that has a terminating *atom* that is not the *empty list*. A *non-nil atom* by itself is not considered to be a *list* of any kind—not even a *dotted 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*.

append	**last**	**nbutlast**	**rest**
butlast	**ldiff**	**nconc**	**revappend**
copy-alist	**list**	**ninth**	**second**
copy-list	**list***	**nreconc**	**seventh**
eighth	**list-length**	**nth**	**sixth**
endp	**make-list**	**nthcdr**	**tailp**
fifth	**member**	**pop**	**tenth**
first	**member-if**	**push**	**third**
fourth	**member-if-not**	**pushnew**	

Figure 14–3. Some defined names relating to lists.

14.1.2.1 Lists as Association Lists

An ***association list*** is a *list* of *conses* representing an association of *keys* with *values*, where the *car* of each *cons* is the *key* and the *cdr* is the *value* associated with that *key*.

acons	**assoc-if**	**pairlis**	**rassoc-if**
assoc	**assoc-if-not**	**rassoc**	**rassoc-if-not**

Figure 14–4. Some defined names related to assocation lists.

14.1.2.2 Lists as Sets

Lists are sometimes viewed as sets by considering their elements unordered and by assuming there is no duplication of elements.

adjoin	**nset-difference**	**set-difference**	**union**
intersection	**nset-exclusive-or**	**set-exclusive-or**	
nintersection	**nunion**	**subsetp**	

Figure 14–5. Some defined names related to sets.

14.1.2.3 General Restrictions on Parameters that must be Lists

Except as explicitly specified otherwise, any *standardized function* that takes a *parameter* that is required to be a *list* should be prepared to signal an error of *type* **type-error** if the *value* received is a *dotted list*.

Except as explicitly specified otherwise, for any *standardized function* that takes a *parameter* that is required to be a *list*, the consequences are undefined if that *list* is *circular*.

list *System Class*

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:

Section 2.4.1 (Left-Parenthesis), Section 22.1.3.5 (Printing Lists and Conses)

null *System Class*

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:

Section 2.3.4 (Symbols as Tokens), Section 2.4.1 (Left-Parenthesis), Section 22.1.3.3 (Printing Symbols)

cons *System Class*

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:

Section 2.4.1 (Left-Parenthesis), Section 22.1.3.5 (Printing Lists and Conses)

atom *Type*

Supertypes:

atom, **t**

Description:

It is equivalent to (not cons).

cons *Function*

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)
```

See Also:

list

Notes:

If *object-2* is a *list*, **cons** can be thought of as producing a new *list* which is like it but has *object-1* prepended.

consp *Function*

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
```

See Also:

listp

Notes:

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

atom *Function*

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
```

Notes:

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

rplaca, rplacd *Function*

Syntax:

rplaca *cons object* → *cons*
rplacd *cons object* → *cons*

Pronunciation:

rplaca: [ˌ**rē** ˈ**plak**ϵ] or [ˌ**r**ϵ ˈ**plak**ϵ]

rplacd: [ˌ**rē** ˈ**plakd**ϵ] or [ˌ**r**ϵ ˈ**plakd**ϵ] or [ˌ**rē** ˈ**plakdē**] or [ˌ**r**ϵ ˈ**plakdē**]

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.

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

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, cdddar, cddddr *Accessor*

Syntax:

car *x*	→ *object*	(**setf** (**car** *x*) *new-object*)
cdr *x*	→ *object*	(**setf** (**cdr** *x*) *new-object*)
caar *x*	→ *object*	(**setf** (**caar** *x*) *new-object*)
cadr *x*	→ *object*	(**setf** (**cadr** *x*) *new-object*)
cdar *x*	→ *object*	(**setf** (**cdar** *x*) *new-object*)
cddr *x*	→ *object*	(**setf** (**cddr** *x*) *new-object*)
caaar *x*	→ *object*	(**setf** (**caaar** *x*) *new-object*)
caadr *x*	→ *object*	(**setf** (**caadr** *x*) *new-object*)
cadar *x*	→ *object*	(**setf** (**cadar** *x*) *new-object*)
caddr *x*	→ *object*	(**setf** (**caddr** *x*) *new-object*)
cdaar *x*	→ *object*	(**setf** (**cdaar** *x*) *new-object*)
cdadr *x*	→ *object*	(**setf** (**cdadr** *x*) *new-object*)
cddar *x*	→ *object*	(**setf** (**cddar** *x*) *new-object*)
cdddr *x*	→ *object*	(**setf** (**cdddr** *x*) *new-object*)
caaaar *x*	→ *object*	(**setf** (**caaaar** *x*) *new-object*)
caaadr *x*	→ *object*	(**setf** (**caaadr** *x*) *new-object*)
caadar *x*	→ *object*	(**setf** (**caadar** *x*) *new-object*)
caaddr *x*	→ *object*	(**setf** (**caaddr** *x*) *new-object*)
cadaar *x*	→ *object*	(**setf** (**cadaar** *x*) *new-object*)
cadadr *x*	→ *object*	(**setf** (**cadadr** *x*) *new-object*)
caddar *x*	→ *object*	(**setf** (**caddar** *x*) *new-object*)
cadddr *x*	→ *object*	(**setf** (**cadddr** *x*) *new-object*)
cdaaar *x*	→ *object*	(**setf** (**cdaaar** *x*) *new-object*)
cdaadr *x*	→ *object*	(**setf** (**cdaadr** *x*) *new-object*)
cdadar *x*	→ *object*	(**setf** (**cdadar** *x*) *new-object*)
cdaddr *x*	→ *object*	(**setf** (**cdaddr** *x*) *new-object*)
cddaar *x*	→ *object*	(**setf** (**cddaar** *x*) *new-object*)
cddadr *x*	→ *object*	(**setf** (**cddadr** *x*) *new-object*)
cdddar *x*	→ *object*	(**setf** (**cdddar** *x*) *new-object*)
cddddr *x*	→ *object*	(**setf** (**cddddr** *x*) *new-object*)

Pronunciation:

cadr: [ˈkaˌdɛr]

car, cdr, caar, cadr, cdar, cddr, caaar, caadr, cadar, ...

caddr: [ˈ**kadə,dər**] or [ˈ**ka,dudər**]

cdr: [ˈ**ku,dər**]

cddr: [ˈ**kudə,dər**] or [ˈ**kə,dudər**]

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. The order in which the `A`'s and `D`'s appear is the inverse of the order in which the corresponding operations are performed. Figure 14–6 defines the relationships precisely.

car, cdr, caar, cadr, cdar, cddr, caaar, caadr, cadar, ...

This *place* ...	**Is equivalent to this** *place* ...
(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*)))
(caadr *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*)))
(cdddr *x*)	(cdr (cdr (cdr *x*)))
(caaaar *x*)	(car (car (car (car *x*))))
(caaadr *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*))))
(cdaadr *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*))))
(cddadr *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)
```

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**, . . . **cddddr**) 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))))
```

copy-tree *Function*

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
 (setf (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))
```

See Also:

tree-equal

sublis, nsublis *Function*

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*.

sublis, nsublis

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.

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")))

 (nsublis '((t . 'temp))
           tree1
           :key #'(lambda (x) (or (atom x) (< (list-length x) 3))))
→ ((QUOTE TEMP) (QUOTE TEMP) QUOTE TEMP)
```

Side Effects:

nsublis modifies *tree*.

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.*, **nsublis**) potentially change the path that is being traversed, their effects in the presence of shared or circular structure 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))
         (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))
```

subst, subst-if, subst-if-not, nsubst, nsubst-if, nsubst-if-not *Function*

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*

nsubst *new old tree* `&key` *key test test-not* → *new-tree*

nsubst-if *new predicate tree* `&key` *key* → *new-tree*

nsubst-if-not *new predicate tree* `&key` *key* → *new-tree*

Arguments and Values:

new—an *object*.

old—an *object*.

subst, subst-if, subst-if-not, nsubst, nsubst-if, ...

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*.

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))))))
```

tree-equal

tree-equal *Function*

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*.

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
```

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.

copy-list *Function*

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))
(setf (caadr lst) "two") → "two"
lst → ("one" ("two" 3))
slst → ("one" ("two" 3))
clst → (1 ("two" 3))
```

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**.

list, list*

list, list* *Function*

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***.

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)
```

See Also:

cons

Notes:

(list* *x*) ≡ *x*

list-length *Function*

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
```

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))))
```

listp *Function*

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
```

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))
```

make-list *Function*

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
```

Exceptional Situations:

Should signal an error of *type* **type-error** if *size* is not a non-negative *integer*.

See Also:

cons, **list**

push *Macro*

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.

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*.

pop *Macro*

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:

```
(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.

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.

first, second, third, fourth, fifth, sixth, seventh, eighth, ninth, tenth *Accessor*

Syntax:

first *list*	→ *object*	(**setf** (**first** *list*) *new-object*)
second *list*	→ *object*	(**setf** (**second** *list*) *new-object*)
third *list*	→ *object*	(**setf** (**third** *list*) *new-object*)
fourth *list*	→ *object*	(**setf** (**fourth** *list*) *new-object*)
fifth *list*	→ *object*	(**setf** (**fifth** *list*) *new-object*)
sixth *list*	→ *object*	(**setf** (**sixth** *list*) *new-object*)
seventh *list*	→ *object*	(**setf** (**seventh** *list*) *new-object*)
eighth *list*	→ *object*	(**setf** (**eighth** *list*) *new-object*)
ninth *list*	→ *object*	(**setf** (**ninth** *list*) *new-object*)
tenth *list*	→ *object*	(**setf** (**tenth** *list*) *new-object*)

Arguments and Values:

list—a *list*, which might be a *dotted list* or a *circular list*.

first, second, third, fourth, fifth, sixth, seventh, ...

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 (cddr list))
(fourth list)   ≡ (car (cdddr list))
(fifth list)    ≡ (car (cddddr list))
(sixth list)    ≡ (car (cdr (cddddr list)))
(seventh list)  ≡ (car (cddr (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
 (setf (fourth lst) "four") → "four"
 lst → (1 2 3 "four" ((V)) VI 7 8 9 10)
```

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)
```

nth *Accessor*

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)
```

See Also:

elt, **first**, **nthcdr**

endp *Function*

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
```

Exceptional Situations:

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

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*.

null *Function*

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
```

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 '())
```

nconc *Function*

Syntax:

nconc &rest *lists* → *concatenated-list*

Arguments and Values:

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.

See Also:

append, **concatenate**

append *Function*

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
```

See Also:

nconc, **concatenate**

revappend, nreconc *Function*

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*.

revappend, nreconc

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)))
   (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)
```

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)
```

butlast, nbutlast *Function*

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
```

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*.

Notes:

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

last *Function*

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)
```

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))))
```

ldiff, tailp *Function*

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*.

ldiff, tailp

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)
▷  object=X            NIL                    (A B C . D)
→ NIL
```

Side Effects:

Neither **ldiff** nor **tailp** modifies either of its *arguments*.

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)))))
```

nthcdr *Function*

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.
```

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**

rest *Accessor*

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")
```

See Also:

cdr, **nthcdr**

Notes:

rest is often preferred stylistically over **cdr** when the argument is to being subjectively viewed as a *list* rather than as a *cons*.

member, member-if, member-if-not *Function*

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:

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)
```

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**).

mapc, mapcar, mapcan, mapl, maplist, mapcon

Function

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*

mapc, mapcar, mapcan, mapl, maplist, mapcon

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 **coerce**d 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.

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))
```

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)

acons *Function*

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")
```

See Also:

assoc, **pairlis**

Notes:

(acons *key datum alist*) ≡ (cons (cons *key datum*) *alist*)

assoc, assoc-if, assoc-if-not *Function*

Syntax:

assoc *item alist* &key *key test test-not* → *entry*

assoc-if *predicate alist* &key *key* → *entry*

assoc, assoc-if, assoc-if-not

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.

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)
```

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**.

copy-alist *Function*

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"))
```

See Also:

copy-list

pairlis *Function*

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))
```

or

```
((two . 2) (one . 1) (three . 3) (four . 19))
```

Examples:

```
 (setq keys '(1 2 3)
        data '("one" "two" "three")
        alist '((4 . "four"))) → ((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"))
```

Exceptional Situations:

Should be prepared to signal an error of *type* **type-error** if *keys* and *data* are not *proper lists*.

See Also:

acons

rassoc, rassoc-if, rassoc-if-not *Function*

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)
```

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**.

get-properties *Function*

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
(setf (getf x 'prop1) 'val1) → VAL1
(eq (getf x 'prop1) 'val1) → true
(get-properties x *indicator-list*) → PROP1, VAL1, (PROP1 VAL1)
x → (PROP1 VAL1)
```

See Also:

get, **getf**

getf *Accessor*

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*.

getf

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 assocation 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:

```
 (setq x '()) → NIL
 (getf x 'prop1) → NIL
 (getf x 'prop1 7) → 7
 (getf x 'prop1) → NIL
 (setf (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)
```

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)
```

remf *Macro*

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.

See Also:

remprop, **getf**

intersection, nintersection *Function*

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.

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.

intersection, nintersection

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 *satifies 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*.

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.

Since the **nintersection** side effect is not required, it should not be used in for-effect-only positions in portable code.

adjoin *Function*

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))
```

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))
```

pushnew *Macro*

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 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.

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`.

set-difference, nset-difference *Function*

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:

```
 (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*.

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.

set-exclusive-or, nset-exclusive-or *Function*

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*.

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.

subsetp *Function*

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
(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
```

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.

union, nunion *Function*

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.

union, nunion

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*.

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.

Programming Language—Common Lisp

15. Arrays

15.1 Array Concepts

15.1.1 Array Elements

An *array* contains a set of *objects* called *elements* that can be referenced individually according to a rectilinear coordinate system.

15.1.1.1 Array Indices

An *array element* is referred to by a (possibly empty) series of indices. The length of the series must equal the *rank* of the *array*. Each index must be a non-negative *fixnum* less than the corresponding *array dimension*. *Array* indexing is zero-origin.

15.1.1.2 Array Dimensions

An axis of an *array* is called a ***dimension***.

Each *dimension* is a non-negative *fixnum*; if any dimension of an *array* is zero, the *array* has no elements. It is permissible for a *dimension* to be zero, in which case the *array* has no elements, and any attempt to *access* an *element* is an error. However, other properties of the *array*, such as the *dimensions* themselves, may be used.

15.1.1.2.1 Implementation Limits on Individual Array Dimensions

An *implementation* may impose a limit on *dimensions* of an *array*, but there is a minimum requirement on that limit. See the *variable* **array-dimension-limit**.

15.1.1.3 Array Rank

An *array* can have any number of *dimensions* (including zero). The number of *dimensions* is called the ***rank***.

If the rank of an *array* is zero then the *array* is said to have no *dimensions*, and the product of the dimensions (see **array-total-size**) is then 1; a zero-rank *array* therefore has a single element.

15.1.1.3.1 Vectors

An *array* of *rank* one (*i.e.*, a one-dimensional *array*) is called a ***vector***.

15.1.1.3.1.1 Fill Pointers

A ***fill pointer*** is a non-negative *integer* no larger than the total number of *elements* in a *vector*. Not all *vectors* have *fill pointers*. See the *functions* **make-array** and **adjust-array**.

An *element* of a *vector* is said to be ***active*** if it has an index that is greater than or equal to zero, but less than the *fill pointer* (if any). For an *array* that has no *fill pointer*, all *elements* are considered *active*.

Only *vectors* may have *fill pointers*; multidimensional *arrays* may not. A multidimensional *array* that is displaced to a *vector* that has a *fill pointer* can be created.

15.1.1.3.2 Multidimensional Arrays

15.1.1.3.2.1 Storage Layout for Multidimensional Arrays

Multidimensional *arrays* store their components in row-major order; that is, internally a multidimensional *array* is stored as a one-dimensional *array*, with the multidimensional index sets ordered lexicographically, last index varying fastest.

15.1.1.3.2.2 Implementation Limits on Array Rank

An *implementation* may impose a limit on the *rank* of an *array*, but there is a minimum requirement on that limit. See the *variable* **array-rank-limit**.

15.1.2 Specialized Arrays

An *array* can be a *general array*, meaning each *element* may be any *object*, or it may be a *specialized array*, meaning that each *element* must be of a restricted *type*.

The phrasing "an *array specialized* to *type* ⟨⟨*type*⟩⟩" is sometimes used to emphasize the *element type* of an *array*. This phrasing is tolerated even when the ⟨⟨*type*⟩⟩ is **t**, even though an *array specialized* to *type t* is a *general array*, not a *specialized array*.

Figure 15–1 lists some *defined names* that are applicable to *array* creation, *access*, and information operations.

adjust-array	**array-has-fill-pointer-p**	**make-array**
adjustable-array-p	**array-in-bounds-p**	**svref**
aref	**array-rank**	**upgraded-array-element-type**
array-dimension	**array-rank-limit**	**upgraded-complex-part-type**
array-dimension-limit	**array-row-major-index**	**vector**
array-dimensions	**array-total-size**	**vector-pop**
array-displacement	**array-total-size-limit**	**vector-push**
array-element-type	**fill-pointer**	**vector-push-extend**

Figure 15–1. General Purpose Array-Related Defined Names

15.1.2.1 Array Upgrading

The ***upgraded array element type*** of a *type* T_1 is a *type* T_2 that is a *supertype* of T_1 and that is used instead of T_1 whenever T_1 is used as an *array element type* for object creation or type discrimination.

During creation of an *array*, the *element type* that was requested is called the ***expressed array element type***. The *upgraded array element type* of the *expressed array element type* becomes the ***actual array element type*** of the *array* that is created.

Type upgrading implies a movement upwards in the type hierarchy lattice. A *type* is always a *subtype* of its *upgraded array element type*. Also, if a *type* T_x is a *subtype* of another *type* T_y, then the *upgraded array element type* of T_x must be a *subtype* of the *upgraded array element type* of T_y. Two *disjoint types* can be *upgraded* to the same *type*.

The *upgraded array element type* T_2 of a *type* T_1 is a function only of T_1 itself; that is, it is independent of any other property of the *array* for which T_2 will be used, such as *rank*, *adjustability*, *fill pointers*, or displacement. The *function* **upgraded-array-element-type** can be used by *conforming programs* to predict how the *implementation* will *upgrade* a given *type*.

15.1.2.2 Required Kinds of Specialized Arrays

Vectors whose *elements* are restricted to *type* **character** or a *subtype* of **character** are called ***strings***. *Strings* are of *type* **string**. Figure 15–2 lists some *defined names* related to *strings*.

Strings are *specialized arrays* and might logically have been included in this chapter. However, for purposes of readability most information about *strings* does not appear in this chapter; see instead Chapter 16 (Strings).

char	**string-equal**	**string-upcase**
make-string	**string-greaterp**	**string/=**
nstring-capitalize	**string-left-trim**	**string<**
nstring-downcase	**string-lessp**	**string<=**
nstring-upcase	**string-not-equal**	**string=**
schar	**string-not-greaterp**	**string>**
string	**string-not-lessp**	**string>=**
string-capitalize	**string-right-trim**	
string-downcase	**string-trim**	

Figure 15–2. Operators that Manipulate Strings

Vectors whose *elements* are restricted to *type* **bit** are called ***bit vectors***. *Bit vectors* are of *type* **bit-vector**. Figure 15–3 lists some *defined names* for operations on *bit arrays*.

bit	**bit-ior**	**bit-orc2**
bit-and	**bit-nand**	**bit-xor**
bit-andc1	**bit-nor**	**sbit**
bit-andc2	**bit-not**	
bit-eqv	**bit-orc1**	

Figure 15–3. Operators that Manipulate Bit Arrays

array *System Class*

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* $\{i_0, i_1, \ldots, i_{r-1}\}$ to corresponding *elements* of the *array*, where $0 \le i_j < d_j$, r is the rank of the array, and d_j 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* 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*.

simple-array *Type*

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:

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**.

vector *System Class*

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:

Section 15.1.2.2 (Required Kinds of Specialized Arrays), Section 2.4.8.3 (Sharpsign Left-Parenthesis), Section 22.1.3.7 (Printing Other Vectors), Section 2.4.8.12 (Sharpsign A)

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**.

simple-vector *Type*

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*`))`.

bit-vector *System Class*

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:

Section 2.4.8.4 (Sharpsign Asterisk), Section 22.1.3.6 (Printing Bit Vectors), Section 15.1.2.2 (Required Kinds of Specialized Arrays)

simple-bit-vector *Type*

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*.

make-array *Function*

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*.

make-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*.

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

make-array

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*.

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
```

See Also:

adjustable-array-p, **aref**, **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*.

adjust-array

adjust-array *Function*

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.

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)
```

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.

adjust-array

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
 (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 ))
```

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**

adjustable-array-p *Function*

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
```

Exceptional Situations:

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

See Also:

adjust-array, **make-array**

aref *Accessor*

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 then 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
```

```
(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
```

See Also:

bit, **char**, **elt**, **row-major-aref**, **svref**, Section 3.2.1 (Compiler Terminology)

array-dimension *Function*

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.

See Also:

array-dimensions, **length**

Notes:

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

array-dimensions *Function*

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)
```

Exceptional Situations:

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

See Also:

array-dimension

array-element-type *Function*

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**

array-has-fill-pointer-p *Function*

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
```

Exceptional Situations:

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

See Also:

make-array, **fill-pointer**

Notes:

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.

array-displacement *Function*

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>
 (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
```

Exceptional Situations:

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

See Also:

make-array

array-in-bounds-p *Function*

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
```

See Also:

array-dimensions

Notes:

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

array-rank *Function*

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
```

Exceptional Situations:

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

See Also:

array-rank-limit, **make-array**

array-row-major-index *Function*

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:

```
(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
```

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))))
```

array-total-size *Function*

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
```

Exceptional Situations:

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))
```

arrayp *Function*

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
```

See Also:

typep

Notes:

```
(arrayp object) ≡ (typep object 'array)
```

fill-pointer *Accessor*

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
(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)
```

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*.

row-major-aref *Accessor*

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**.

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))
```

upgraded-array-element-type *Function*

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*.

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)))
```

array-dimension-limit *Constant Variable*

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*.

See Also:

make-array

array-rank-limit *Constant Variable*

Constant Value:

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*.

See Also:

make-array

array-total-size-limit *Constant Variable*

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.

See Also:

make-array, **array-element-type**

simple-vector-p *Function*

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
```

See Also:

simple-vector

Notes:

```
(simple-vector-p object) ≡ (typep object 'simple-vector)
```

svref *Accessor*

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
(svref v 2) → SIRENS
(setf (svref v 1) 'newcomer) → NEWCOMER
v → #(1 NEWCOMER SIRENS)
```

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)
```

vector *Function*

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
```

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))
```

vector-pop *Function*

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**

vector-push, vector-push-extend *Function*

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 `(>= (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.

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**

vectorp *Function*

Syntax:

vectorp *object* → *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 "aaaaaa") → true
(vectorp (make-array 6 :fill-pointer t)) → true
(vectorp (make-array '(2 3 4))) → false
(vectorp #*11) → true
(vectorp #b11) → false
```

Notes:

```
(vectorp object) ≡ (typep object 'vector)
```

bit, sbit *Accessor*

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
                             :element-type 'bit
                             :initial-element 1))
      3) → 1
(setf (bit ba 3) 0) → 0
(bit ba 3) → 0
(sbit ba 5) → 1
(setf (sbit ba 5) 1) → 1
```

```
(sbit ba 5) → 1
```

See Also:

aref, 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*.

bit-and, bit-andc1, bit-andc2, bit-eqv, bit-ior, bit-nand, bit-nor, bit-not, bit-orc1, bit-orc2, bit-xor

Function

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*
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.

Figure 15–4 indicates the logical operation performed by each of the *functions*.

Function	Operation
bit-and	and
bit-eqv	equivalence (exclusive nor)
bit-not	complement
bit-ior	inclusive or
bit-xor	exclusive or
bit-nand	complement of *bit-array1* and *bit-array2*
bit-nor	complement of *bit-array1* or *bit-array2*
bit-andc1	and complement of *bit-array1* with *bit-array2*
bit-andc2	and *bit-array1* with complement of *bit-array2*
bit-orc1	or complement of *bit-array1* with *bit-array2*
bit-orc2	or *bit-array1* with complement of *bit-array2*

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

Examples:

```
 (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
```

See Also:

lognot, **logand**

bit-vector-p *Function*

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
```

See Also:

typep

Notes:

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

simple-bit-vector-p *Function*

Syntax:

simple-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* **simple-bit-vector**; otherwise, returns *false*.

simple-bit-vector-p

Examples:

```
(simple-bit-vector-p (make-array 6)) → false
(simple-bit-vector-p #*) → true
```

See Also:

simple-vector-p

Notes:

```
(simple-bit-vector-p object) ≡ (typep object 'simple-bit-vector)
```

Programming Language—Common Lisp

16. Strings

16.1 String Concepts

16.1.1 Implications of Strings Being Arrays

Since all *strings* are *arrays*, all rules which apply generally to *arrays* also apply to *strings*. See Section 15.1 (Array Concepts).

For example, *strings* can have *fill pointers*, and *strings* are also subject to the rules of *element type upgrading* that apply to *arrays*.

16.1.2 Subtypes of STRING

All functions that operate on *strings* will operate on *subtypes* of *string* as well.

However, the consequences are undefined if a *character* is inserted into a *string* for which the *element type* of the *string* does not include that *character*.

string *System Class*

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:

Section 16.1 (String Concepts), Section 2.4.5 (Double-Quote), Section 22.1.3.4 (Printing Strings)

base-string *Type*

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*.

simple-string *Type*

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*.

simple-base-string *Type*

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*.

simple-string-p *Function*

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
```

Notes:

```
(simple-string-p object) ≡ (typep object 'simple-string)
```

char, schar *Accessor*

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
(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 → "AAACBD"
```

See Also:

aref, **elt**, Section 3.2.1 (Compiler Terminology)

Notes:

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

string *Function*

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"
```

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*.

string-upcase, string-downcase, string-capitalize, nstring-upcase, nstring-downcase, nstring-capitalize *Function*

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*.

Description:

string-upcase, **string-downcase**, **string-capitalize**, **nstring-upcase**, **nstring-downcase**, **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-upcase, string-downcase, string-capitalize, ...

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, foo16c") → "Pipe 13a, Foo16c"

 (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*.

See Also:

char-upcase, **char-downcase**

Notes:

The result is always of the same length as *string*.

string-trim, string-left-trim, string-right-trim *Function*

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"
 (string-trim " (*)" " ( *three (silly) words* ) ")
→ "three (silly) words"

 (string-left-trim "abc" "labcabcabc") → "labcabcabc"
 (string-left-trim " (*)" " ( *three (silly) words* ) ")
→ "three (silly) words* ) "

 (string-right-trim " (*)" " ( *three (silly) words* ) ")
→ " ( *three (silly) words"
```

Affected By:

The *implementation*.

string=, string/=, string<, string>, string<=, string>=, string-equal, string-not-equal, string-lessp, string-greaterp, string-not-greaterp, string-not-lessp *Function*

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*

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>= "aaaaa" "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
```

See Also:

char=

Notes:

equal calls **string=** if applied to two *strings*.

stringp *Function*

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
```

See Also:

typep, **string** (*type*)

Notes:

```
(stringp object) ≡ (typep object 'string)
```

make-string *Function*

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*.

Programming Language—Common Lisp

17. Sequences

17.1 Sequence Concepts

A ***sequence*** is an ordered collection of *elements*, implemented as either a *vector* or a *list*.

Sequences can be created by the *function* **make-sequence**, as well as other *functions* that create *objects* of *types* that are *subtypes* of **sequence** (*e.g.*, **list**, **make-list**, **mapcar**, and **vector**).

A ***sequence function*** is a *function* defined by this specification or added as an extension by the *implementation* that operates on one or more *sequences*. Whenever a *sequence function* must construct and return a new *vector*, it always returns a *simple vector*. Similarly, any *strings* constructed will be *simple strings*.

concatenate	**length**	**remove**
copy-seq	**map**	**remove-duplicates**
count	**map-into**	**remove-if**
count-if	**merge**	**remove-if-not**
count-if-not	**mismatch**	**replace**
delete	**notany**	**reverse**
delete-duplicates	**notevery**	**search**
delete-if	**nreverse**	**some**
delete-if-not	**nsubstitute**	**sort**
elt	**nsubstitute-if**	**stable-sort**
every	**nsubstitute-if-not**	**subseq**
fill	**position**	**substitute**
find	**position-if**	**substitute-if**
find-if	**position-if-not**	**substitute-if-not**
find-if-not	**reduce**	

Figure 17–1. Standardized Sequence Functions

17.1.1 General Restrictions on Parameters that must be Sequences

In general, *lists* (including *association lists* and *property lists*) that are treated as *sequences* must be *proper lists*.

17.2 Rules about Test Functions

17.2.1 Satisfying a Two-Argument Test

When an *object* O is being considered iteratively against each *element* E_i of a *sequence* S by an *operator* F listed in Figure 17–2, it is sometimes useful to control the way in which the presence of O is tested in S is tested by F. This control is offered on the basis of a *function* designated with either a `:test` or `:test-not` *argument*.

adjoin	**nset-exclusive-or**	**search**
assoc	**nsublis**	**set-difference**
count	**nsubst**	**set-exclusive-or**
delete	**nsubstitute**	**sublis**
find	**nunion**	**subsetp**
intersection	**position**	**subst**
member	**pushnew**	**substitute**
mismatch	**rassoc**	**tree-equal**
nintersection	**remove**	**union**
nset-difference	**remove-duplicates**	

Figure 17–2. Operators that have Two-Argument Tests to be Satisfied

The object O might not be compared directly to E_i. If a `:key` *argument* is provided, it is a *designator* for a *function* of one *argument* to be called with each E_i as an *argument*, and *yielding* an *object* Z_i to be used for comparison. (If there is no `:key` *argument*, Z_i is E_i.)

The *function* designated by the `:key` *argument* is never called on O itself. However, if the function operates on multiple sequences (*e.g.*, as happens in **set-difference**), O will be the result of calling the `:key` function on an *element* of the other sequence.

A `:test` *argument*, if supplied to F, is a *designator* for a *function* of two *arguments*, O and Z_i. An E_i is said (or, sometimes, an O and an E_i are said) to ***satisfy the test*** if this `:test` *function* returns a *generalized boolean* representing *true*.

A `:test-not` *argument*, if supplied to F, is *designator* for a *function* of two *arguments*, O and Z_i. An E_i is said (or, sometimes, an O and an E_i are said) to ***satisfy the test*** if this `:test-not` *function* returns a *generalized boolean* representing *false*.

If neither a `:test` nor a `:test-not` *argument* is supplied, it is as if a `:test` argument of `#'eql` was supplied.

The consequences are unspecified if both a `:test` and a `:test-not` *argument* are supplied in the same *call* to F.

17.2.1.1 Examples of Satisfying a Two-Argument Test

```
 (remove "FOO" '(foo bar "FOO" "BAR" "foo" "bar") :test #'equal)
→ (foo bar "BAR" "foo" "bar")
 (remove "FOO" '(foo bar "FOO" "BAR" "foo" "bar") :test #'equalp)
→ (foo bar "BAR" "bar")
 (remove "FOO" '(foo bar "FOO" "BAR" "foo" "bar") :test #'string-equal)
→ (bar "BAR" "bar")
 (remove "FOO" '(foo bar "FOO" "BAR" "foo" "bar") :test #'string=)
→ (BAR "BAR" "foo" "bar")

 (remove 1 '(1 1.0 #C(1.0 0.0) 2 2.0 #C(2.0 0.0)) :test-not #'eql)
→ (1)
 (remove 1 '(1 1.0 #C(1.0 0.0) 2 2.0 #C(2.0 0.0)) :test-not #'=)
→ (1 1.0 #C(1.0 0.0))
 (remove 1 '(1 1.0 #C(1.0 0.0) 2 2.0 #C(2.0 0.0)) :test (complement #'=))
→ (1 1.0 #C(1.0 0.0))

 (count 1 '((one 1) (uno 1) (two 2) (dos 2)) :key #'cadr) → 2

 (count 2.0 '(1 2 3) :test #'eql :key #'float) → 1

 (count "FOO" (list (make-pathname :name "FOO" :type "X")
                    (make-pathname :name "FOO" :type "Y"))
        :key #'pathname-name
        :test #'equal)
→ 2
```

17.2.2 Satisfying a One-Argument Test

When using one of the *functions* in Figure 17–3, the elements E of a *sequence* S are filtered not on the basis of the presence or absence of an object O under a two *argument predicate*, as with the *functions* described in Section 17.2.1 (Satisfying a Two-Argument Test), but rather on the basis of a one *argument predicate*.

assoc-if	**member-if**	**rassoc-if**
assoc-if-not	**member-if-not**	**rassoc-if-not**
count-if	**nsubst-if**	**remove-if**
count-if-not	**nsubst-if-not**	**remove-if-not**
delete-if	**nsubstitute-if**	**subst-if**
delete-if-not	**nsubstitute-if-not**	**subst-if-not**
find-if	**position-if**	**substitute-if**
find-if-not	**position-if-not**	**substitute-if-not**

Figure 17–3. Operators that have One-Argument Tests to be Satisfied

The element E_i might not be considered directly. If a `:key` *argument* is provided, it is a *designator* for a *function* of one *argument* to be called with each E_i as an *argument*, and *yielding* an *object* Z_i to be used for comparison. (If there is no `:key` *argument*, Z_i is E_i.)

Functions defined in this specification and having a name that ends in "`-if`" accept a first *argument* that is a *designator* for a *function* of one *argument*, Z_i. An E_i is said to ***satisfy the test*** if this `:test` *function* returns a *generalized boolean* representing *true*.

Functions defined in this specification and having a name that ends in "`-if-not`" accept a first *argument* that is a *designator* for a *function* of one *argument*, Z_i. An E_i is said to ***satisfy the test*** if this `:test` *function* returns a *generalized boolean* representing *false*.

17.2.2.1 Examples of Satisfying a One-Argument Test

```
 (count-if #'zerop '(1 #C(0.0 0.0) 0 0.0d0 0.0s0 3)) → 4

 (remove-if-not #'symbolp '(0 1 2 3 4 5 6 7 8 9 A B C D E F))
→ (A B C D E F)
 (remove-if (complement #'symbolp) '(0 1 2 3 4 5 6 7 8 9 A B C D E F))
→ (A B C D E F)

 (count-if #'zerop '("foo" "" "bar" "" "" "baz" "quux") :key #'length)
→ 3
```

sequence *System Class*

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*.

copy-seq *Function*

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
```

Exceptional Situations:

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

See Also:

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.

elt *Accessor*

Syntax:

elt *sequence index* → *object*

(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"
```

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*.

fill *Function*

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.

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*)

make-sequence

make-sequence *Function*

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)
```

subseq *Accessor*

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"
```

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

map *Function*

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 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)) → "AAAA"
(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
```

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)

map-into

map-into *Function*

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:

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)
```

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*.

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)
```

reduce *Function*

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*.

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 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))))
```

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)

count, count-if, count-if-not *Function*

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*.

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
```

Exceptional Situations:

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

See Also:

Section 17.2 (Rules about Test Functions), Section 3.6 (Traversal Rules and Side Effects)

Notes:

The `:test-not` *argument* is deprecated.

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

length *Function*

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
(setf (fill-pointer str) 2) → 2
(length str) → 2
```

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**

reverse, nreverse *Function*

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*.)

Exceptional Situations:

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

sort, stable-sort *Function*

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.

sort, stable-sort

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")
                     (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"))
```

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*.

find, find-if, find-if-not *Function*

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
```

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.

position, position-if, position-if-not *Function*

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
```

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.

search *Function*

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
```

See Also:

Section 3.6 (Traversal Rules and Side Effects)

Notes:

The `:test-not` *argument* is deprecated.

mismatch *Function*

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
```

See Also:

Section 3.6 (Traversal Rules and Side Effects)

Notes:

The `:test-not` *argument* is deprecated.

replace *Function*

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 "abcdefghij" "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-1* is modified.

See Also:

fill

substitute, substitute-if, substitute-if-not, nsubstitute, nsubstitute-if, nsubstitute-if-not *Function*

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*.

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.

substitute, substitute-if, substitute-if-not, ...

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.

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*.

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#)
```

concatenate *Function*

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
```

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

merge *Function*

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*.

merge

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 `x1` and `x2` are two elements of `x`, and `x1` precedes `x2` in `x`, then `x1` precedes `x2` 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
```

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)

remove, remove-if, remove-if-not, delete, delete-if, delete-if-not *Function*

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.

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:

remove, remove-if, remove-if-not, delete, delete-if, ...

```
(delete nil sequence
            :test #'(lambda (ignore item) (funcall test item))
            ...)
```

Examples:

```
(remove 4 '(1 3 4 5 9)) → (1 3 5 9)
(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.

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.

remove-duplicates, delete-duplicates *Function*

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**.

remove-duplicates, delete-duplicates

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))
    :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*.

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)

remove-duplicates, delete-duplicates

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.

Programming Language—Common Lisp

18. Hash Tables

18.1 Hash Table Concepts

18.1.1 Hash-Table Operations

Figure 18–1 lists some *defined names* that are applicable to *hash tables*. The following rules apply to *hash tables*.

– A *hash table* can only associate one value with a given key. If an attempt is made to add a second value for a given key, the second value will replace the first. Thus, adding a value to a *hash table* is a destructive operation; the *hash table* is modified.

– There are four kinds of *hash tables*: those whose keys are compared with **eq**, those whose keys are compared with **eql**, those whose keys are compared with **equal**, and those whose keys are compared with **equalp**.

– *Hash tables* are created by **make-hash-table**. **gethash** is used to look up a key and find the associated value. New entries are added to *hash tables* using **setf** with **gethash**. **remhash** is used to remove an entry. For example:

```
(setq a (make-hash-table)) → #<HASH-TABLE EQL 0/120 32536573>
(setf (gethash 'color a) 'brown) → BROWN
(setf (gethash 'name a) 'fred) → FRED
(gethash 'color a) → BROWN, true
(gethash 'name a) → FRED, true
(gethash 'pointy a) → NIL, false
```

In this example, the symbols `color` and `name` are being used as keys, and the symbols `brown` and `fred` are being used as the associated values. The *hash table* has two items in it, one of which associates from `color` to `brown`, and the other of which associates from `name` to `fred`.

– A key or a value may be any *object*.

– The existence of an entry in the *hash table* can be determined from the *secondary value* returned by **gethash**.

clrhash	**hash-table-p**	**remhash**
gethash	**make-hash-table**	**sxhash**
hash-table-count	**maphash**	

Figure 18–1. Hash-table defined names

18.1.2 Modifying Hash Table Keys

The function supplied as the `:test` argument to **make-hash-table** specifies the 'equivalence test' for the *hash table* it creates.

An *object* is 'visibly modified' with regard to an equivalence test if there exists some set of *objects* (or potential *objects*) which are equivalent to the *object* before the modification but are no longer equivalent afterwards.

If an *object* O_1 is used as a key in a *hash table* H and is then visibly modified with regard to the equivalence test of H, then the consequences are unspecified if O_1, or any *object* O_2 equivalent to O_1 under the equivalence test (either before or after the modification), is used as a key in further operations on H. The consequences of using O_1 as a key are unspecified even if O_1 is visibly modified and then later modified again in such a way as to undo the visible modification.

Following are specifications of the modifications which are visible to the equivalence tests which must be supported by *hash tables*. The modifications are described in terms of modification of components, and are defined recursively. Visible modifications of components of the *object* are visible modifications of the *object*.

18.1.2.1 Visible Modification of Objects with respect to EQ and EQL

No *standardized function* is provided that is capable of visibly modifying an *object* with regard to **eq** or **eql**.

18.1.2.2 Visible Modification of Objects with respect to EQUAL

As a consequence of the behavior for **equal**, the rules for visible modification of *objects* not explicitly mentioned in this section are inherited from those in Section 18.1.2.1 (Visible Modification of Objects with respect to EQ and EQL).

18.1.2.2.1 Visible Modification of Conses with respect to EQUAL

Any visible change to the *car* or the *cdr* of a *cons* is considered a visible modification with regard to **equal**.

18.1.2.2.2 Visible Modification of Bit Vectors and Strings with respect to EQUAL

For a *vector* of *type* **bit-vector** or of *type* **string**, any visible change to an *active element* of the *vector*, or to the *length* of the *vector* (if it is *actually adjustable* or has a *fill pointer*) is considered a visible modification with regard to **equal**.

18.1.2.3 Visible Modification of Objects with respect to EQUALP

As a consequence of the behavior for **equalp**, the rules for visible modification of *objects* not explicitly mentioned in this section are inherited from those in Section 18.1.2.2 (Visible Modification of Objects with respect to EQUAL).

18.1.2.3.1 Visible Modification of Structures with respect to EQUALP

Any visible change to a *slot* of a *structure* is considered a visible modification with regard to **equalp**.

18.1.2.3.2 Visible Modification of Arrays with respect to EQUALP

In an *array*, any visible change to an *active element*, to the *fill pointer* (if the *array* can and does have one), or to the *dimensions* (if the *array* is *actually adjustable*) is considered a visible modification with regard to **equalp**.

18.1.2.3.3 Visible Modification of Hash Tables with respect to EQUALP

In a *hash table*, any visible change to the count of entries in the *hash table*, to the keys, or to the values associated with the keys is considered a visible modification with regard to **equalp**.

Note that the visibility of modifications to the keys depends on the equivalence test of the *hash table*, not on the specification of **equalp**.

18.1.2.4 Visible Modifications by Language Extensions

Implementations that extend the language by providing additional mutator functions (or additional behavior for existing mutator functions) must document how the use of these extensions interacts with equivalence tests and *hash table* searches.

Implementations that extend the language by defining additional acceptable equivalence tests for *hash tables* (allowing additional values for the `:test` argument to **make-hash-table**) must document the visible components of these tests.

hash-table *System Class*

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:

Section 18.1 (Hash Table Concepts), Section 22.1.3.13 (Printing Other Objects)

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.

make-hash-table *Function*

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-theshold* 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>
```

See Also:

gethash, **hash-table**

hash-table-p *Function*

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
(hash-table-p '((a . 1) (b . 2))) → false
```

Notes:

```
(hash-table-p object) ≡ (typep object 'hash-table)
```

hash-table-count *Function*

Syntax:

hash-table-count *hash-table* → *count*

Arguments and Values:

hash-table—a *hash table*.

count—a non-negative *integer*.

Description:

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
```

Affected By:

clrhash, **remhash**, **setf** of **gethash**

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)
```

hash-table-rehash-size *Function*

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:

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
```

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*.

hash-table-rehash-threshold *Function*

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
```

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**

hash-table-size *Function*

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*.

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**

hash-table-test *Function*

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*.

Exceptional Situations:

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

See Also:

make-hash-table

gethash *Accessor*

Syntax:

gethash *key hash-table* &optional *default* → *value, present-p*

(**setf** (**gethash** *key hash-table* &optional *default*) *new-value*)

Arguments and Values:

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 foo bar bar baz)) (count-it x))
(how-many 'foo) → 2
(how-many 'bar) → 3
(how-many 'quux) → 0
```

See Also:

remhash

Notes:

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*.

remhash *Function*

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.

maphash

maphash *Function*

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 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*.

See Also:

loop, **with-hash-table-iterator**, Section 3.6 (Traversal Rules and Side Effects)

with-hash-table-iterator *Macro*

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:

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)))))
```

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)

clrhash *Function*

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.

sxhash *Function*

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).

sxhash

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
```

Affected By:

The *implementation*.

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*.

Programming Language—Common Lisp

19. Filenames

19.1 Overview of Filenames

There are many kinds of *file systems*, varying widely both in their superficial syntactic details, and in their underlying power and structure. The facilities provided by Common Lisp for referring to and manipulating *files* has been chosen to be compatible with many kinds of *file systems*, while at the same time minimizing the program-visible differences between kinds of *file systems*.

Since *file systems* vary in their conventions for naming *files*, there are two distinct ways to represent *filenames*: as *namestrings* and as *pathnames*.

19.1.1 Namestrings as Filenames

A ***namestring*** is a *string* that represents a *filename*.

In general, the syntax of *namestrings* involves the use of *implementation-defined* conventions, usually those customary for the *file system* in which the named *file* resides. The only exception is the syntax of a *logical pathname namestring*, which is defined in this specification; see Section 19.3.1 (Syntax of Logical Pathname Namestrings).

A *conforming program* must never unconditionally use a *literal namestring* other than a *logical pathname namestring* because Common Lisp does not define any *namestring* syntax other than that for *logical pathnames* that would be guaranteed to be portable. However, a *conforming program* can, if it is careful, successfully manipulate user-supplied data which contains or refers to non-portable *namestrings*.

A *namestring* can be *coerced* to a *pathname* by the *functions* **pathname** or **parse-namestring**.

19.1.2 Pathnames as Filenames

Pathnames are structured *objects* that can represent, in an *implementation-independent* way, the *filenames* that are used natively by an underlying *file system*.

In addition, *pathnames* can also represent certain partially composed *filenames* for which an underlying *file system* might not have a specific *namestring* representation.

A *pathname* need not correspond to any file that actually exists, and more than one *pathname* can refer to the same file. For example, the *pathname* with a version of `:newest` might refer to the same file as a *pathname* with the same components except a certain number as the version. Indeed, a *pathname* with version `:newest` might refer to different files as time passes, because the meaning of such a *pathname* depends on the state of the file system.

Some *file systems* naturally use a structural model for their *filenames*, while others do not. Within the Common Lisp *pathname* model, all *filenames* are seen as having a particular structure, even if that structure is not reflected in the underlying *file system*. The nature of the mapping between structure imposed by *pathnames* and the structure, if any, that is used by the underlying *file system* is *implementation-defined*.

Every *pathname* has six components: a host, a device, a directory, a name, a type, and a version. By naming *files* with *pathnames*, Common Lisp programs can work in essentially the same way even in *file systems* that seem superficially quite different. For a detailed description of these components, see Section 19.2.1 (Pathname Components).

The mapping of the *pathname* components into the concepts peculiar to each *file system* is *implementation-defined*. There exist conceivable *pathnames* for which there is no mapping to a syntactically valid *filename* in a particular *implementation*. An *implementation* may use various strategies in an attempt to find a mapping; for example, an *implementation* may quietly truncate *filenames* that exceed length limitations imposed by the underlying *file system*, or ignore certain *pathname* components for which the *file system* provides no support. If such a mapping cannot be found, an error of *type* **file-error** is signaled.

The time at which this mapping and associated error signaling occurs is *implementation-dependent*. Specifically, it may occur at the time the *pathname* is constructed, when coercing a *pathname* to a *namestring*, or when an attempt is made to *open* or otherwise access the *file* designated by the *pathname*.

Figure 19–1 lists some *defined names* that are applicable to *pathnames*.

default-pathname-defaults	**namestring**	**pathname-name**
directory-namestring	**open**	**pathname-type**
enough-namestring	**parse-namestring**	**pathname-version**
file-namestring	**pathname**	**pathnamep**
file-string-length	**pathname-device**	**translate-pathname**
host-namestring	**pathname-directory**	**truename**
make-pathname	**pathname-host**	**user-homedir-pathname**
merge-pathnames	**pathname-match-p**	**wild-pathname-p**

Figure 19–1. Pathname Operations

19.1.3 Parsing Namestrings Into Pathnames

Parsing is the operation used to convert a *namestring* into a *pathname*. Except in the case of parsing *logical pathname namestrings*, this operation is *implementation-dependent*, because the format of *namestrings* is *implementation-dependent*.

A *conforming implementation* is free to accommodate other *file system* features in its *pathname* representation and provides a parser that can process such specifications in *namestrings*. *Conforming programs* must not depend on any such features, since those features will not be portable.

19.2 Pathnames

19.2.1 Pathname Components

A *pathname* has six components: a host, a device, a directory, a name, a type, and a version.

19.2.1.1 The Pathname Host Component

The name of the file system on which the file resides, or the name of a *logical host*.

19.2.1.2 The Pathname Device Component

Corresponds to the "device" or "file structure" concept in many host file systems: the name of a logical or physical device containing files.

19.2.1.3 The Pathname Directory Component

Corresponds to the "directory" concept in many host file systems: the name of a group of related files.

19.2.1.4 The Pathname Name Component

The "name" part of a group of *files* that can be thought of as conceptually related.

19.2.1.5 The Pathname Type Component

Corresponds to the "filetype" or "extension" concept in many host file systems. This says what kind of file this is. This component is always a *string*, **nil**, `:wild`, or `:unspecific`.

19.2.1.6 The Pathname Version Component

Corresponds to the "version number" concept in many host file systems.

The version is either a positive *integer* or a *symbol* from the following list: **nil**, `:wild`, `:unspecific`, or `:newest` (refers to the largest version number that already exists in the file system when reading a file, or to a version number greater than any already existing in the file system when writing a new file). Implementations can define other special version *symbols*.

19.2.2 Interpreting Pathname Component Values

19.2.2.1 Strings in Component Values

19.2.2.1.1 Special Characters in Pathname Components

Strings in *pathname* component values never contain special *characters* that represent separation between *pathname* fields, such as *slash* in Unix *filenames*. Whether separator *characters* are permitted as part of a *string* in a *pathname* component is *implementation-defined*; however, if the *implementation* does permit it, it must arrange to properly "quote" the character for the *file system* when constructing a *namestring*. For example,

```
 ;; In a TOPS-20 implementation, which uses ^V to quote
 (NAMESTRING (MAKE-PATHNAME :HOST "OZ" :NAME "<TEST>"))
→ #P"OZ:PS:^V<TEST^V>"
not→ #P"OZ:PS:<TEST>"
```

19.2.2.1.2 Case in Pathname Components

Namestrings always use local file system *case* conventions, but Common Lisp *functions* that manipulate *pathname* components allow the caller to select either of two conventions for representing *case* in component values by supplying a value for the `:case` keyword argument. Figure 19–2 lists the functions relating to *pathnames* that permit a `:case` argument:

make-pathname	**pathname-directory**	**pathname-name**
pathname-device	**pathname-host**	**pathname-type**

Figure 19–2. Pathname functions using a :CASE argument

19.2.2.1.2.1 Local Case in Pathname Components

For the functions in Figure 19–2, a value of `:local` for the `:case` argument (the default for these functions) indicates that the functions should receive and yield *strings* in component values as if they were already represented according to the host *file system*'s convention for *case*.

If the *file system* supports both *cases*, *strings* given or received as *pathname* component values under this protocol are to be used exactly as written. If the file system only supports one *case*, the *strings* will be translated to that *case*.

19.2.2.1.2.2 Common Case in Pathname Components

For the functions in Figure 19–2, a value of `:common` for the `:case` argument that these *functions* should receive and yield *strings* in component values according to the following conventions:

- All *uppercase* means to use a file system's customary *case*.
- All *lowercase* means to use the opposite of the customary *case*.
- Mixed *case* represents itself.

Note that these conventions have been chosen in such a way that translation from `:local` to `:common` and back to `:local` is information-preserving.

19.2.2.2 Special Pathname Component Values

19.2.2.2.1 NIL as a Component Value

As a *pathname* component value, **nil**represents that the component is "unfilled"; see Section 19.2.3 (Merging Pathnames).

The value of any *pathname* component can be **nil**.

When constructing a *pathname*, **nil** in the host component might mean a default host rather than an actual **nil** in some *implementations*.

19.2.2.2.2 :WILD as a Component Value

If `:wild` is the value of a *pathname* component, that component is considered to be a wildcard, which matches anything.

A *conforming program* must be prepared to encounter a value of `:wild` as the value of any *pathname* component, or as an *element* of a *list* that is the value of the directory component.

When constructing a *pathname*, a *conforming program* may use `:wild` as the value of any or all of the directory, name, type, or version component, but must not use `:wild` as the value of the host, or device component.

If `:wild` is used as the value of the directory component in the construction of a *pathname*, the effect is equivalent to specifying the list `(:absolute :wild-inferiors)`, or the same as `(:absolute :wild)` in a *file system* that does not support `:wild-inferiors`.

19.2.2.2.3 :UNSPECIFIC as a Component Value

If `:unspecific` is the value of a *pathname* component, the component is considered to be "absent" or to "have no meaning" in the *filename* being represented by the *pathname*.

Whether a value of `:unspecific` is permitted for any component on any given *file system* accessible to the *implementation* is *implementation-defined*. A *conforming program* must never unconditionally use a `:unspecific` as the value of a *pathname* component because such a value is not guaranteed to be permissible in all implementations. However, a *conforming program* can, if it is careful, successfully manipulate user-supplied data which contains or refers to non-portable *pathname* components. And certainly a *conforming program* should be prepared for the possibility that any components of a *pathname* could be `:unspecific`.

When *reading*$_1$ the value of any *pathname* component, *conforming programs* should be prepared for the value to be `:unspecific`.

When *writing*$_1$ the value of any *pathname* component, the consequences are undefined if `:unspecific` is given for a *pathname* in a *file system* for which it does not make sense.

19.2.2.2.3.1 Relation between component values NIL and :UNSPECIFIC

If a *pathname* is converted to a *namestring*, the *symbols* **nil** and `:unspecific` cause the field to be treated as if it were empty. That is, both **nil** and `:unspecific` cause the component not to appear in the *namestring*.

However, when merging a *pathname* with a set of defaults, only a **nil** value for a component will be replaced with the default for that component, while a value of `:unspecific` will be left alone as if the field were "filled"; see the *function* **merge-pathnames** and Section 19.2.3 (Merging Pathnames).

19.2.2.3 Restrictions on Wildcard Pathnames

Wildcard *pathnames* can be used with **directory** but not with **open**, and return true from **wild-pathname-p**. When examining wildcard components of a wildcard *pathname*, conforming programs must be prepared to encounter any of the following additional values in any component or any element of a *list* that is the directory component:

- The *symbol* `:wild`, which matches anything.
- A *string* containing *implementation-dependent* special wildcard *characters*.
- Any *object*, representing an *implementation-dependent* wildcard pattern.

19.2.2.4 Restrictions on Examining Pathname Components

The space of possible *objects* that a *conforming program* must be prepared to *read*$_1$ as the value of a *pathname* component is substantially larger than the space of possible *objects* that a *conforming program* is permitted to *write*$_1$ into such a component.

While the values discussed in the subsections of this section, in Section 19.2.2.2 (Special Pathname Component Values), and in Section 19.2.2.3 (Restrictions on Wildcard Pathnames) apply to values that might be seen when reading the component values, substantially more restrictive rules apply to constructing pathnames; see Section 19.2.2.5 (Restrictions on Constructing Pathnames).

When examining *pathname* components, *conforming programs* should be aware of the following restrictions.

19.2.2.4.1 Restrictions on Examining a Pathname Host Component

It is *implementation-dependent* what *object* is used to represent the host.

19.2.2.4.2 Restrictions on Examining a Pathname Device Component

The device might be a *string*, `:wild`, `:unspecific`, or **nil**.

Note that `:wild` might result from an attempt to *read*$_1$ the *pathname* component, even though portable programs are restricted from *writing*$_1$ such a component value; see Section 19.2.2.3 (Restrictions on Wildcard Pathnames) and Section 19.2.2.5 (Restrictions on Constructing Pathnames).

19.2.2.4.3 Restrictions on Examining a Pathname Directory Component

The directory might be a *string*, `:wild`, `:unspecific`, or **nil**.

The directory can be a *list* of *strings* and *symbols*. The *car* of the *list* is one of the symbols `:absolute` or `:relative`, meaning:

`:absolute`

A *list* whose *car* is the symbol `:absolute` represents a directory path starting from the root directory. The list `(:absolute)` represents the root directory. The list `(:absolute "foo" "bar" "baz")` represents the directory called `"/foo/bar/baz"` in Unix (except possibly for *case*).

`:relative`

A *list* whose *car* is the symbol `:relative` represents a directory path starting from a default directory. The list `(:relative)` has the same meaning as **nil** and hence is not used. The list `(:relative "foo" "bar")` represents the directory named `"bar"` in the directory named `"foo"` in the default directory.

Each remaining element of the *list* is a *string* or a *symbol*.

Each *string* names a single level of directory structure. The *strings* should contain only the directory names themselves—no punctuation characters.

In place of a *string*, at any point in the *list*, *symbols* can occur to indicate special file notations. Figure 19–3 lists the *symbols* that have standard meanings. Implementations are permitted to add additional *objects* of any *type* that is disjoint from **string** if necessary to represent features of their file systems that cannot be represented with the standard *strings* and *symbols*.

Supplying any non-*string*, including any of the *symbols* listed below, to a file system for which it does not make sense signals an error of *type* **file-error**. For example, Unix does not support `:wild-inferiors` in most implementations.

Symbol	Meaning
`:wild`	Wildcard match of one level of directory structure
`:wild-inferiors`	Wildcard match of any number of directory levels
`:up`	Go upward in directory structure (semantic)
`:back`	Go upward in directory structure (syntactic)

Figure 19–3. Special Markers In Directory Component

The following notes apply to the previous figure:

Invalid Combinations

Using `:absolute` or `:wild-inferiors` immediately followed by `:up` or `:back` signals an error of *type* **file-error**.

Syntactic vs Semantic

"Syntactic" means that the action of `:back` depends only on the *pathname* and not on the contents of the file system.

"Semantic" means that the action of `:up` depends on the contents of the file system; to resolve a *pathname* containing `:up` to a *pathname* whose directory component contains only `:absolute` and *strings* requires probing the file system.

`:up` differs from `:back` only in file systems that support multiple names for directories, perhaps via symbolic links. For example, suppose that there is a directory `(:absolute "X" "Y" "Z")` linked to `(:absolute "A" "B" "C")` and there also exist directories `(:absolute "A" "B" "Q")` and `(:absolute "X" "Y" "Q")`. Then `(:absolute "X" "Y" "Z" :up "Q")` designates `(:absolute "A" "B" "Q")` while `(:absolute "X" "Y" "Z" :back "Q")` designates `(:absolute "X" "Y" "Q")`

19.2.2.4.3.1 Directory Components in Non-Hierarchical File Systems

In non-hierarchical *file systems*, the only valid *list* values for the directory component of a *pathname* are `(:absolute` *string*`)` and `(:absolute :wild)`. `:relative` directories and the keywords `:wild-inferiors`, `:up`, and `:back` are not used in non-hierarchical *file systems*.

19.2.2.4.4 Restrictions on Examining a Pathname Name Component

The name might be a *string*, `:wild`, `:unspecific`, or **nil**.

19.2.2.4.5 Restrictions on Examining a Pathname Type Component

The type might be a *string*, `:wild`, `:unspecific`, or **nil**.

19.2.2.4.6 Restrictions on Examining a Pathname Version Component

The version can be any *symbol* or any *integer*.

The symbol `:newest` refers to the largest version number that already exists in the *file system* when reading, overwriting, appending, superseding, or directory listing an existing *file*. The symbol `:newest` refers to the smallest version number greater than any existing version number when creating a new file.

The symbols **nil**, `:unspecific`, and `:wild` have special meanings and restrictions; see Section 19.2.2.2 (Special Pathname Component Values) and Section 19.2.2.5 (Restrictions on Constructing Pathnames).

Other *symbols* and *integers* have *implementation-defined* meaning.

19.2.2.4.7 Notes about the Pathname Version Component

It is suggested, but not required, that implementations do the following:

- Use positive *integers* starting at 1 as version numbers.
- Recognize the symbol `:oldest` to designate the smallest existing version number.
- Use *keywords* for other special versions.

19.2.2.5 Restrictions on Constructing Pathnames

When constructing a *pathname* from components, conforming programs must follow these rules:

- Any component can be **nil**. **nil** in the host might mean a default host rather than an actual **nil** in some implementations.

- The host, device, directory, name, and type can be *strings*. There are *implementation-dependent* limits on the number and type of *characters* in these *strings*.

- The directory can be a *list* of *strings* and *symbols*. There are *implementation-dependent* limits on the *list*'s length and contents.

- The version can be `:newest`.

- Any component can be taken from the corresponding component of another *pathname*. When the two *pathnames* are for different file systems (in implementations that support multiple file systems), an appropriate translation occurs. If no meaningful translation is possible, an error is signaled. The definitions of "appropriate" and "meaningful" are *implementation-dependent*.

- An implementation might support other values for some components, but a portable program cannot use those values. A conforming program can use *implementation-dependent* values but this can make it non-portable; for example, it might work only with Unix file systems.

19.2.3 Merging Pathnames

Merging takes a *pathname* with unfilled components and supplies values for those components from a source of defaults.

If a component's value is **nil**, that component is considered to be unfilled. If a component's value is any *non-nil object*, including `:unspecific`, that component is considered to be filled.

Except as explicitly specified otherwise, for functions that manipulate or inquire about *files* in the *file system*, the pathname argument to such a function is merged with ***default-pathname-defaults*** before accessing the *file system* (as if by **merge-pathnames**).

19.2.3.1 Examples of Merging Pathnames

Although the following examples are possible to execute only in *implementations* which permit `:unspecific` in the indicated position andwhich permit four-letter type components, they serve to illustrate the basic concept of *pathname* merging.

```
 (pathname-type
   (merge-pathnames (make-pathname :type "LISP")
                    (make-pathname :type "TEXT")))
→ "LISP"


 (pathname-type
   (merge-pathnames (make-pathname :type nil)
                    (make-pathname :type "LISP")))
→ "LISP"


 (pathname-type
   (merge-pathnames (make-pathname :type :unspecific)
                    (make-pathname :type "LISP")))
→ :UNSPECIFIC
```

19.3 Logical Pathnames

19.3.1 Syntax of Logical Pathname Namestrings

The syntax of a *logical pathname namestring* is as follows. (Note that unlike many notational descriptions in this document, this is a syntactic description of character sequences, not a structural description of *objects*.)

logical-pathname::=[↓*host* *host-marker*]
[↓*relative-directory-marker*] {↓*directory* *directory-marker*}*
[↓*name*] [*type-marker* ↓*type* [*version-marker* ↓*version*]]

host::=↓*word*

directory::=↓*word* | ↓*wildcard-word* | ↓*wild-inferiors-word*

name::=↓*word* | ↓*wildcard-word*

type::=↓*word* | ↓*wildcard-word*

version::=↓*pos-int* | *newest-word* | *wildcard-version*

host-marker—a *colon*.

relative-directory-marker—a *semicolon*.

directory-marker—a *semicolon*.

type-marker—a *dot*.

version-marker—a *dot*.

wild-inferiors-word—The two character sequence "`**`" (two *asterisks*).

newest-word—The six character sequence "`newest`" or the six character sequence "`NEWEST`".

wildcard-version—an *asterisk*.

wildcard-word—one or more *asterisks*, uppercase letters, digits, and hyphens, including at least one *asterisk*, with no two *asterisks* adjacent.

word—one or more uppercase letters, digits, and hyphens.

pos-int—a positive *integer*.

19.3.1.1 Additional Information about Parsing Logical Pathname Namestrings

19.3.1.1.1 The Host part of a Logical Pathname Namestring

The *host* must have been defined as a *logical pathname* host; this can be done by using **setf** of **logical-pathname-translations**.

The *logical pathname* host name `"SYS"` is reserved for the implementation. The existence and meaning of `SYS:` *logical pathnames* is *implementation-defined*.

19.3.1.1.2 The Device part of a Logical Pathname Namestring

There is no syntax for a *logical pathname* device since the device component of a *logical pathname* is always `:unspecific`; see Section 19.3.2.1 (Unspecific Components of a Logical Pathname).

19.3.1.1.3 The Directory part of a Logical Pathname Namestring

If a *relative-directory-marker* precedes the *directories*, the directory component parsed is as *relative*; otherwise, the directory component is parsed as *absolute*.

If a *wild-inferiors-marker* is specified, it parses into `:wild-inferiors`.

19.3.1.1.4 The Type part of a Logical Pathname Namestring

The *type* of a *logical pathname* for a *source file* is `"LISP"`. This should be translated into whatever type is appropriate in a physical pathname.

19.3.1.1.5 The Version part of a Logical Pathname Namestring

Some *file systems* do not have *versions*. *Logical pathname* translation to such a *file system* ignores the *version*. This implies that a program cannot rely on being able to store more than one version of a file named by a *logical pathname*.

If a *wildcard-version* is specified, it parses into `:wild`.

19.3.1.1.6 Wildcard Words in a Logical Pathname Namestring

Each *asterisk* in a *wildcard-word* matches a sequence of zero or more characters. The *wildcard-word* "`*`" parses into `:wild`; other *wildcard-words* parse into *strings*.

19.3.1.1.7 Lowercase Letters in a Logical Pathname Namestring

When parsing *words* and *wildcard-words*, lowercase letters are translated to uppercase.

19.3.1.1.8 Other Syntax in a Logical Pathname Namestring

The consequences of using characters other than those specified here in a *logical pathname namestring* are unspecified.

The consequences of using any value not specified here as a *logical pathname* component are unspecified.

19.3.2 Logical Pathname Components

19.3.2.1 Unspecific Components of a Logical Pathname

The device component of a *logical pathname* is always `:unspecific`; no other component of a *logical pathname* can be `:unspecific`.

19.3.2.2 Null Strings as Components of a Logical Pathname

The null string, `""`, is not a valid value for any component of a *logical pathname*.

pathname *System Class*

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*.

logical-pathname *System Class*

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:

Section 20.1 (File System Concepts), Section 2.4.8.14 (Sharpsign P), Section 22.1.3.11 (Printing Pathnames)

pathname *Function*

Syntax:

pathname *pathspec* → *pathname*

Arguments and Values:

pathspec—a *pathname designator*.

pathname—a *pathname*.

Description:

Returns the *pathname* denoted by *pathspec*.

pathname

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"
```

See Also:

pathname, **logical-pathname**, Section 20.1 (File System Concepts), Section 19.1.2 (Pathnames as

Filenames)

make-pathname *Function*

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*.

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*.

make-pathname

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*.

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 `:unspecific` for any component. See Section 19.2.2.2.3 (:UNSPECIFIC as a Component Value).

pathnamep *Function*

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
```

Notes:

```
(pathnamep object) ≡ (typep object 'pathname)
```

pathname-host, pathname-device, pathname-directory, pathname-name, pathname-type, pathname-version *Function*

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"
 (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)

load-logical-pathname-translations *Function*

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
```

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.

logical-pathname-translations *Accessor*

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*.

logical-pathname-translations

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
;;;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-translations

```
                               ,(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))))
                        (setq phy (concatenate 'string phy "B"))
                        (list com
                              (make-pathname :name phy
                                             :defaults phypath))))
                  l))))
```

```
;;;Sample use of that logical pathname.  The return value
;;;is implementation-dependent.
(translate-logical-pathname "prog:code;documentation.lisp")
→ #P"PGDOC"
```

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.

logical-pathname *Function*

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**.

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)

∗default-pathname-defaults∗ *Variable*

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*.

namestring, file-namestring, directory-namestring, host-namestring, enough-namestring *Function*

Syntax:

namestring *pathname* → *namestring*

file-namestring *pathname* → *namestring*

directory-namestring *pathname* → *namestring*

host-namestring *pathname* → *namestring*

enough-namestring *pathname* `&optional` *defaults* → *namestring*

namestring, file-namestring, directory-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")
→ "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"
```

See Also:

truename, **merge-pathnames**, **pathname**, **logical-pathname**, Section 20.1 (File System Concepts), Section 19.1.2 (Pathnames as Filenames)

---

parse-namestring *Function*

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***.

parse-namestring

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")
→ #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"
```

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)

wild-pathname-p *Function*

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
```

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 19.2.2.2.2 (:WILD as a

Component Value) and Section 19.2.2.3 (Restrictions on Wildcard Pathnames).

pathname-match-p *Function*

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*.

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)

translate-logical-pathname *Function*

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**.

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)

translate-pathname *Function*

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.

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*.

translate-pathname

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
```

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.

merge-pathnames *Function*

Syntax:

merge-pathnames *pathname* &optional *default-pathname default-version*
→ *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*.

merge-pathnames

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`,

merge-pathnames

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"
```

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.

Programming Language—Common Lisp

20. Files

20.1 File System Concepts

This section describes the Common Lisp interface to file systems. The model used by this interface assumes that ***files*** are named by ***filenames***, that a *filename* can be represented by a *pathname object*, and that given a *pathname* a ***stream*** can be constructed that connects to a *file* whose *filename* it represents.

For information about opening and closing *files*, and manipulating their contents, see Chapter 21 (Streams).

Figure 20–1 lists some *operators* that are applicable to *files* and directories.

compile-file	**file-length**	**open**
delete-file	**file-position**	**probe-file**
directory	**file-write-date**	**rename-file**
file-author	**load**	**with-open-file**

Figure 20–1. File and Directory Operations

20.1.1 Coercion of Streams to Pathnames

A ***stream associated with a file*** is either a *file stream* or a *synonym stream* whose target is a ***stream associated with a file***. Such streams can be used as *pathname designators*.

Normally, when a *stream associated with a file* is used as a *pathname designator*, it denotes the *pathname* used to open the *file*; this may be, but is not required to be, the actual name of the *file*.

Some functions, such as **truename** and **delete-file**, coerce *streams* to *pathnames* in a different way that involves referring to the actual *file* that is open, which might or might not be the file whose name was opened originally. Such special situations are always notated specifically and are not the default.

20.1.2 File Operations on Open and Closed Streams

Many *functions* that perform *file* operations accept either *open* or *closed streams* as *arguments*; see Section 21.1.3 (Stream Arguments to Standardized Functions).

Of these, the *functions* in Figure 20–2 treat *open* and *closed streams* differently.

delete-file	**file-author**	**probe-file**
directory	**file-write-date**	**truename**

Figure 20–2. File Functions that Treat Open and Closed Streams Differently

Since treatment of *open streams* by the *file system* may vary considerably between *implementations*, however, a *closed stream* might be the most reliable kind of *argument* for some of these functions—in particular, those in Figure 20–3. For example, in some *file systems*, *open files* are written under temporary names and not renamed until *closed* and/or are held invisible until *closed*. In general, any code that is intended to be portable should use such *functions* carefully.

directory	**probe-file**	**truename**

Figure 20–3. File Functions where Closed Streams Might Work Best

20.1.3 Truenames

Many *file systems* permit more than one *filename* to designate a particular *file*.

Even where multiple names are possible, most *file systems* have a convention for generating a canonical *filename* in such situations. Such a canonical *filename* (or the *pathname* representing such a *filename*) is called a ***truename***.

The *truename* of a *file* may differ from other *filenames* for the file because of symbolic links, version numbers, logical device translations in the *file system*, *logical pathname* translations within Common Lisp, or other artifacts of the *file system*.

The *truename* for a *file* is often, but not necessarily, unique for each *file*. For instance, a Unix *file* with multiple hard links could have several *truenames*.

20.1.3.1 Examples of Truenames

For example, a DEC TOPS-20 system with *files* `PS:<JOE>FOO.TXT.1` and `PS:<JOE>FOO.TXT.2` might permit the second *file* to be referred to as `PS:<JOE>FOO.TXT.0`, since the "`.0`" notation denotes "newest" version of several *files*. In the same *file system*, a "logical device" "`JOE:`" might be taken to refer to `PS:<JOE>`" and so the names `JOE:FOO.TXT.2` or `JOE:FOO.TXT.0` might refer to `PS:<JOE>FOO.TXT.2`. In all of these cases, the *truename* of the file would probably be `PS:<JOE>FOO.TXT.2`.

If a *file* is a symbolic link to another *file* (in a *file system* permitting such a thing), it is conventional for the *truename* to be the canonical name of the *file* after any symbolic links have been followed; that is, it is the canonical name of the *file* whose contents would become available if an *input stream* to that *file* were opened.

In the case of a *file* still being created (that is, of an *output stream* open to such a *file*), the exact *truename* of the file might not be known until the *stream* is closed. In this case, the *function* **truename** might return different values for such a *stream* before and after it was closed. In fact, before it is closed, the name returned might not even be a valid name in the *file system*—for example, while a file is being written, it might have version `:newest` and might only take on a specific numeric value later when the file is closed even in a *file system* where all files have numeric versions.

directory *Function*

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**.

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 21.1.1.1.2 (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*.

probe-file *Function*

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**.

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 21.1.1.1.2 (Open and Closed Streams), Section 19.1.2 (Pathnames as Filenames)

ensure-directories-exist *Function*

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.

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*.

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)

truename *Function*

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")
   (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"
```

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*.

file-author *Function*

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)

file-write-date *Function*

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:

Section 25.1.4.2 (Universal Time), Section 19.1.2 (Pathnames as Filenames)

rename-file *Function*

Syntax:

rename-file *filespec new-name* → *defaulted-new-name, old-truename, new-truename*

Arguments and Values:

filespec—a *pathname designator*.

new-name—a *pathname designator* other than a *stream*.

defaulted-new-name—a *pathname*

old-truename—a *physical pathname*.

new-truename—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-truename*, is the *truename* of the *file* before it was renamed. The *tertiary value*, *new-truename*, is the *truename* 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) (truename 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"
```

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)

delete-file *Function*

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
```

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)

file-error *Condition Type*

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**

file-error-pathname *Function*

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**.

Exceptional Situations:

See Also:

file-error, Chapter 9 (Conditions)

Programming Language—Common Lisp

21. Streams

21.1 Stream Concepts

21.1.1 Introduction to Streams

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. A ***character stream*** is a source or sink of *characters*. A ***binary stream*** is a source or sink of *bytes*.

Some operations may be performed on any kind of *stream*; Figure 21–1 provides a list of *standardized* operations that are potentially useful with any kind of *stream*.

close	**stream-element-type**
input-stream-p	**streamp**
interactive-stream-p	**with-open-stream**
output-stream-p	

Figure 21–1. Some General-Purpose Stream Operations

Other operations are only meaningful on certain *stream types*. For example, **read-char** is only defined for *character streams* and **read-byte** is only defined for *binary streams*.

21.1.1.1 Abstract Classifications of Streams

21.1.1.1.1 Input, Output, and Bidirectional Streams

A *stream*, whether a *character stream* or a *binary stream*, can be an ***input stream*** (source of data), an ***output stream*** (sink for data), both, or (*e.g.*, when "`:direction :probe`" is given to **open**) neither.

Figure 21–2 shows *operators* relating to *input streams*.

clear-input	**read-byte**	**read-from-string**
listen	**read-char**	**read-line**
peek-char	**read-char-no-hang**	**read-preserving-whitespace**
read	**read-delimited-list**	**unread-char**

Figure 21–2. Operators relating to Input Streams.

Figure 21–3 shows *operators* relating to *output streams*.

clear-output	**prin1**	**write**
finish-output	**prin1-to-string**	**write-byte**
force-output	**princ**	**write-char**
format	**princ-to-string**	**write-line**
fresh-line	**print**	**write-string**
pprint	**terpri**	**write-to-string**

Figure 21–3. Operators relating to Output Streams.

A *stream* that is both an *input stream* and an *output stream* is called a ***bidirectional stream***. See the *functions* **input-stream-p** and **output-stream-p**.

Any of the *operators* listed in Figure 21–2 or Figure 21–3 can be used with *bidirectional streams*. In addition, Figure 21–4 shows a list of *operators* that relate specificaly to *bidirectional streams*.

y-or-n-p	**yes-or-no-p**

Figure 21–4. Operators relating to Bidirectional Streams.

21.1.1.1.2 Open and Closed Streams

Streams are either ***open*** or ***closed***.

Except as explicitly specified otherwise, operations that create and return *streams* return *open streams*.

The action of *closing* a *stream* marks the end of its use as a source or sink of data, permitting the *implementation* to reclaim its internal data structures, and to free any external resources which might have been locked by the *stream* when it was opened.

Except as explicitly specified otherwise, the consequences are undefined when a *closed stream* is used where a *stream* is called for.

Coercion of *streams* to *pathnames* is permissible for *closed streams*; in some situations, such as for a *truename* computation, the result might be different for an *open stream* and for that same *stream* once it has been *closed*.

21.1.1.1.3 Interactive Streams

An ***interactive stream*** is one on which it makes sense to perform interactive querying.

The precise meaning of an *interactive stream* is *implementation-defined*, and may depend on the underlying operating system. Some examples of the things that an *implementation* might choose to use as identifying characteristics of an *interactive stream* include:

- The *stream* is connected to a person (or equivalent) in such a way that the program can prompt for information and expect to receive different input depending on the prompt.

- The program is expected to prompt for input and support "normal input editing".

- **read-char** might wait for the user to type something before returning instead of immediately returning a character or end-of-file.

The general intent of having some *streams* be classified as *interactive streams* is to allow them to be distinguished from streams containing batch (or background or command-file) input. Output to batch streams is typically discarded or saved for later viewing, so interactive queries to such streams might not have the expected effect.

Terminal I/O might or might not be an *interactive stream*.

21.1.1.2 Abstract Classifications of Streams

21.1.1.2.1 File Streams

Some *streams*, called ***file streams***, provide access to *files*. An *object* of *class* **file-stream** is used to represent a *file stream*.

The basic operation for opening a *file* is **open**, which typically returns a *file stream* (see its dictionary entry for details). The basic operation for closing a *stream* is **close**. The macro **with-open-file** is useful to express the common idiom of opening a *file* for the duration of a given body of *code*, and assuring that the resulting *stream* is closed upon exit from that body.

21.1.1.3 Other Subclasses of Stream

The *class* **stream** has a number of *subclasses* defined by this specification. Figure 21–5 shows some information about these subclasses.

Class	Related Operators
broadcast-stream	**make-broadcast-stream**
	broadcast-stream-streams
concatenated-stream	**make-concatenated-stream**
	concatenated-stream-streams
echo-stream	**make-echo-stream**
	echo-stream-input-stream
	echo-stream-output-stream
string-stream	**make-string-input-stream**
	with-input-from-string
	make-string-output-stream
	with-output-to-string
	get-output-stream-string
synonym-stream	**make-synonym-stream**
	synonym-stream-symbol
two-way-stream	**make-two-way-stream**
	two-way-stream-input-stream
	two-way-stream-output-stream

Figure 21–5. Defined Names related to Specialized Streams

21.1.2 Stream Variables

Variables whose *values* must be *streams* are sometimes called ***stream variables***.

Certain *stream variables* are defined by this specification to be the proper source of input or output in various *situations* where no specific *stream* has been specified instead. A complete list of such *standardized stream variables* appears in Figure 21–6. The consequences are undefined if at any time the *value* of any of these *variables* is not an *open stream*.

Glossary Term	Variable Name
debug I/O	***debug-io***
error output	***error-output***
query I/O	***query-io***
standard input	***standard-input***
standard output	***standard-output***
terminal I/O	***terminal-io***
trace output	***trace-output***

Figure 21–6. Standardized Stream Variables

Note that, by convention, *standardized stream variables* have names ending in "`-input*`" if they must be *input streams*, ending in "-output*" if they must be *output streams*, or ending in "-io*" if they must be *bidirectional streams*.

User programs may *assign* or *bind* any *standardized stream variable* except ***terminal-io***.

21.1.3 Stream Arguments to Standardized Functions

The *operators* in Figure 21–7 accept *stream arguments* that might be either *open* or *closed streams*.

broadcast-stream-streams	**file-author**	**pathnamep**
close	**file-namestring**	**probe-file**
compile-file	**file-write-date**	**rename-file**
compile-file-pathname	**host-namestring**	**streamp**
concatenated-stream-streams	**load**	**synonym-stream-symbol**
delete-file	**logical-pathname**	**translate-logical-pathname**
directory	**merge-pathnames**	**translate-pathname**
directory-namestring	**namestring**	**truename**
dribble	**open**	**two-way-stream-input-stream**
echo-stream-input-stream	**open-stream-p**	**two-way-stream-output-stream**
echo-stream-ouput-stream	**parse-namestring**	**wild-pathname-p**
ed	**pathname**	**with-open-file**
enough-namestring	**pathname-match-p**	

Figure 21–7. Operators that accept either Open or Closed Streams

The *operators* in Figure 21–8 accept *stream arguments* that must be *open streams*.

clear-input	**output-stream-p**	**read-char-no-hang**
clear-output	**peek-char**	**read-delimited-list**
file-length	**pprint**	**read-line**
file-position	**pprint-fill**	**read-preserving-whitespace**
file-string-length	**pprint-indent**	**stream-element-type**
finish-output	**pprint-linear**	**stream-external-format**
force-output	**pprint-logical-block**	**terpri**
format	**pprint-newline**	**unread-char**
fresh-line	**pprint-tab**	**with-open-stream**
get-output-stream-string	**pprint-tabular**	**write**
input-stream-p	**prin1**	**write-byte**
interactive-stream-p	**princ**	**write-char**
listen	**print**	**write-line**
make-broadcast-stream	**print-object**	**write-string**
make-concatenated-stream	**print-unreadable-object**	**y-or-n-p**
make-echo-stream	**read**	**yes-or-no-p**
make-synonym-stream	**read-byte**	
make-two-way-stream	**read-char**	

Figure 21–8. Operators that accept Open Streams only

21.1.4 Restrictions on Composite Streams

The consequences are undefined if any *component* of a *composite stream* is *closed* before the *composite stream* is *closed*.

The consequences are undefined if the *synonym stream symbol* is not *bound* to an *open stream* from the time of the *synonym stream*'s creation until the time it is *closed*.

stream *System Class*

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:

Section 21.1 (Stream Concepts), Section 22.1.3.13 (Printing Other Objects), Chapter 22 (Printer), Chapter 23 (Reader)

broadcast-stream *System Class*

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**

concatenated-stream *System Class*

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**

echo-stream *System Class*

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**

file-stream *System Class*

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**

string-stream *System Class*

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**

synonym-stream *System Class*

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**

two-way-stream *System Class*

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**

input-stream-p, output-stream-p *Function*

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
```

Exceptional Situations:

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

interactive-stream-p *Function*

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))))
```

Exceptional Situations:

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

See Also:

Section 21.1 (Stream Concepts)

open-stream-p *Function*

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
```

Affected By:

close.

Exceptional Situations:

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

stream-element-type *Function*

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))
```

Exceptional Situations:

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

streamp *Function*

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
```

Notes:

```
(streamp object) ≡ (typep object 'stream)
```

read-byte *Function*

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*.

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**

write-byte *Function*

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**

peek-char *Function*

Syntax:

peek-char &optional *peek-type input-stream eof-error-p* $\rightarrow$ *char*
eof-value recursive-p

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.

read-char *Function*

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 " 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**.

read-char-no-hang *Function*

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↩
→ (#\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.

terpri, fresh-line *Function*

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)
    (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.

Notes:

terpri is identical in effect to

```
(write-char #\Newline output-stream)
```

unread-char *Function*

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***.

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*.

write-char *Function*

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***.

See Also:

read-char, **write-byte**, **write-sequence**

read-line

read-line *Function*

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
 (read-line input-stream nil nil)
→ NIL, true
```

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**.

write-string, write-line *Function*

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
```

Affected By:

standard-output, ***terminal-io***.

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)
≡ (prog1 (write-string string) (terpri))
```

read-sequence *Function*

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*.

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:

Section 3.2.1 (Compiler Terminology), **write-sequence**, **read-line**

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*.

write-sequence *Function*

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*.

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:

Section 3.2.1 (Compiler Terminology), **read-sequence**, **write-string**, **write-line**

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*.

file-length *Function*

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"
```

```
                    :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
```

Exceptional Situations:

Should signal an error of *type* **type-error** if *stream* is not a *stream associated with a file*.

See Also:

open

file-position *Function*

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.

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

file-position

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:

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.

file-string-length *Function*

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.

open

open *Function*

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.

open

`:overwrite`

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...>
 (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**.

stream-external-format *Function*

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>
```

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*.

with-open-file *macro*

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.

with-open-file

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"))
→ #<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)

close *Function*

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.

See Also:

open

with-open-stream *Macro*

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).

See Also:

close

listen *Function*

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!
```

Affected By:

standard-input

See Also:

interactive-stream-p, **read-char-no-hang**

clear-input

clear-input *Function*

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:

```
;; 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

finish-output, force-output, clear-output *Function*

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
```

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

y-or-n-p, yes-or-no-p *Function*

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?")
▷ 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.

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.

make-synonym-stream *Function*

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:

```
 (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
```

Exceptional Situations:

Should signal **type-error** if its argument is not a *symbol*.

See Also:

Section 21.1 (Stream Concepts)

synonym-stream-symbol *Function*

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.

See Also:

make-synonym-stream

broadcast-stream-streams *Function*

Syntax:

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.

make-broadcast-stream *Function*

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"
```

Exceptional Situations:

Should signal an error of *type* **type-error** if any *stream* is not an *output stream*.

See Also:

broadcast-stream-streams

make-two-way-stream *Function*

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...
```

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*.

two-way-stream-input-stream, two-way-stream-output-stream *Function*

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.

echo-stream-input-stream, echo-stream-output-stream *Function*

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.

make-echo-stream *Function*

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"
```

See Also:

echo-stream-input-stream, **echo-stream-output-stream**, **make-two-way-stream**

concatenated-stream-streams *Function*

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.

make-concatenated-stream *Function*

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
```

Exceptional Situations:

Should signal **type-error** if any argument is not an *input stream*.

See Also:

concatenated-stream-streams

get-output-stream-string *Function*

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.

Exceptional Situations:

The consequences are undefined if *stream-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

make-string-input-stream *Function*

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
```

See Also:

with-input-from-string

make-string-output-stream *Function*

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"
```

None..

See Also:

get-output-stream-string, **with-output-to-string**

with-input-from-string *Macro*

Syntax:

with-input-from-string (*var string* &key *index start end*) {*declaration*}* {*form*}*
→ {*result*}*

Arguments and Values:

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.

See Also:

make-string-input-stream, Section 3.6 (Traversal Rules and Side Effects)

with-output-to-string *Macro*

Syntax:

with-output-to-string (*var* `&optional` *string-form* `&key` *element-type*) {*declaration*}* {*form*}*
→ {*result*}*

Arguments and Values:

var—a *variable name*.

with-output-to-string

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.

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)

∗debug-io∗, ∗error-output∗, ∗query-io∗, ∗standard-input∗, ∗standard-output∗, ∗trace-output∗ *Variable*

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.

debug-io, *error-output*, *query-io*, ...

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**, Chapter 9 (Conditions), Chapter 23 (Reader), Chapter 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.

∗terminal-io∗ *Variable*

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 (prin1 'foo) (prin1 'bar *terminal-io*))
```

```
▷ FOOBAR
→ BAR
 (with-output-to-string (*standard-output*)
   (prin1 'foo)
   (prin1 'bar *terminal-io*))
▷ BAR
→ "FOO"
```

See Also:

debug-io, ***error-output***, ***query-io***, ***standard-input***, ***standard-output***, ***trace-output***

stream-error *Condition Type*

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 `:stream` initialization argument to **make-condition**, and is *accessed* by the *function* **stream-error-stream**.

See Also:

stream-error-stream

stream-error-stream *Function*

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>."
```

See Also:

stream-error, Chapter 9 (Conditions)

end-of-file *Condition Type*

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

Programming Language—Common Lisp

22. Printer

22.1 The Lisp Printer

22.1.1 Overview of The Lisp Printer

Common Lisp provides a representation of most *objects* in the form of printed text called the printed representation. Functions such as **print** take an *object* and send the characters of its printed representation to a *stream*. The collection of routines that does this is known as the (Common Lisp) printer.

Reading a printed representation typically produces an *object* that is **equal** to the originally printed *object*.

22.1.1.1 Multiple Possible Textual Representations

Most *objects* have more than one possible textual representation. For example, the positive *integer* with a magnitude of twenty-seven can be textually expressed in any of these ways:

```
 27    27.    #o33    #x1B    #b11011    #.(* 3 3 3)    81/3
```

A list containing the two symbols `A` and `B` can also be textually expressed in a variety of ways:

```
 (A B)    (a b)    (  a  b )    (\A |B|)
(|\A|
  B
)
```

In general, from the point of view of the *Lisp reader*, wherever *whitespace* is permissible in a textual representation, any number of *spaces* and *newlines* can appear in *standard syntax*.

When a function such as **print** produces a printed representation, it must choose from among many possible textual representations. In most cases, it chooses a program readable representation, but in certain cases it might use a more compact notation that is not program-readable.

A number of option variables, called ***printer control variables***, are provided to permit control of individual aspects of the printed representation of *objects*. Figure 22–1 shows the *standardized printer control variables*; there might also be *implementation-defined printer control variables*.

print-array	***print-gensym***	***print-pprint-dispatch***
print-base	***print-length***	***print-pretty***
print-case	***print-level***	***print-radix***
print-circle	***print-lines***	***print-readably***
print-escape	***print-miser-width***	***print-right-margin***

Figure 22–1. Standardized Printer Control Variables

In addition to the *printer control variables*, the following additional *defined names* relate to or affect the behavior of the *Lisp printer*:

package	***read-eval***	**readtable-case**
read-default-float-format	***readtable***	

Figure 22–2. Additional Influences on the Lisp printer.

22.1.1.1.1 Printer Escaping

The *variable* ***print-escape*** controls whether the *Lisp printer* tries to produce notations such as escape characters and package prefixes.

The *variable* ***print-readably*** can be used to override many of the individual aspects controlled by the other *printer control variables* when program-readable output is especially important.

One of the many effects of making the *value* of ***print-readably*** be *true* is that the *Lisp printer* behaves as if ***print-escape*** were also *true*. For notational convenience, we say that if the value of either ***print-readably*** or ***print-escape*** is *true*, then ***printer escaping*** is "enabled"; and we say that if the values of both ***print-readably*** and ***print-escape*** are *false*, then *printer escaping* is "disabled".

22.1.2 Printer Dispatching

The *Lisp printer* makes its determination of how to print an *object* as follows:

If the *value* of ***print-pretty*** is *true*, printing is controlled by the *current pprint dispatch table*; see Section 22.2.1.4 (Pretty Print Dispatch Tables).

Otherwise (if the *value* of ***print-pretty*** is *false*), the object's **print-object** method is used; see Section 22.1.3 (Default Print-Object Methods).

22.1.3 Default Print-Object Methods

This section describes the default behavior of **print-object** methods for the *standardized types*.

22.1.3.1 Printing Numbers

22.1.3.1.1 Printing Integers

Integers are printed in the radix specified by the *current output base* in positional notation, most significant digit first. If appropriate, a radix specifier can be printed; see ***print-radix***. If an *integer* is negative, a minus sign is printed and then the absolute value of the *integer* is printed. The *integer* zero is represented by the single digit `0` and never has a sign. A decimal point might be printed, depending on the *value* of ***print-radix***.

For related information about the syntax of an *integer*, see Section 2.3.2.1.1 (Syntax of an Integer).

22.1.3.1.2 Printing Ratios

Ratios are printed as follows: the absolute value of the numerator is printed, as for an *integer*; then a `/`; then the denominator. The numerator and denominator are both printed in the radix specified by the *current output base*; they are obtained as if by **numerator** and **denominator**, and so *ratios* are printed in reduced form (lowest terms). If appropriate, a radix specifier can be printed; see ***print-radix***. If the ratio is negative, a minus sign is printed before the numerator.

For related information about the syntax of a *ratio*, see Section 2.3.2.1.2 (Syntax of a Ratio).

22.1.3.1.3 Printing Floats

If the magnitude of the *float* is either zero or between 10^{-3} (inclusive) and 10^7 (exclusive), it is printed as the integer part of the number, then a decimal point, followed by the fractional part of the number; there is always at least one digit on each side of the decimal point. If the sign of the number (as determined by **float-sign**) is negative, then a minus sign is printed before the number. If the format of the number does not match that specified by ***read-default-float-format***, then the *exponent marker* for that format and the digit `0` are also printed. For example, the base of the natural logarithms as a *short float* might be printed as `2.71828S0`.

For non-zero magnitudes outside of the range 10^{-3} to 10^7, a *float* is printed in computerized scientific notation. The representation of the number is scaled to be between 1 (inclusive) and 10 (exclusive) and then printed, with one digit before the decimal point and at least one digit after the decimal point. Next the *exponent marker* for the format is printed, except that if the format of the number matches that specified by ***read-default-float-format***, then the *exponent marker* `E` is used. Finally, the power of ten by which the fraction must be multiplied to equal the original number is printed as a decimal integer. For example, Avogadro's number as a *short float* is printed as `6.02S23`.

For related information about the syntax of a *float*, see Section 2.3.2.2 (Syntax of a Float).

22.1.3.1.4 Printing Complexes

A *complex* is printed as `#C`, an open parenthesis, the printed representation of its real part, a space, the printed representation of its imaginary part, and finally a close parenthesis.

For related information about the syntax of a *complex*, see Section 2.3.2.3 (Syntax of a Complex) and Section 2.4.8.11 (Sharpsign C).

22.1.3.1.5 Note about Printing Numbers

The printed representation of a number must not contain *escape characters*; see Section 2.3.1.1.1 (Escape Characters and Potential Numbers).

22.1.3.2 Printing Characters

When *printer escaping* is disabled, a *character* prints as itself; it is sent directly to the output *stream*. When *printer escaping* is enabled, then `#\` syntax is used.

When the printer types out the name of a *character*, it uses the same table as the `#\` *reader macro* would use; therefore any *character* name that is typed out is acceptable as input (in that *implementation*). If a *non-graphic character* has a *standardized* $name_5$, that *name* is preferred over non-standard *names* for printing in `#\` notation. For the *graphic standard characters*, the *character* itself is always used for printing in `#\` notation—even if the *character* also has a $name_5$.

For details about the `#\` *reader macro*, see Section 2.4.8.1 (Sharpsign Backslash).

22.1.3.3 Printing Symbols

When *printer escaping* is disabled, only the characters of the *symbol*'s *name* are output (but the case in which to print characters in the *name* is controlled by ***print-case***; see Section 22.1.3.3.2 (Effect of Readtable Case on the Lisp Printer)).

The remainder of this section applies only when *printer escaping* is enabled.

When printing a *symbol*, the printer inserts enough *single escape* and/or *multiple escape* characters (*backslashes* and/or *vertical-bars*) so that if **read** were called with the same ***readtable*** and with ***read-base*** bound to the *current output base*, it would return the same *symbol* (if it is not *apparently uninterned*) or an *uninterned symbol* with the same *print name* (otherwise).

For example, if the *value* of ***print-base*** were `16` when printing the symbol `face`, it would have to be printed as `\FACE` or `\Face` or `|FACE|`, because the token `face` would be read as a hexadecimal number (decimal value 64206) if the *value* of ***read-base*** were `16`.

For additional restrictions concerning characters with nonstandard *syntax types* in the *current readtable*, see the *variable* ***print-readably***

For information about how the *Lisp reader* parses *symbols*, see Section 2.3.4 (Symbols as Tokens) and Section 2.4.8.5 (Sharpsign Colon).

nil might be printed as `()` when ***print-pretty*** is *true* and *printer escaping* is enabled.

22.1.3.3.1 Package Prefixes for Symbols

Package prefixes are printed if necessary. The rules for *package prefixes* are as follows. When the *symbol* is printed, if it is in the `KEYWORD` *package*, then it is printed with a preceding *colon*; otherwise, if it is *accessible* in the *current package*, it is printed without any *package prefix*; otherwise, it is printed with a *package prefix*.

A *symbol* that is *apparently uninterned* is printed preceded by "`#:`" if ***print-gensym*** is *true* and *printer escaping* is enabled; if ***print-gensym*** is *false* or *printer escaping* is disabled, then the *symbol* is printed without a prefix, as if it were in the *current package*.

Because the `#:` syntax does not intern the following symbol, it is necessary to use circular-list syntax if ***print-circle*** is *true* and the same uninterned symbol appears several times in an expression to be printed. For example, the result of

```
(let ((x (make-symbol "FOO"))) (list x x))
```

would be printed as `(#:foo #:foo)` if ***print-circle*** were *false*, but as `(#1=#:foo #1#)` if ***print-circle*** were *true*.

A summary of the preceding package prefix rules follows:

`foo:bar`

`foo:bar` is printed when *symbol* `bar` is external in its *home package* `foo` and is not *accessible* in the *current package*.

`foo::bar`

`foo::bar` is printed when `bar` is internal in its *home package* `foo` and is not *accessible* in the *current package*.

`:bar`

`:bar` is printed when the home package of `bar` is the `KEYWORD` *package*.

`#:bar`

`#:bar` is printed when `bar` is *apparently uninterned*, even in the pathological case that `bar` has no *home package* but is nevertheless somehow *accessible* in the *current package*.

22.1.3.3.2 Effect of Readtable Case on the Lisp Printer

When *printer escaping* is disabled, or the characters under consideration are not already quoted specifically by *single escape* or *multiple escape* syntax, the *readtable case* of the *current readtable* affects the way the *Lisp printer* writes *symbols* in the following ways:

`:upcase`

When the *readtable case* is `:upcase`, *uppercase characters* are printed in the case specified by ***print-case***, and *lowercase characters* are printed in their own case.

`:downcase`

When the *readtable case* is `:downcase`, *uppercase characters* are printed in their own case, and *lowercase characters* are printed in the case specified by ***print-case***.

`:preserve`

When the *readtable case* is `:preserve`, all *alphabetic characters* are printed in their own case.

`:invert`

When the *readtable case* is `:invert`, the case of all *alphabetic characters* in single case symbol names is inverted. Mixed-case symbol names are printed as is.

The rules for escaping *alphabetic characters* in symbol names are affected by the **readtable-case** if *printer escaping* is enabled. *Alphabetic characters* are escaped as follows:

`:upcase`

When the *readtable case* is `:upcase`, all *lowercase characters* must be escaped.

`:downcase`

When the *readtable case* is `:downcase`, all *uppercase characters* must be escaped.

`:preserve`

When the *readtable case* is `:preserve`, no *alphabetic characters* need be escaped.

`:invert`

When the *readtable case* is `:invert`, no *alphabetic characters* need be escaped.

22.1.3.3.2.1 Examples of Effect of Readtable Case on the Lisp Printer

```
(defun test-readtable-case-printing ()
  (let ((*readtable* (copy-readtable nil))
        (*print-case* *print-case*))
    (format t "READTABLE-CASE *PRINT-CASE*  Symbol-name  Output~
             ~%--------------------------~
             ~%")
    (dolist (readtable-case '(:upcase :downcase :preserve :invert))
      (setf (readtable-case *readtable*) readtable-case)
      (dolist (print-case '(:upcase :downcase :capitalize))
        (dolist (symbol '(|ZEBRA| |Zebra| |zebra|))
          (setq *print-case* print-case)
          (format t "~&:~A~15T:~A~29T~A~42T~A"
```

```
                   (string-upcase readtable-case)
                   (string-upcase print-case)
                   (symbol-name symbol)
                   (prin1-to-string symbol)))))))
```

The output from (test-readtable-case-printing) should be as follows:

```
READTABLE-CASE *PRINT-CASE*  Symbol-name  Output
--------------------------
:UPCASE        :UPCASE       ZEBRA        ZEBRA
:UPCASE        :UPCASE       Zebra        |Zebra|
:UPCASE        :UPCASE       zebra        |zebra|
:UPCASE        :DOWNCASE     ZEBRA        zebra
:UPCASE        :DOWNCASE     Zebra        |Zebra|
:UPCASE        :DOWNCASE     zebra        |zebra|
:UPCASE        :CAPITALIZE   ZEBRA        Zebra
:UPCASE        :CAPITALIZE   Zebra        |Zebra|
:UPCASE        :CAPITALIZE   zebra        |zebra|
:DOWNCASE      :UPCASE       ZEBRA        |ZEBRA|
:DOWNCASE      :UPCASE       Zebra        |Zebra|
:DOWNCASE      :UPCASE       zebra        ZEBRA
:DOWNCASE      :DOWNCASE     ZEBRA        |ZEBRA|
:DOWNCASE      :DOWNCASE     Zebra        |Zebra|
:DOWNCASE      :DOWNCASE     zebra        zebra
:DOWNCASE      :CAPITALIZE   ZEBRA        |ZEBRA|
:DOWNCASE      :CAPITALIZE   Zebra        |Zebra|
:DOWNCASE      :CAPITALIZE   zebra        Zebra
:PRESERVE      :UPCASE       ZEBRA        ZEBRA
:PRESERVE      :UPCASE       Zebra        Zebra
:PRESERVE      :UPCASE       zebra        zebra
:PRESERVE      :DOWNCASE     ZEBRA        ZEBRA
:PRESERVE      :DOWNCASE     Zebra        Zebra
:PRESERVE      :DOWNCASE     zebra        zebra
:PRESERVE      :CAPITALIZE   ZEBRA        ZEBRA
:PRESERVE      :CAPITALIZE   Zebra        Zebra
:PRESERVE      :CAPITALIZE   zebra        zebra
:INVERT        :UPCASE       ZEBRA        zebra
:INVERT        :UPCASE       Zebra        Zebra
:INVERT        :UPCASE       zebra        ZEBRA
:INVERT        :DOWNCASE     ZEBRA        zebra
:INVERT        :DOWNCASE     Zebra        Zebra
:INVERT        :DOWNCASE     zebra        ZEBRA
:INVERT        :CAPITALIZE   ZEBRA        zebra
:INVERT        :CAPITALIZE   Zebra        Zebra
:INVERT        :CAPITALIZE   zebra        ZEBRA
```

22.1.3.4 Printing Strings

The characters of the *string* are output in order. If *printer escaping* is enabled, a *double-quote* is output before and after, and all *double-quotes* and *single escapes* are preceded by *backslash*. The printing of *strings* is not affected by ***print-array***. Only the *active elements* of the *string* are printed.

For information on how the *Lisp reader* parses *strings*, see Section 2.4.5 (Double-Quote).

22.1.3.5 Printing Lists and Conses

Wherever possible, list notation is preferred over dot notation. Therefore the following algorithm is used to print a *cons* x:

1. A *left-parenthesis* is printed.

2. The *car* of x is printed.

3. If the *cdr* of x is itself a *cons*, it is made to be the current *cons* (*i.e.*, x becomes that *cons*), a *space* is printed, and step 2 is re-entered.

4. If the *cdr* of x is not *null*, a *space*, a *dot*, a *space*, and the *cdr* of x are printed.

5. A *right-parenthesis* is printed.

Actually, the above algorithm is only used when ***print-pretty*** is *false*. When ***print-pretty*** is *true* (or when **pprint** is used), additional *whitespace*$_1$ may replace the use of a single *space*, and a more elaborate algorithm with similar goals but more presentational flexibility is used; see Section 22.1.2 (Printer Dispatching).

Although the two expressions below are equivalent, and the reader accepts either one and produces the same *cons*, the printer always prints such a *cons* in the second form.

```
(a . (b . ((c . (d . nil)) . (e . nil))))
(a b (c d) e)
```

The printing of *conses* is affected by ***print-level***, ***print-length***, and ***print-circle***.

Following are examples of printed representations of *lists*:

```
(a . b)     ;A dotted pair of a and b
(a.b)       ;A list of one element, the symbol named a.b
(a. b)      ;A list of two elements a. and b
(a .b)      ;A list of two elements a and .b
```

```
(a b . c)    ;A dotted list of a and b with c at the end; two conses
.iot         ;The symbol whose name is .iot
(. b)        ;Invalid - an error is signaled if an attempt is made to read
             ;this syntax.
(a .)        ;Invalid - an error is signaled.
(a .. b)     ;Invalid - an error is signaled.
(a . . b)    ;Invalid - an error is signaled.
(a b c ...)  ;Invalid - an error is signaled.
(a \. b)     ;A list of three elements a, ., and b
(a |.| b)    ;A list of three elements a, ., and b
(a \... b)   ;A list of three elements a, ..., and b
(a |...| b)  ;A list of three elements a, ..., and b
```

For information on how the *Lisp reader* parses *lists* and *conses*, see Section 2.4.1 (Left-Parenthesis).

22.1.3.6 Printing Bit Vectors

A *bit vector* is printed as `#*` followed by the bits of the *bit vector* in order. If ***print-array*** is *false*, then the *bit vector* is printed in a format (using `#<`) that is concise but not readable. Only the *active elements* of the *bit vector* are printed.

For information on *Lisp reader* parsing of *bit vectors*, see Section 2.4.8.4 (Sharpsign Asterisk).

22.1.3.7 Printing Other Vectors

If ***print-array*** is *true* and ***print-readably*** is *false*, any *vector* other than a *string* or *bit vector* is printed using general-vector syntax; this means that information about specialized vector representations does not appear. The printed representation of a zero-length *vector* is `#()`. The printed representation of a non-zero-length *vector* begins with `#(`. Following that, the first element of the *vector* is printed. If there are any other elements, they are printed in turn, with each such additional element preceded by a *space* if ***print-pretty*** is *false*, or *whitespace*$_1$ if ***print-pretty*** is *true*. A *right-parenthesis* after the last element terminates the printed representation of the *vector*. The printing of *vectors* is affected by ***print-level*** and ***print-length***. If the *vector* has a *fill pointer*, then only those elements below the *fill pointer* are printed.

If both ***print-array*** and ***print-readably*** are *false*, the *vector* is not printed as described above, but in a format (using `#<`) that is concise but not readable.

If ***print-readably*** is *true*, the *vector* prints in an *implementation-defined* manner; see the *variable* ***print-readably***.

For information on how the *Lisp reader* parses these "other *vectors*," see Section 2.4.8.3 (Sharpsign Left-Parenthesis).

22.1.3.8 Printing Other Arrays

If ***print-array*** is *true* and ***print-readably*** is *false*, any *array* other than a *vector* is printed using `#nA` format. Let `n` be the *rank* of the *array*. Then `#` is printed, then `n` as a decimal integer, then `A`, then `n` open parentheses. Next the *elements* are scanned in row-major order, using **write** on each *element*, and separating *elements* from each other with *whitespace*$_1$. The array's dimensions are numbered 0 to `n`-1 from left to right, and are enumerated with the rightmost index changing fastest. Every time the index for dimension `j` is incremented, the following actions are taken:

- If `j` < `n`-1, then a close parenthesis is printed.
- If incrementing the index for dimension `j` caused it to equal dimension `j`, that index is reset to zero and the index for dimension `j`-1 is incremented (thereby performing these three steps recursively), unless `j`=0, in which case the entire algorithm is terminated. If incrementing the index for dimension `j` did not cause it to equal dimension `j`, then a space is printed.
- If `j` < `n`-1, then an open parenthesis is printed.

This causes the contents to be printed in a format suitable for `:initial-contents` to **make-array**. The lists effectively printed by this procedure are subject to truncation by ***print-level*** and ***print-length***.

If the *array* is of a specialized *type*, containing bits or characters, then the innermost lists generated by the algorithm given above can instead be printed using bit-vector or string syntax, provided that these innermost lists would not be subject to truncation by ***print-length***.

If both ***print-array*** and ***print-readably*** are *false*, then the *array* is printed in a format (using `#<`) that is concise but not readable.

If ***print-readably*** is *true*, the *array* prints in an *implementation-defined* manner; see the *variable* ***print-readably***. In particular, this may be important for arrays having some dimension `0`.

For information on how the *Lisp reader* parses these "other *arrays*," see Section 2.4.8.12 (Sharpsign A).

22.1.3.9 Examples of Printing Arrays

```
 (let ((a (make-array '(3 3)))
       (*print-pretty* t)
       (*print-array* t))
   (dotimes (i 3) (dotimes (j 3) (setf (aref a i j) (format nil "<~D,~D>" i j))))
   (print a)
   (print (make-array 9 :displaced-to a)))
▷ #2A(("<0,0>" "<0,1>" "<0,2>")
▷     ("<1,0>" "<1,1>" "<1,2>")
```

```
▷     ("<2,0>" "<2,1>" "<2,2>"))
▷ #("<0,0>" "<0,1>" "<0,2>" "<1,0>" "<1,1>" "<1,2>" "<2,0>" "<2,1>" "<2,2>")
→ #<ARRAY 9 indirect 36363476>
```

22.1.3.10 Printing Random States

A specific syntax for printing *objects* of *type* **random-state** is not specified. However, every *implementation* must arrange to print a *random state object* in such a way that, within the same implementation, **read** can construct from the printed representation a copy of the *random state* object as if the copy had been made by **make-random-state**.

If the type *random state* is effectively implemented by using the machinery for **defstruct**, the usual structure syntax can then be used for printing *random state* objects; one might look something like

```
#S(RANDOM-STATE :DATA #(14 49 98436589 786345 8734658324 ... ))
```

where the components are *implementation-dependent*.

22.1.3.11 Printing Pathnames

When *printer escaping* is enabled, the syntax `#P"..."` is how a *pathname* is printed by **write** and the other functions herein described. The `"..."` is the namestring representation of the pathname.

When *printer escaping* is disabled, **write** writes a *pathname* *P* by writing `(namestring` *P*`)` instead.

For information on how the *Lisp reader* parses *pathnames*, see Section 2.4.8.14 (Sharpsign P).

22.1.3.12 Printing Structures

By default, a *structure* of type *S* is printed using `#S` syntax. This behavior can be customized by specifying a `:print-function` or `:print-object` option to the **defstruct** *form* that defines *S*, or by writing a **print-object** *method* that is *specialized* for *objects* of type *S*.

Different structures might print out in different ways; the default notation for structures is:

`#S(`*structure-name* {*slot-key* *slot-value*}*`)`

where `#S` indicates structure syntax, *structure-name* is a *structure name*, each *slot-key* is an initialization argument *name* for a *slot* in the *structure*, and each corresponding *slot-value* is a representation of the *object* in that *slot*.

For information on how the *Lisp reader* parses *structures*, see Section 2.4.8.13 (Sharpsign S).

22.1.3.13 Printing Other Objects

Other *objects* are printed in an *implementation-dependent* manner. It is not required that an *implementation* print those *objects readably*.

For example, *hash tables*, *readtables*, *packages*, *streams*, and *functions* might not print *readably*.

A common notation to use in this circumstance is `#<...>`. Since `#<` is not readable by the *Lisp reader*, the precise format of the text which follows is not important, but a common format to use is that provided by the **print-unreadable-object** *macro*.

For information on how the *Lisp reader* treats this notation, see Section 2.4.8.20 (Sharpsign Less-Than-Sign). For information on how to notate *objects* that cannot be printed *readably*, see Section 2.4.8.6 (Sharpsign Dot).

22.1.4 Examples of Printer Behavior

```
 (let ((*print-escape* t)) (fresh-line) (write #\a))
▷ #\a
→ #\a
 (let ((*print-escape* nil) (*print-readably* nil))
   (fresh-line)
   (write #\a))
▷ a
→ #\a
 (progn (fresh-line) (prin1 #\a))
▷ #\a
→ #\a
 (progn (fresh-line) (print #\a))
▷
▷ #\a
→ #\a
 (progn (fresh-line) (princ #\a))
▷ a
→ #\a
```

```
 (dolist (val '(t nil))
   (let ((*print-escape* val) (*print-readably* val))
     (print '#\a)
     (prin1 #\a) (write-char #\Space)
     (princ #\a) (write-char #\Space)
     (write #\a)))
▷ #\a #\a a #\a
```

```
▷ #\a #\a a a
→ NIL
```

```
 (progn (fresh-line) (write '(let ((a 1) (b 2)) (+ a b))))
▷ (LET ((A 1) (B 2)) (+ A B))
→ (LET ((A 1) (B 2)) (+ A B))
```

```
 (progn (fresh-line) (pprint '(let ((a 1) (b 2)) (+ a b))))
▷ (LET ((A 1)
▷       (B 2))
▷   (+ A B))
→ (LET ((A 1) (B 2)) (+ A B))
```

```
 (progn (fresh-line)
        (write '(let ((a 1) (b 2)) (+ a b)) :pretty t))
▷ (LET ((A 1)
▷       (B 2))
▷   (+ A B))
→ (LET ((A 1) (B 2)) (+ A B))
```

```
 (with-output-to-string (s)
    (write 'write :stream s)
    (prin1 'prin1 s))
→ "WRITEPRIN1"
```

22.2 The Lisp Pretty Printer

22.2.1 Pretty Printer Concepts

The facilities provided by the ***pretty printer*** permit *programs* to redefine the way in which *code* is displayed, and allow the full power of *pretty printing* to be applied to complex combinations of data structures.

Whether any given style of output is in fact "pretty" is inherently a somewhat subjective issue. However, since the effect of the *pretty printer* can be customized by *conforming programs*, the necessary flexibility is provided for individual *programs* to achieve an arbitrary degree of aesthetic control.

By providing direct access to the mechanisms within the pretty printer that make dynamic decisions about layout, the macros and functions **pprint-logical-block**, **pprint-newline**, and **pprint-indent** make it possible to specify pretty printing layout rules as a part of any function that produces output. They also make it very easy for the detection of circularity and sharing, and abbreviation based on length and nesting depth to be supported by the function.

The *pretty printer* is driven entirely by dispatch based on the *value* of ***print-pprint-dispatch***. The *function* **set-pprint-dispatch** makes it possible for *conforming programs* to associate new pretty printing functions with a *type*.

22.2.1.1 Dynamic Control of the Arrangement of Output

The actions of the *pretty printer* when a piece of output is too large to fit in the space available can be precisely controlled. Three concepts underlie the way these operations work—***logical blocks***, ***conditional newlines***, and ***sections***. Before proceeding further, it is important to define these terms.

The first line of Figure 22–3 shows a schematic piece of output. Each of the characters in the output is represented by "-". The positions of conditional newlines are indicated by digits. The beginnings and ends of logical blocks are indicated by "<" and ">" respectively.

The output as a whole is a logical block and the outermost section. This section is indicated by the `0`'s on the second line of Figure 1. Logical blocks nested within the output are specified by the macro **pprint-logical-block**. Conditional newline positions are specified by calls to **pprint-newline**. Each conditional newline defines two sections (one before it and one after it) and is associated with a third (the section immediately containing it).

The section after a conditional newline consists of: all the output up to, but not including, (a) the next conditional newline immediately contained in the same logical block; or if (a) is not applicable, (b) the next newline that is at a lesser level of nesting in logical blocks; or if (b) is not applicable, (c) the end of the output.

The section before a conditional newline consists of: all the output back to, but not including, (a) the previous conditional newline that is immediately contained in the same logical block; or if (a) is not applicable, (b) the beginning of the immediately containing logical block. The last four lines in Figure 1 indicate the sections before and after the four conditional newlines.

The section immediately containing a conditional newline is the shortest section that contains the conditional newline in question. In Figure 22–3, the first conditional newline is immediately contained in the section marked with 0's, the second and third conditional newlines are immediately contained in the section before the fourth conditional newline, and the fourth conditional newline is immediately contained in the section after the first conditional newline.

```
<-1---<-<--2--3->-4-->->
000000000000000000000000000
11 111111111111111111111111
          22 222
             333 3333
       44444444444444 44444
```

Figure 22–3. Example of Logical Blocks, Conditional Newlines, and Sections

Whenever possible, the pretty printer displays the entire contents of a section on a single line. However, if the section is too long to fit in the space available, line breaks are inserted at conditional newline positions within the section.

22.2.1.2 Format Directive Interface

The primary interface to operations for dynamically determining the arrangement of output is provided through the functions and macros of the pretty printer. Figure 22–4 shows the defined names related to *pretty printing*.

print-lines	**pprint-dispatch**	**pprint-pop**
print-miser-width	**pprint-exit-if-list-exhausted**	**pprint-tab**
print-pprint-dispatch	**pprint-fill**	**pprint-tabular**
print-right-margin	**pprint-indent**	**set-pprint-dispatch**
copy-pprint-dispatch	**pprint-linear**	**write**
format	**pprint-logical-block**	
formatter	**pprint-newline**	

Figure 22–4. Defined names related to pretty printing.

Figure 22–5 identifies a set of *format directives* which serve as an alternate interface to the same pretty printing operations in a more textually compact form.

`~I`	`~W`	`~<...~:>`
`~:T`	`~/.../`	`~_`

Figure 22–5. Format directives related to Pretty Printing

22.2.1.3 Compiling Format Strings

A *format string* is essentially a program in a special-purpose language that performs printing, and that is interpreted by the *function* **format**. The **formatter** *macro* provides the efficiency of using a *compiled function* to do that same printing but without losing the textual compactness of *format strings*.

A ***format control*** is either a *format string* or a *function* that was returned by the the **formatter** *macro*.

22.2.1.4 Pretty Print Dispatch Tables

A ***pprint dispatch table*** is a mapping from keys to pairs of values. Each key is a *type specifier*. The values associated with a key are a "function" (specifically, a *function designator* or **nil**) and a "numerical priority" (specifically, a *real*). Basic insertion and retrieval is done based on the keys with the equality of keys being tested by **equal**.

When ***print-pretty*** is *true*, the ***current pprint dispatch table*** (in ***print-pprint-dispatch***) controls how *objects* are printed. The information in this table takes precedence over all other mechanisms for specifying how to print *objects*. In particular, it has priority over user-defined **print-object** *methods* because the *current pprint dispatch table* is consulted first.

The function is chosen from the *current pprint dispatch table* by finding the highest priority function that is associated with a *type specifier* that matches the *object*; if there is more than one such function, it is *implementation-dependent* which is used.

However, if there is no information in the table about how to *pretty print* a particular kind of *object*, a *function* is invoked which uses **print-object** to print the *object*. The value of ***print-pretty*** is still *true* when this function is *called*, and individual methods for **print-object** might still elect to produce output in a special format conditional on the *value* of ***print-pretty***.

22.2.1.5 Pretty Printer Margins

A primary goal of pretty printing is to keep the output between a pair of margins. The column where the output begins is taken as the left margin. If the current column cannot be determined at the time output begins, the left margin is assumed to be zero. The right margin is controlled by ***print-right-margin***.

22.2.2 Examples of using the Pretty Printer

As an example of the interaction of logical blocks, conditional newlines, and indentation, consider the function `simple-pprint-defun` below. This function prints out lists whose *cars* are **defun** in the standard way assuming that the list has exactly length `4`.

```
(defun simple-pprint-defun (*standard-output* list)
  (pprint-logical-block (*standard-output* list :prefix "(" :suffix ")")
    (write (first list))
```

```
    (write-char #\Space)
    (pprint-newline :miser)
    (pprint-indent :current 0)
    (write (second list))
    (write-char #\Space)
    (pprint-newline :fill)
    (write (third list))
    (pprint-indent :block 1)
    (write-char #\Space)
    (pprint-newline :linear)
    (write (fourth list))))
```

Suppose that one evaluates the following:

```
(simple-pprint-defun *standard-output* '(defun prod (x y) (* x y)))
```

If the line width available is greater than or equal to 26, then all of the output appears on one line. If the line width available is reduced to 25, a line break is inserted at the linear-style conditional newline before the *expression* (* x y), producing the output shown. The (pprint-indent :block 1) causes (* x y) to be printed at a relative indentation of 1 in the logical block.

```
 (DEFUN PROD (X Y)
   (* X Y))
```

If the line width available is 15, a line break is also inserted at the fill style conditional newline before the argument list. The call on (pprint-indent :current 0) causes the argument list to line up under the function name.

```
(DEFUN PROD
       (X Y)
  (* X Y))
```

If ***print-miser-width*** were greater than or equal to 14, the example output above would have been as follows, because all indentation changes are ignored in miser mode and line breaks are inserted at miser-style conditional newlines.

```
 (DEFUN
  PROD
  (X Y)
  (* X Y))
```

As an example of a per-line prefix, consider that evaluating the following produces the output shown with a line width of 20 and ***print-miser-width*** of **nil**.

```
 (pprint-logical-block (*standard-output* nil :per-line-prefix ";;; ")
   (simple-pprint-defun *standard-output* '(defun prod (x y) (* x y))))

 ;;; (DEFUN PROD
```

```
;;;        (X Y)
;;;   (* X Y))
```

As a more complex (and realistic) example, consider the function `pprint-let` below. This specifies how to print a **let** *form* in the traditional style. It is more complex than the example above, because it has to deal with nested structure. Also, unlike the example above it contains complete code to readably print any possible list that begins with the *symbol* **let**. The outermost **pprint-logical-block** *form* handles the printing of the input list as a whole and specifies that parentheses should be printed in the output. The second **pprint-logical-block** *form* handles the list of binding pairs. Each pair in the list is itself printed by the innermost **pprint-logical-block**. (A **loop** *form* is used instead of merely decomposing the pair into two *objects* so that readable output will be produced no matter whether the list corresponding to the pair has one element, two elements, or (being malformed) has more than two elements.) A space and a fill-style conditional newline are placed after each pair except the last. The loop at the end of the topmost **pprint-logical-block** *form* prints out the forms in the body of the **let** *form* separated by spaces and linear-style conditional newlines.

```
(defun pprint-let (*standard-output* list)
  (pprint-logical-block (nil list :prefix "(" :suffix ")")
    (write (pprint-pop))
    (pprint-exit-if-list-exhausted)
    (write-char #\Space)
    (pprint-logical-block (nil (pprint-pop) :prefix "(" :suffix ")")
      (pprint-exit-if-list-exhausted)
      (loop (pprint-logical-block (nil (pprint-pop) :prefix "(" :suffix ")")
              (pprint-exit-if-list-exhausted)
              (loop (write (pprint-pop))
                    (pprint-exit-if-list-exhausted)
                    (write-char #\Space)
                    (pprint-newline :linear)))
            (pprint-exit-if-list-exhausted)
            (write-char #\Space)
            (pprint-newline :fill)))
    (pprint-indent :block 1)
    (loop (pprint-exit-if-list-exhausted)
          (write-char #\Space)
          (pprint-newline :linear)
          (write (pprint-pop)))))
```

Suppose that one evaluates the following with ***print-level*** being 4, and ***print-circle*** being *true*.

```
(pprint-let *standard-output*
            '#1=(let (x (*print-length* (f (g 3)))
                      (z . 2) (k (car y)))
                  (setq x (sqrt z)) #1#))
```

If the line length is greater than or equal to `77`, the output produced appears on one line. However, if the line length is 76, line breaks are inserted at the linear-style conditional newlines separating the forms in the body and the output below is produced. Note that, the degenerate binding pair x is printed readably even though it fails to be a list; a depth abbreviation marker is printed in place of (g 3); the binding pair (z . 2) is printed readably even though it is not a proper list; and appropriate circularity markers are printed.

```
#1=(LET (X (*PRINT-LENGTH* (F #)) (Z . 2) (K (CAR Y)))
     (SETQ X (SQRT Z))
     #1#)
```

If the line length is reduced to 35, a line break is inserted at one of the fill-style conditional newlines separating the binding pairs.

```
#1=(LET (X (*PRINT-PRETTY* (F #))
         (Z . 2) (K (CAR Y)))
     (SETQ X (SQRT Z))
     #1#)
```

Suppose that the line length is further reduced to 22 and ***print-length*** is set to 3. In this situation, line breaks are inserted after both the first and second binding pairs. In addition, the second binding pair is itself broken across two lines. Clause (b) of the description of fill-style conditional newlines (see the *function* **pprint-newline**) prevents the binding pair (z . 2) from being printed at the end of the third line. Note that the length abbreviation hides the circularity from view and therefore the printing of circularity markers disappears.

```
(LET (X
      (*PRINT-LENGTH*
       (F #))
      (Z . 2) ...)
  (SETQ X (SQRT Z))
  ...)
```

The next function prints a vector using "#(...)" notation.

```
(defun pprint-vector (*standard-output* v)
  (pprint-logical-block (nil nil :prefix "#(" :suffix ")")
    (let ((end (length v)) (i 0))
      (when (plusp end)
        (loop (pprint-pop)
              (write (aref v i))
              (if (= (incf i) end) (return nil))
              (write-char #\Space)
              (pprint-newline :fill))))))
```

Evaluating the following with a line length of 15 produces the output shown.

```
(pprint-vector *standard-output* '#(12 34 567 8 9012 34 567 89 0 1 23))
```

```
#(12 34 567 8
  9012 34 567
  89 0 1 23)
```

As examples of the convenience of specifying pretty printing with *format strings*, consider that the functions `simple-pprint-defun` and `pprint-let` used as examples above can be compactly defined as follows. (The function `pprint-vector` cannot be defined using **format** because the data structure it traverses is not a list.)

```
(defun simple-pprint-defun (*standard-output* list)
  (format T "~:<~W ~@_~:I~W ~:_~W~1I ~_~W~:>" list))

(defun pprint-let (*standard-output* list)
  (format T "~:<~W~^~:<~@{~:<~@{~W~^~_~}~:>~^~:_~}~:>~1I~@{~^~_~W~}~:>" list))
```

In the following example, the first *form* restores ***print-pprint-dispatch*** to the equivalent of its initial value. The next two forms then set up a special way to pretty print ratios. Note that the more specific *type specifier* has to be associated with a higher priority.

```
(setq *print-pprint-dispatch* (copy-pprint-dispatch nil))

(set-pprint-dispatch 'ratio
  #'(lambda (s obj)
      (format s "#.(/ ~W ~W)"
                (numerator obj) (denominator obj))))

(set-pprint-dispatch '(and ratio (satisfies minusp))
  #'(lambda (s obj)
      (format s "#.(- (/ ~W ~W))"
              (- (numerator obj)) (denominator obj)))
  5)

(pprint '(1/3 -2/3))
(#.(/ 1 3) #.(- (/ 2 3)))
```

The following two *forms* illustrate the definition of pretty printing functions for types of *code*. The first *form* illustrates how to specify the traditional method for printing quoted objects using *single-quote*. Note the care taken to ensure that data lists that happen to begin with **quote** will be printed readably. The second form specifies that lists beginning with the symbol `my-let` should print the same way that lists beginning with **let** print when the initial *pprint dispatch table* is in effect.

```
(set-pprint-dispatch '(cons (member quote)) ()
  #'(lambda (s list)
      (if (and (consp (cdr list)) (null (cddr list)))
         (funcall (formatter "'~W") s (cadr list))
         (pprint-fill s list))))
```

```
(set-pprint-dispatch '(cons (member my-let))
                     (pprint-dispatch '(let) nil))
```

The next example specifies a default method for printing lists that do not correspond to function calls. Note that the functions **pprint-linear**, **pprint-fill**, and **pprint-tabular** are all defined with optional *colon-p* and *at-sign-p* arguments so that they can be used as **pprint dispatch functions** as well as `~/.../` functions.

```
(set-pprint-dispatch '(cons (not (and symbol (satisfies fboundp))))
                     #'pprint-fill -5)

;; Assume a line length of 9
(pprint '(0 b c d e f g h i j k))
(0 b c d
 e f g h
 i j k)
```

This final example shows how to define a pretty printing function for a user defined data structure.

```
(defstruct family mom kids)

(set-pprint-dispatch 'family
  #'(lambda (s f)
      (funcall (formatter "~@<#<~;~W and ~2I~_~/pprint-fill/~;>~:>")
              s (family-mom f) (family-kids f))))
```

The pretty printing function for the structure `family` specifies how to adjust the layout of the output so that it can fit aesthetically into a variety of line widths. In addition, it obeys the printer control variables ***print-level***, ***print-length***, ***print-lines***, ***print-circle*** and ***print-escape***, and can tolerate several different kinds of malformity in the data structure. The output below shows what is printed out with a right margin of `25`, ***print-pretty*** being *true*, ***print-escape*** being *false*, and a malformed `kids` list.

```
(write (list 'principal-family
             (make-family :mom "Lucy"
                          :kids '("Mark" "Bob" . "Dan")))
       :right-margin 25 :pretty T :escape nil :miser-width nil)
(PRINCIPAL-FAMILY
 #<Lucy and
     Mark Bob . Dan>)
```

Note that a pretty printing function for a structure is different from the structure's **print-object** *method*. While **print-object** *methods* are permanently associated with a structure, pretty printing functions are stored in *pprint dispatch tables* and can be rapidly changed to reflect different printing needs. If there is no pretty printing function for a structure in the current *pprint dispatch table*, its **print-object** *method* is used instead.

22.2.3 Notes about the Pretty Printer's Background

For a background reference to the abstract concepts detailed in this section, see *XP: A Common Lisp Pretty Printing System*. The details of that paper are not binding on this document, but may be helpful in establishing a conceptual basis for understanding this material.

22.3 Formatted 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*.

The *control-string* argument to **format** is actually a *format control*. That is, it can be either a *format string* or a *function*, for example a *function* returned by the **formatter** *macro*.

If it is a *function*, the *function* is called with the appropriate output stream as its first argument and the data arguments to **format** as its remaining arguments. The function should perform whatever output is necessary and return the unused tail of the arguments (if any).

The compilation process performed by **formatter** produces a *function* that would do with its *arguments* as the **format** interpreter would do with those *arguments*.

The remainder of this section describes what happens if the *control-string* is a *format string*.

Control-string is composed of simple text (*characters*) and embedded directives.

format writes the simple text as is; each embedded directive specifies further text output that is to appear at the corresponding point within the simple text. Most directives use one or more elements of *args* to create their output.

A directive consists of a *tilde*, optional prefix parameters separated by commas, optional *colon* and *at-sign* modifiers, and a single character indicating what kind of directive this is. There is no required ordering between the *at-sign* and *colon* modifier. The *case* of the directive character is ignored. Prefix parameters are notated as signed (sign is optional) decimal numbers, or as a *single-quote* followed by a character. For example, `~5,'0d` can be used to print an *integer* in decimal radix in five columns with leading zeros, or `~5,'*d` to get leading asterisks.

In place of a prefix parameter to a directive, `V` (or `v`) can be used. In this case, **format** takes an argument from *args* as a parameter to the directive. The argument should be an *integer* or *character*. If the *arg* used by a `V` parameter is **nil**, the effect is as if the parameter had been omitted. `#` can be used in place of a prefix parameter; it represents the number of *args* remaining to be processed. When used within a recursive format, in the context of `~?` or `~{`, the `#` prefix parameter represents the number of *format arguments* remaining within the recursive call.

Examples of *format strings*:

`"~S"`	;This is an S directive with no parameters or modifiers.
`"~3,-4:@s"`	;This is an S directive with two parameters, `3` and `-4`,
	; and both the *colon* and *at-sign* flags.
`"~,+4S"`	;Here the first prefix parameter is omitted and takes
	; on its default value, while the second parameter is `4`.

Figure 22–6. Examples of format control strings

format sends the output to *destination*. If *destination* is **nil**, **format** creates and returns a *string* containing the output from *control-string*. If *destination* is *non-nil*, it must be a *string* with a *fill pointer*, a *stream*, or the symbol **t**. If *destination* is a *string* with a *fill pointer*, the output is added to the end of the *string*. If *destination* is a *stream*, the output is sent to that *stream*. If *destination* is **t**, the output is sent to *standard output*.

In the description of the directives that follows, the term *arg* in general refers to the next item of the set of *args* to be processed. The word or phrase at the beginning of each description is a mnemonic for the directive. **format** directives do not bind any of the printer control variables (***print-...***) except as specified in the following descriptions. Implementations may specify the binding of new, implementation-specific printer control variables for each **format** directive, but they may neither bind any standard printer control variables not specified in description of a **format** directive nor fail to bind any standard printer control variables as specified in the description.

22.3.1 FORMAT Basic Output

22.3.1.1 Tilde C: Character

The next *arg* should be a *character*; it is printed according to the modifier flags.

`~C` prints the *character* as if by using **write-char** if it is a *simple character*. *Characters* that are not *simple* are not necessarily printed as if by **write-char**, but are displayed in an *implementation-defined*, abbreviated format. For example,

```
 (format nil "~C" #\A) → "A"
 (format nil "~C" #\Space) → " "
```

`~:C` is the same as `~C` for *printing characters*, but other *characters* are "spelled out." The intent is that this is a "pretty" format for printing characters. For *simple characters* that are not *printing*, what is spelled out is the *name* of the *character* (see **char-name**). For *characters* that are not *simple* and not *printing*, what is spelled out is *implementation-defined*. For example,

```
 (format nil "~:C" #\A) → "A"
 (format nil "~:C" #\Space) → "Space"
;; This next example assumes an implementation-defined "Control" attribute.
 (format nil "~:C" #\Control-Space)
→ "Control-Space"
or→ "c-Space"
```

`~:@C` prints what `~:C` would, and then if the *character* requires unusual shift keys on the keyboard to type it, this fact is mentioned. For example,

```
 (format nil "~:@C" #\Control-Partial) → "Control-∂ (Top-F)"
```

This is the format used for telling the user about a key he is expected to type, in prompts, for instance. The precise output may depend not only on the implementation, but on the particular I/O devices in use.

`~@C` prints the *character* in a way that the *Lisp reader* can understand, using `#\` syntax.

`~@C` binds ***print-escape*** to **t**.

22.3.1.2 Tilde Percent: Newline

This outputs a `#\Newline` character, thereby terminating the current output line and beginning a new one. `~`*n*`%` outputs *n* newlines. No *arg* is used.

22.3.1.3 Tilde Ampersand: Fresh-Line

Unless it can be determined that the output stream is already at the beginning of a line, this outputs a newline. `~`*n*`&` calls **fresh-line** and then outputs $n-1$ newlines. `~0&` does nothing.

22.3.1.4 Tilde Vertical-Bar: Page

This outputs a page separator character, if possible. `~`*n*`|` does this *n* times.

22.3.1.5 Tilde Tilde: Tilde

This outputs a *tilde*. `~`*n*`~` outputs *n* tildes.

22.3.2 FORMAT Radix Control

22.3.2.1 Tilde R: Radix

`~`*n*`R` prints *arg* in radix *n*. The modifier flags and any remaining parameters are used as for the `~D` directive. `~D` is the same as `~10R`. The full form is `~`*radix*`,`*mincol*`,`*padchar*`,`*commachar*`,`*comma-interval*`R`.

If no prefix parameters are given to `~R`, then a different interpretation is given. The argument should be an *integer*. For example, if *arg* is 4:

- `~R` prints *arg* as a cardinal English number: `four`.
- `~:R` prints *arg* as an ordinal English number: `fourth`.
- `~@R` prints *arg* as a Roman numeral: `IV`.

- `~:@R` prints *arg* as an old Roman numeral: `IIII`.

For example:

```
(format nil "~␣' ,4:B" 13) → "1101"
(format nil "~␣' ,4:B" 17) → "1 0001"
(format nil "~19,0,' ,4:B" 3333) → "0000 1101 0000 0101"
(format nil "~3␣,' ,2:R" 17) → "1 22"
(format nil "~␣'|,2:D" #xFFFF) →  "6|55|35"
```

If and only if the first parameter, *n*, is supplied, `~R` binds ***print-escape*** to *false*, ***print-radix*** to *false*, ***print-base*** to *n*, and ***print-readably*** to *false*.

If and only if no parameters are supplied, `~R` binds ***print-base*** to `10`.

22.3.2.2 Tilde D: Decimal

An *arg*, which should be an *integer*, is printed in decimal radix. `~D` will never put a decimal point after the number.

`~`*mincol*`D` uses a column width of *mincol*; spaces are inserted on the left if the number requires fewer than *mincol* columns for its digits and sign. If the number doesn't fit in *mincol* columns, additional columns are used as needed.

`~`*mincol*`,`*padchar*`D` uses *padchar* as the pad character instead of space.

If *arg* is not an *integer*, it is printed in `~A` format and decimal base.

The `@` modifier causes the number's sign to be printed always; the default is to print it only if the number is negative. The `:` modifier causes commas to be printed between groups of digits; *commachar* may be used to change the character used as the comma. *comma-interval* must be an *integer* and defaults to 3. When the `:` modifier is given to any of these directives, the *commachar* is printed between groups of *comma-interval* digits.

Thus the most general form of `~D` is `~`*mincol*`,`*padchar*`,`*commachar*`,`*comma-interval*`D`.

`~D` binds ***print-escape*** to *false*, ***print-radix*** to *false*, ***print-base*** to `10`, and ***print-readably*** to *false*.

22.3.2.3 Tilde B: Binary

This is just like `~D` but prints in binary radix (radix 2) instead of decimal. The full form is therefore `~`*mincol*`,`*padchar*`,`*commachar*`,`*comma-interval*`B`.

`~B` binds ***print-escape*** to *false*, ***print-radix*** to *false*, ***print-base*** to `2`, and ***print-readably*** to *false*.

22.3.2.4 Tilde O: Octal

This is just like `~D` but prints in octal radix (radix 8) instead of decimal. The full form is therefore `~`*mincol*`,`*padchar*`,`*commachar*`,`*comma-interval*`O`.

`~O` binds ***print-escape*** to *false*, ***print-radix*** to *false*, ***print-base*** to `8`, and ***print-readably*** to *false*.

22.3.2.5 Tilde X: Hexadecimal

This is just like `~D` but prints in hexadecimal radix (radix 16) instead of decimal. The full form is therefore `~`*mincol*`,`*padchar*`,`*commachar*`,`*comma-interval*`X`.

`~X` binds ***print-escape*** to *false*, ***print-radix*** to *false*, ***print-base*** to `16`, and ***print-readably*** to *false*.

22.3.3 FORMAT Floating-Point Printers

22.3.3.1 Tilde F: Fixed-Format Floating-Point

The next *arg* is printed as a *float*.

The full form is `~`*w*`,`*d*`,`*k*`,`*overflowchar*`,`*padchar*`F`. The parameter *w* is the width of the field to be printed; *d* is the number of digits to print after the decimal point; *k* is a scale factor that defaults to zero.

Exactly *w* characters will be output. First, leading copies of the character *padchar* (which defaults to a space) are printed, if necessary, to pad the field on the left. If the *arg* is negative, then a minus sign is printed; if the *arg* is not negative, then a plus sign is printed if and only if the `@` modifier was supplied. Then a sequence of digits, containing a single embedded decimal point, is printed; this represents the magnitude of the value of *arg* times 10^k, rounded to *d* fractional digits. When rounding up and rounding down would produce printed values equidistant from the scaled value of *arg*, then the implementation is free to use either one. For example, printing the argument `6.375` using the format `~4,2F` may correctly produce either `6.37` or `6.38`. Leading zeros are not permitted, except that a single zero digit is output before the decimal point if the printed value is less than one, and this single zero digit is not output at all if $w{=}d{+}1$.

If it is impossible to print the value in the required format in a field of width *w*, then one of two actions is taken. If the parameter *overflowchar* is supplied, then *w* copies of that parameter are printed instead of the scaled value of *arg*. If the *overflowchar* parameter is omitted, then the scaled value is printed using more than *w* characters, as many more as may be needed.

If the *w* parameter is omitted, then the field is of variable width. In effect, a value is chosen for *w* in such a way that no leading pad characters need to be printed and exactly *d* characters will follow the decimal point. For example, the directive `~,2F` will print exactly two digits after the decimal point and as many as necessary before the decimal point.

If the parameter d is omitted, then there is no constraint on the number of digits to appear after the decimal point. A value is chosen for d in such a way that as many digits as possible may be printed subject to the width constraint imposed by the parameter w and the constraint that no trailing zero digits may appear in the fraction, except that if the fraction to be printed is zero, then a single zero digit should appear after the decimal point if permitted by the width constraint.

If both w and d are omitted, then the effect is to print the value using ordinary free-format output; **prin1** uses this format for any number whose magnitude is either zero or between 10^{-3} (inclusive) and 10^{7} (exclusive).

If w is omitted, then if the magnitude of *arg* is so large (or, if d is also omitted, so small) that more than 100 digits would have to be printed, then an implementation is free, at its discretion, to print the number using exponential notation instead, as if by the directive `~E` (with all parameters to `~E` defaulted, not taking their values from the `~F` directive).

If *arg* is a *rational* number, then it is coerced to be a *single float* and then printed. Alternatively, an implementation is permitted to process a *rational* number by any other method that has essentially the same behavior but avoids loss of precision or overflow because of the coercion. If w and d are not supplied and the number has no exact decimal representation, for example `1/3`, some precision cutoff must be chosen by the implementation since only a finite number of digits may be printed.

If *arg* is a *complex* number or some non-numeric *object*, then it is printed using the format directive `~`*w*`D`, thereby printing it in decimal radix and a minimum field width of w.

`~F` binds ***print-escape*** to *false* and ***print-readably*** to *false*.

22.3.3.2 Tilde E: Exponential Floating-Point

The next *arg* is printed as a *float* in exponential notation.

The full form is `~`*w*`,`*d*`,`*e*`,`*k*`,`*overflowchar*`,`*padchar*`,`*exponentchar*`E`. The parameter w is the width of the field to be printed; d is the number of digits to print after the decimal point; e is the number of digits to use when printing the exponent; k is a scale factor that defaults to one (not zero).

Exactly w characters will be output. First, leading copies of the character *padchar* (which defaults to a space) are printed, if necessary, to pad the field on the left. If the *arg* is negative, then a minus sign is printed; if the *arg* is not negative, then a plus sign is printed if and only if the `@` modifier was supplied. Then a sequence of digits containing a single embedded decimal point is printed. The form of this sequence of digits depends on the scale factor k. If k is zero, then d digits are printed after the decimal point, and a single zero digit appears before the decimal point if the total field width will permit it. If k is positive, then it must be strictly less than $d+2$; k significant digits are printed before the decimal point, and $d-k+1$ digits are printed after the decimal point. If k is negative, then it must be strictly greater than $-d$; a single zero digit appears before the decimal point if the total field width will permit it, and after the decimal point are printed first $-k$ zeros and then $d+k$ significant digits. The printed fraction must be properly rounded. When rounding up and rounding down would produce printed values equidistant from the scaled value of *arg*, then the implementation is free to use either one. For example, printing the argument `637.5` using the format `~8,2E` may correctly produce either `6.37E+2` or `6.38E+2`.

Following the digit sequence, the exponent is printed. First the character parameter *exponentchar* is printed; if this parameter is omitted, then the *exponent marker* that **prin1** would use is printed, as determined from the type of the *float* and the current value of ***read-default-float-format***. Next, either a plus sign or a minus sign is printed, followed by e digits representing the power of ten by which the printed fraction must be multiplied to properly represent the rounded value of *arg*.

If it is impossible to print the value in the required format in a field of width w, possibly because k is too large or too small or because the exponent cannot be printed in e character positions, then one of two actions is taken. If the parameter *overflowchar* is supplied, then w copies of that parameter are printed instead of the scaled value of *arg*. If the *overflowchar* parameter is omitted, then the scaled value is printed using more than w characters, as many more as may be needed; if the problem is that d is too small for the supplied k or that e is too small, then a larger value is used for d or e as may be needed.

If the w parameter is omitted, then the field is of variable width. In effect a value is chosen for w in such a way that no leading pad characters need to be printed.

If the parameter d is omitted, then there is no constraint on the number of digits to appear. A value is chosen for d in such a way that as many digits as possible may be printed subject to the width constraint imposed by the parameter w, the constraint of the scale factor k, and the constraint that no trailing zero digits may appear in the fraction, except that if the fraction to be printed is zero then a single zero digit should appear after the decimal point.

If the parameter e is omitted, then the exponent is printed using the smallest number of digits necessary to represent its value.

If all of w, d, and e are omitted, then the effect is to print the value using ordinary free-format exponential-notation output; **prin1** uses a similar format for any non-zero number whose magnitude is less than 10^{-3} or greater than or equal to 10^7. The only difference is that the `~E` directive always prints a plus or minus sign in front of the exponent, while **prin1** omits the plus sign if the exponent is non-negative.

If *arg* is a *rational* number, then it is coerced to be a *single float* and then printed. Alternatively, an implementation is permitted to process a *rational* number by any other method that has essentially the same behavior but avoids loss of precision or overflow because of the coercion. If w and d are unsupplied and the number has no exact decimal representation, for example `1/3`, some precision cutoff must be chosen by the implementation since only a finite number of digits may be printed.

If *arg* is a *complex* number or some non-numeric *object*, then it is printed using the format directive `~`*w*`D`, thereby printing it in decimal radix and a minimum field width of w.

`~E` binds ***print-escape*** to *false* and ***print-readably*** to *false*.

22.3.3.3 Tilde G: General Floating-Point

The next *arg* is printed as a *float* in either fixed-format or exponential notation as appropriate.

The full form is `~`*w*`,`*d*`,`*e*`,`*k*`,`*overflowchar*`,`*padchar*`,`*exponentchar*`G`. The format in which to print *arg* depends on the magnitude (absolute value) of the *arg*. Let n be an integer such that $10^{n-1} \leq |arg| < 10^n$. Let *ee* equal e+2, or 4 if e is omitted. Let *ww* equal $w-ee$, or **nil** if w is omitted. If d is omitted, first let q be the number of digits needed to print *arg* with no loss of information and without leading or trailing zeros; then let d equal `(max` *q* `(min` *n* `7))`. Let *dd* equal $d-n$.

If $0 \leq dd \leq d$, then *arg* is printed as if by the format directives

`~`*ww*`,`*dd*`,,`*overflowchar*`,`*padchar*`F~`*ee*`@T`

Note that the scale factor k is not passed to the `~F` directive. For all other values of *dd*, *arg* is printed as if by the format directive

`~`*w*`,`*d*`,`*e*`,`*k*`,`*overflowchar*`,`*padchar*`,`*exponentchar*`E`

In either case, an `@` modifier is supplied to the `~F` or `~E` directive if and only if one was supplied to the `~G` directive.

`~G` binds ***print-escape*** to *false* and ***print-readably*** to *false*.

22.3.3.4 Tilde Dollarsign: Monetary Floating-Point

The next *arg* is printed as a *float* in fixed-format notation.

The full form is `~`*d*`,`*n*`,`*w*`,`*padchar*`$`. The parameter d is the number of digits to print after the decimal point (default value 2); n is the minimum number of digits to print before the decimal point (default value 1); w is the minimum total width of the field to be printed (default value 0).

First padding and the sign are output. If the *arg* is negative, then a minus sign is printed; if the *arg* is not negative, then a plus sign is printed if and only if the `@` modifier was supplied. If the `:` modifier is used, the sign appears before any padding, and otherwise after the padding. If w is supplied and the number of other characters to be output is less than w, then copies of *padchar* (which defaults to a space) are output to make the total field width equal w. Then n digits are printed for the integer part of *arg*, with leading zeros if necessary; then a decimal point; then d digits of fraction, properly rounded.

If the magnitude of *arg* is so large that more than m digits would have to be printed, where m is the larger of w and 100, then an implementation is free, at its discretion, to print the number using exponential notation instead, as if by the directive `~`*w*`,`*q*`,,,`*padchar*`E`, where w and *padchar* are present or omitted according to whether they were present or omitted in the `~$` directive, and where $q=d+n-1$, where d and n are the (possibly default) values given to the `~$` directive.

If *arg* is a *rational* number, then it is coerced to be a *single float* and then printed. Alternatively, an implementation is permitted to process a *rational* number by any other method that has essentially the same behavior but avoids loss of precision or overflow because of the coercion.

If *arg* is a *complex* number or some non-numeric *object*, then it is printed using the format directive `~`*w*`D`, thereby printing it in decimal radix and a minimum field width of w.

~$ binds ***print-escape*** to *false* and ***print-readably*** to *false*.

22.3.4 FORMAT Printer Operations

22.3.4.1 Tilde A: Aesthetic

An *arg*, any *object*, is printed without escape characters (as by **princ**). If *arg* is a *string*, its *characters* will be output verbatim. If *arg* is **nil** it will be printed as **nil**; the *colon* modifier (~:A) will cause an *arg* of **nil** to be printed as (), but if *arg* is a composite structure, such as a *list* or *vector*, any contained occurrences of **nil** will still be printed as **nil**.

~*mincol*A inserts spaces on the right, if necessary, to make the width at least *mincol* columns. The @ modifier causes the spaces to be inserted on the left rather than the right.

~*mincol*,*colinc*,*minpad*,*padchar*A is the full form of ~A, which allows control of the padding. The *string* is padded on the right (or on the left if the @ modifier is used) with at least *minpad* copies of *padchar*; padding characters are then inserted *colinc* characters at a time until the total width is at least *mincol*. The defaults are 0 for *mincol* and *minpad*, 1 for *colinc*, and the space character for *padchar*.

~A binds ***print-escape*** to *false*, and ***print-readably*** to *false*.

22.3.4.2 Tilde S: Standard

This is just like ~A, but *arg* is printed with escape characters (as by **prin1** rather than princ). The output is therefore suitable for input to **read**. ~S accepts all the arguments and modifiers that ~A does.

~S binds ***print-escape*** to **t**.

22.3.4.3 Tilde W: Write

An argument, any *object*, is printed obeying every printer control variable (as by **write**). In addition, ~W interacts correctly with depth abbreviation, by not resetting the depth counter to zero. ~W does not accept parameters. If given the *colon* modifier, ~W binds ***print-pretty*** to *true*. If given the *at-sign* modifier, ~W binds ***print-level*** and ***print-length*** to **nil**.

~W provides automatic support for the detection of circularity and sharing. If the *value* of ***print-circle*** is not **nil** and ~W is applied to an argument that is a circular (or shared) reference, an appropriate #*n*# marker is inserted in the output instead of printing the argument.

22.3.5 FORMAT Pretty Printer Operations

The following constructs provide access to the *pretty printer*:

22.3.5.1 Tilde Underscore: Conditional Newline

Without any modifiers, `~_` is the same as `(pprint-newline :linear)`. `~@_` is the same as `(pprint-newline :miser)`. `~:_` is the same as `(pprint-newline :fill)`. `~:@_` is the same as `(pprint-newline :mandatory)`.

22.3.5.2 Tilde Less-Than-Sign: Logical Block

`~<...~:>`

If `~:>` is used to terminate a `~<...~>`, the directive is equivalent to a call to **pprint-logical-block**. The argument corresponding to the `~<...~:>` directive is treated in the same way as the *list* argument to **pprint-logical-block**, thereby providing automatic support for non-*list* arguments and the detection of circularity, sharing, and depth abbreviation. The portion of the *control-string* nested within the `~<...~:>` specifies the `:prefix` (or `:per-line-prefix`), `:suffix`, and body of the **pprint-logical-block**.

The *control-string* portion enclosed by `~<...~:>` can be divided into segments `~<`*prefix*`~;`*body*`~;`*suffix*`~:>` by `~;` directives. If the first section is terminated by `~@;`, it specifies a per-line prefix rather than a simple prefix. The *prefix* and *suffix* cannot contain format directives. An error is signaled if either the prefix or suffix fails to be a constant string or if the enclosed portion is divided into more than three segments.

If the enclosed portion is divided into only two segments, the *suffix* defaults to the null string. If the enclosed portion consists of only a single segment, both the *prefix* and the *suffix* default to the null string. If the *colon* modifier is used (*i.e.*, `~:<...~:>`), the *prefix* and *suffix* default to `"("` and `")"` (respectively) instead of the null string.

The body segment can be any arbitrary *format string*. This *format string* is applied to the elements of the list corresponding to the `~<...~:>` directive as a whole. Elements are extracted from this list using **pprint-pop**, thereby providing automatic support for malformed lists, and the detection of circularity, sharing, and length abbreviation. Within the body segment, `~^` acts like **pprint-exit-if-list-exhausted**.

`~<...~:>` supports a feature not supported by **pprint-logical-block**. If `~:@>` is used to terminate the directive (*i.e.*, `~<...~:@>`), then a fill-style conditional newline is automatically inserted after each group of blanks immediately contained in the body (except for blanks after a ⟨*Newline*⟩ directive). This makes it easy to achieve the equivalent of paragraph filling.

If the *at-sign* modifier is used with `~<...~:>`, the entire remaining argument list is passed to the directive as its argument. All of the remaining arguments are always consumed by `~@<...~:>`, even if they are not all used by the *format string* nested in the directive. Other than the difference in its argument, `~@<...~:>` is exactly the same as `~<...~:>` except that circularity detection is not applied if `~@<...~:>` is encountered at top level in a *format string*. This ensures that circularity detection is applied only to data lists, not to *format argument lists*.

`" . #`*n*`#"` is printed if circularity or sharing has to be indicated for its argument as a whole.

To a considerable extent, the basic form of the directive `~<...~>` is incompatible with the dynamic control of the arrangement of output by `~W`, `~_`, `~<...~:>`, `~I`, and `~:T`. As a result, an error is signaled if any of these directives is nested within `~<...~>`. Beyond this, an error is also signaled if the `~<...~:;...~>` form of `~<...~>` is used in the same *format string* with `~W`, `~_`, `~<...~:>`, `~I`, or `~:T`.

See also Section 22.3.6.2 (Tilde Less-Than-Sign: Justification).

22.3.5.3 Tilde I: Indent

`~nI` is the same as `(pprint-indent :block n)`.

`~n:I` is the same as `(pprint-indent :current n)`. In both cases, *n* defaults to zero, if it is omitted.

22.3.5.4 Tilde Slash: Call Function

~/name/

User defined functions can be called from within a format string by using the directive *~/name/*. The *colon* modifier, the *at-sign* modifier, and arbitrarily many parameters can be specified with the *~/name/* directive. *name* can be any arbitrary string that does not contain a "/". All of the characters in *name* are treated as if they were upper case. If *name* contains a single *colon* (`:`) or double *colon* (`::`), then everything up to but not including the first `":"` or `"::"` is taken to be a *string* that names a *package*. Everything after the first `":"` or `"::"` (if any) is taken to be a *string* that names a `symbol`. The function corresponding to a `~/name/` directive is obtained by looking up the *symbol* that has the indicated name in the indicated *package*. If *name* does not contain a `":"` or `"::"`, then the whole *name* string is looked up in the `COMMON-LISP-USER` *package*.

When a `~/name/` directive is encountered, the indicated function is called with four or more arguments. The first four arguments are: the output stream, the *format argument* corresponding to the directive, a *generalized boolean* that is *true* if the *colon* modifier was used, and a *generalized boolean* that is *true* if the *at-sign* modifier was used. The remaining arguments consist of any parameters specified with the directive. The function should print the argument appropriately. Any values returned by the function are ignored.

The three *functions* **pprint-linear**, **pprint-fill**, and **pprint-tabular** are specifically designed so that they can be called by `~/.../` (*i.e.*, `~/pprint-linear/`, `~/pprint-fill/`, and `~/pprint-tabular/`). In particular they take *colon* and *at-sign* arguments.

22.3.6 FORMAT Layout Control

22.3.6.1 Tilde T: Tabulate

This spaces over to a given column. `~`*colnum*`,`*colinc*`T` will output sufficient spaces to move the cursor to column *colnum*. If the cursor is already at or beyond column *colnum*, it will output spaces to move it to column *colnum*+*k***colinc* for the smallest positive integer *k* possible, unless *colinc* is zero, in which case no spaces are output if the cursor is already at or beyond column *colnum*. *colnum* and *colinc* default to `1`.

If for some reason the current absolute column position cannot be determined by direct inquiry, **format** may be able to deduce the current column position by noting that certain directives (such as `~%`, or `~&`, or `~A` with the argument being a string containing a newline) cause the column position to be reset to zero, and counting the number of characters emitted since that point. If that fails, **format** may attempt a similar deduction on the riskier assumption that the destination was at column zero when **format** was invoked. If even this heuristic fails or is implementationally inconvenient, at worst the `~T` operation will simply output two spaces.

`~@T` performs relative tabulation. `~`*colrel*`,`*colinc*`@T` outputs *colrel* spaces and then outputs the smallest non-negative number of additional spaces necessary to move the cursor to a column that is a multiple of *colinc*. For example, the directive `~3,8@T` outputs three spaces and then moves the cursor to a "standard multiple-of-eight tab stop" if not at one already. If the current output column cannot be determined, however, then *colinc* is ignored, and exactly *colrel* spaces are output.

If the *colon* modifier is used with the `~T` directive, the tabbing computation is done relative to the horizontal position where the section immediately containing the directive begins, rather than with respect to a horizontal position of zero. The numerical parameters are both interpreted as being in units of *ems* and both default to `1`. `~`*n*`,`*m*`:T` is the same as `(pprint-tab :section` *n m*`)`. `~`*n*`,`*m*`:@T` is the same as `(pprint-tab :section-relative` *n m*`)`.

22.3.6.2 Tilde Less-Than-Sign: Justification

`~`*mincol*`,`*colinc*`,`*minpad*`,`*padchar*`<`*str*`~>`

This justifies the text produced by processing *str* within a field at least *mincol* columns wide. *str* may be divided up into segments with `~;`, in which case the spacing is evenly divided between the text segments.

With no modifiers, the leftmost text segment is left justified in the field, and the rightmost text segment is right justified. If there is only one text element, as a special case, it is right justified. The : modifier causes spacing to be introduced before the first text segment; the `@` modifier causes spacing to be added after the last. The *minpad* parameter (default `0`) is the minimum number of padding characters to be output between each segment. The padding character is supplied by *padchar*, which defaults to the space character. If the total width needed to satisfy these constraints is greater than *mincol*, then the width used is *mincol*+*k***colinc* for the smallest possible non-negative integer value *k*. *colinc* defaults to `1`, and *mincol* defaults to `0`.

Note that *str* may include **format** directives. All the clauses in *str* are processed in order; it is the resulting pieces of text that are justified.

The `~^` directive may be used to terminate processing of the clauses prematurely, in which case only the completely processed clauses are justified.

If the first clause of a `~<` is terminated with `~:;` instead of `~;`, then it is used in a special way. All of the clauses are processed (subject to `~^`, of course), but the first one is not used in performing the spacing and padding. When the padded result has been determined, then if it will fit on the current line of output, it is output, and the text for the first clause is discarded. If, however, the padded text will not fit on the current line, then the text segment for the first clause is output before the padded text. The first clause ought to contain a newline (such as a `~%` directive). The first clause is always processed, and so any arguments it refers to will be used; the decision is whether to use the resulting segment of text, not whether to process the first clause. If the `~:;` has a prefix parameter *n*, then the padded text must fit on the current line with *n* character positions to spare to avoid outputting the first clause's text. For example, the control string

```
"~%;; ~{~<~%;; ~1:; ~S~>~^,~}.~%"
```

can be used to print a list of items separated by commas without breaking items over line boundaries, beginning each line with `;;` . The prefix parameter `1` in `~1:;` accounts for the width of the comma that will follow the justified item if it is not the last element in the list, or the period if it is. If `~:;` has a second prefix parameter, then it is used as the width of the line, thus overriding the natural line width of the output stream. To make the preceding example use a line width of 50, one would write

```
"~%;; ~{~<~%;; ~1,50:; ~S~>~^,~} .~%"
```

If the second argument is not supplied, then **format** uses the line width of the *destination* output stream. If this cannot be determined (for example, when producing a *string* result), then **format** uses `72` as the line length.

See also Section 22.3.5.2 (Tilde Less-Than-Sign: Logical Block).

22.3.6.3 Tilde Greater-Than-Sign: End of Justification

`~>` terminates a `~<`. The consequences of using it elsewhere are undefined.

22.3.7 FORMAT Control-Flow Operations

22.3.7.1 Tilde Asterisk: Go-To

The next *arg* is ignored. `~`*n*`*` ignores the next *n* arguments.

`~:*` backs up in the list of arguments so that the argument last processed will be processed again. `~`*n*`:*` backs up *n* arguments.

When within a `~{` construct (see below), the ignoring (in either direction) is relative to the list of arguments being processed by the iteration.

`~n@*` goes to the *n*th *arg*, where 0 means the first one; *n* defaults to 0, so `~@*` goes back to the first *arg*. Directives after a `~n@*` will take arguments in sequence beginning with the one gone to. When within a `~{` construct, the "goto" is relative to the list of arguments being processed by the iteration.

22.3.7.2 Tilde Left-Bracket: Conditional Expression

`~[`*str0*`~;`*str1*`~;`*...*`~;`*strn*`~]`

This is a set of control strings, called *clauses*, one of which is chosen and used. The clauses are separated by `~;` and the construct is terminated by `~]`. For example,

```
"~[Siamese~;Manx~;Persian~] Cat"
```

The *arg*th clause is selected, where the first clause is number 0. If a prefix parameter is given (as `~n[`), then the parameter is used instead of an argument. If *arg* is out of range then no clause is selected and no error is signaled. After the selected alternative has been processed, the control string continues after the `~]`.

`~[`*str0*`~;`*str1*`~;`*...*`~;`*strn*`~:;`*default*`~]` has a default case. If the *last* `~;` used to separate clauses is `~:;` instead, then the last clause is an else clause that is performed if no other clause is selected. For example:

```
"~[Siamese~;Manx~;Persian~:;Alley~] Cat"
```

`~:[`*alternative*`~;`*consequent*`~]` selects the *alternative* control string if *arg* is *false*, and selects the *consequent* control string otherwise.

`~@[`*consequent*`~]` tests the argument. If it is *true*, then the argument is not used up by the `~[` command but remains as the next one to be processed, and the one clause *consequent* is processed. If the *arg* is *false*, then the argument is used up, and the clause is not processed. The clause therefore should normally use exactly one argument, and may expect it to be *non-nil*. For example:

```
 (setq *print-level* nil *print-length* 5)
 (format nil
        "~@[ print level = ~D~]~@[ print length = ~D~]"
        *print-level* *print-length*)
→  " print length = 5"
```

Note also that

```
 (format stream "...~@[str~]..." ...)
≡ (format stream "...~:[~;~:*str~]..." ...)
```

The combination of `~[` and `#` is useful, for example, for dealing with English conventions for printing lists:

```
 (setq foo "Items:~#[ none~; ~S~; ~S and ~S~
          ~:;~@{~#[~; and~] ~S~^,~}~].")
```

```
(format nil foo) →  "Items: none."
(format nil foo 'foo) →  "Items: FOO."
(format nil foo 'foo 'bar) →  "Items: FOO and BAR."
(format nil foo 'foo 'bar 'baz) →  "Items: FOO, BAR, and BAZ."
(format nil foo 'foo 'bar 'baz 'quux) →  "Items: FOO, BAR, BAZ, and QUUX."
```

22.3.7.3 Tilde Right-Bracket: End of Conditional Expression

`~]` terminates a `~[`. The consequences of using it elsewhere are undefined.

22.3.7.4 Tilde Left-Brace: Iteration

~{*str*~}

This is an iteration construct. The argument should be a *list*, which is used as a set of arguments as if for a recursive call to **format**. The *string str* is used repeatedly as the control string. Each iteration can absorb as many elements of the *list* as it likes as arguments; if *str* uses up two arguments by itself, then two elements of the *list* will get used up each time around the loop. If before any iteration step the *list* is empty, then the iteration is terminated. Also, if a prefix parameter *n* is given, then there will be at most *n* repetitions of processing of *str*. Finally, the `~^` directive can be used to terminate the iteration prematurely.

For example:

```
 (format nil "The winners are:~{ ~S~}."
         '(fred harry jill))
→ "The winners are: FRED HARRY JILL."
 (format nil "Pairs:~{ <~S,~S>~}."
         '(a 1 b 2 c 3))
→ "Pairs: <A,1> <B,2> <C,3>."
```

~:{*str*~} is similar, but the argument should be a *list* of sublists. At each repetition step, one sublist is used as the set of arguments for processing *str*; on the next repetition, a new sublist is used, whether or not all of the last sublist had been processed. For example:

```
 (format nil "Pairs:~:{ <~S,~S>~}."
                   '((a 1) (b 2) (c 3)))
→ "Pairs: <A,1> <B,2> <C,3>."
```

~@{*str*~} is similar to ~{*str*~}, but instead of using one argument that is a list, all the remaining arguments are used as the list of arguments for the iteration. Example:

```
 (format nil "Pairs:~@{ <~S,~S>~}." 'a 1 'b 2 'c 3)
→ "Pairs: <A,1> <B,2> <C,3>."
```

If the iteration is terminated before all the remaining arguments are consumed, then any arguments not processed by the iteration remain to be processed by any directives following the iteration construct.

`~:@{`*str*`~}` combines the features of `~:{`*str*`~}` and `~@{`*str*`~}`. All the remaining arguments are used, and each one must be a *list*. On each iteration, the next argument is used as a *list* of arguments to *str*. Example:

```
 (format nil "Pairs:~:@{ <~S,~S>~}."
              '(a 1) '(b 2) '(c 3))
→ "Pairs: <A,1> <B,2> <C,3>."
```

Terminating the repetition construct with `~:}` instead of `~}` forces *str* to be processed at least once, even if the initial list of arguments is null. However, this will not override an explicit prefix parameter of zero.

If *str* is empty, then an argument is used as *str*. It must be a *format control* and precede any arguments processed by the iteration. As an example, the following are equivalent:

```
    (apply #'format stream string arguments)
 ≡ (format stream "~1{~:}" string arguments)
```

This will use `string` as a formatting string. The `~1{` says it will be processed at most once, and the `~:}` says it will be processed at least once. Therefore it is processed exactly once, using `arguments` as the arguments. This case may be handled more clearly by the `~?` directive, but this general feature of `~{` is more powerful than `~?`.

22.3.7.5 Tilde Right-Brace: End of Iteration

`~}` terminates a `~{`. The consequences of using it elsewhere are undefined.

22.3.7.6 Tilde Question-Mark: Recursive Processing

The next *arg* must be a *format control*, and the one after it a *list*; both are consumed by the `~?` directive. The two are processed as a *control-string*, with the elements of the *list* as the arguments. Once the recursive processing has been finished, the processing of the control string containing the `~?` directive is resumed. Example:

```
 (format nil "~? ~D" "<~A ~D>" '("Foo" 5) 7) → "<Foo 5> 7"
 (format nil "~? ~D" "<~A ~D>" '("Foo" 5 14) 7) → "<Foo 5> 7"
```

Note that in the second example three arguments are supplied to the *format string* `"<~A ~D>"`, but only two are processed and the third is therefore ignored.

With the `@` modifier, only one *arg* is directly consumed. The *arg* must be a *string*; it is processed as part of the control string as if it had appeared in place of the `~@?` construct, and any directives in the recursively processed control string may consume arguments of the control string containing the `~@?` directive. Example:

```
 (format nil "~@? ~D" "<~A ~D>" "Foo" 5 7) → "<Foo 5> 7"
 (format nil "~@? ~D" "<~A ~D>" "Foo" 5 14 7) → "<Foo 5> 14"
```

22.3.8 FORMAT Miscellaneous Operations

22.3.8.1 Tilde Left-Paren: Case Conversion

`~(`*str*`~)`

The contained control string *str* is processed, and what it produces is subject to case conversion.

With no flags, every *uppercase character* is converted to the corresponding *lowercase character*.

`~:(` capitalizes all words, as if by **string-capitalize**.

`~@(` capitalizes just the first word and forces the rest to lower case.

`~:@(` converts every lowercase character to the corresponding uppercase character.

In this example `~@(` is used to cause the first word produced by `~@R` to be capitalized:

```
(format nil "~@R ~(~@R~)" 14 14)
→ "XIV xiv"
(defun f (n) (format nil "~@(~R~) error~:P detected." n)) → F
(f 0) → "Zero errors detected."
(f 1) → "One error detected."
(f 23) → "Twenty-three errors detected."
```

When case conversions appear nested, the outer conversion dominates, as illustrated in the following example:

```
(format nil "~@(how is ~:(BOB SMITH~)?~)")
→ "How is bob smith?"
not→ "How is Bob Smith?"
```

22.3.8.2 Tilde Right-Paren: End of Case Conversion

`~)` terminates a `~(`. The consequences of using it elsewhere are undefined.

22.3.8.3 Tilde P: Plural

If *arg* is not **eql** to the integer `1`, a lowercase `s` is printed; if *arg* is **eql** to `1`, nothing is printed. If *arg* is a floating-point `1.0`, the `s` is printed.

`~:P` does the same thing, after doing a `~:*` to back up one argument; that is, it prints a lowercase `s` if the previous argument was not `1`.

`~@P` prints `y` if the argument is `1`, or `ies` if it is not. `~:@P` does the same thing, but backs up first.

```
(format nil "~D tr~:@P/~D win~:P" 7 1) → "7 tries/1 win"
```

```
(format nil "~D tr~:@P/~D win~:P" 1 0) → "1 try/0 wins"
(format nil "~D tr~:@P/~D win~:P" 1 3) → "1 try/3 wins"
```

22.3.9 FORMAT Miscellaneous Pseudo-Operations

22.3.9.1 Tilde Semicolon: Clause Separator

This separates clauses in `~[` and `~<` constructs. The consequences of using it elsewhere are undefined.

22.3.9.2 Tilde Circumflex: Escape Upward

`~^`

This is an escape construct. If there are no more arguments remaining to be processed, then the immediately enclosing `~{` or `~<` construct is terminated. If there is no such enclosing construct, then the entire formatting operation is terminated. In the `~<` case, the formatting is performed, but no more segments are processed before doing the justification. `~^` may appear anywhere in a `~{` construct.

```
 (setq donestr "Done.~^ ~D warning~:P.~^ ~D error~:P.")
→ "Done.~^ ~D warning~:P.~^ ~D error~:P."
 (format nil donestr) → "Done."
 (format nil donestr 3) → "Done. 3 warnings."
 (format nil donestr 1 5) → "Done. 1 warning. 5 errors."
```

If a prefix parameter is given, then termination occurs if the parameter is zero. (Hence `~^` is equivalent to `~#^`.) If two parameters are given, termination occurs if they are equal. If three parameters are given, termination occurs if the first is less than or equal to the second and the second is less than or equal to the third. Of course, this is useless if all the prefix parameters are constants; at least one of them should be a `#` or a `V` parameter.

If `~^` is used within a `~:{` construct, then it terminates the current iteration step because in the standard case it tests for remaining arguments of the current step only; the next iteration step commences immediately. `~:^` is used to terminate the iteration process. `~:^` may be used only if the command it would terminate is `~:{` or `~:@{`. The entire iteration process is terminated if and only if the sublist that is supplying the arguments for the current iteration step is the last sublist in the case of `~:{`, or the last **format** argument in the case of `~:@{`. `~:^` is not equivalent to `~#:^`; the latter terminates the entire iteration if and only if no arguments remain for the current iteration step. For example:

```
(format nil "~:{~@?~:^...~}" '(("a") ("b"))) → "a...b"
```

If `~^` appears within a control string being processed under the control of a `~?` directive, but not within any `~{` or `~<` construct within that string, then the string being processed will be terminated, thereby ending processing of the `~?` directive. Processing then continues within the string containing the `~?` directive at the point following that directive.

If `~^` appears within a `~[` or `~(` construct, then all the commands up to the `~^` are properly selected or case-converted, the `~[` or `~(` processing is terminated, and the outward search continues for a `~{` or `~<` construct to be terminated. For example:

```
 (setq tellstr "~@(~@[~R~]~^ ~A!~)")
→ "~@(~@[~R~]~^ ~A!~)"
 (format nil tellstr 23) → "Twenty-three!"
 (format nil tellstr nil "losers") → " Losers!"
 (format nil tellstr 23 "losers") → "Twenty-three losers!"
```

Following are examples of the use of `~^` within a `~<` construct.

```
 (format nil "~15<~S~;~^~S~;~^~S~>" 'foo)
→  "            FOO"
 (format nil "~15<~S~;~^~S~;~^~S~>" 'foo 'bar)
→  "FOO         BAR"
 (format nil "~15<~S~;~^~S~;~^~S~>" 'foo 'bar 'baz)
→  "FOO   BAR   BAZ"
```

22.3.9.3 Tilde Newline: Ignored Newline

Tilde immediately followed by a *newline* ignores the *newline* and any following non-newline *whitespace*$_1$ characters. With a `:`, the *newline* is ignored, but any following *whitespace*$_1$ is left in place. With an `@`, the *newline* is left in place, but any following *whitespace*$_1$ is ignored. For example:

```
 (defun type-clash-error (fn nargs argnum right-type wrong-type)
   (format *error-output*
           "~&~S requires its ~:[~:R~;~*~]~
           argument to be of type ~S,~%but it was called ~
           with an argument of type ~S.~%"
           fn (eql nargs 1) argnum right-type wrong-type))
 (type-clash-error 'aref nil 2 'integer 'vector)  prints:
AREF requires its second argument to be of type INTEGER,
but it was called with an argument of type VECTOR.
NIL
 (type-clash-error 'car 1 1 'list 'short-float)  prints:
CAR requires its argument to be of type LIST,
but it was called with an argument of type SHORT-FLOAT.
NIL
```

Note that in this example newlines appear in the output only as specified by the `~&` and `~%` directives; the actual newline characters in the control string are suppressed because each is preceded by a tilde.

22.3.10 Additional Information about FORMAT Operations

22.3.10.1 Nesting of FORMAT Operations

The case-conversion, conditional, iteration, and justification constructs can contain other formatting constructs by bracketing them. These constructs must nest properly with respect to each other. For example, it is not legitimate to put the start of a case-conversion construct in each arm of a conditional and the end of the case-conversion construct outside the conditional:

```
 (format nil "~:[abc~:@(def~;ghi~
:@(jkl~]mno~)" x) ;Invalid!
```

This notation is invalid because the `~[...~;...~]` and `~(...~)` constructs are not properly nested.

The processing indirection caused by the `~?` directive is also a kind of nesting for the purposes of this rule of proper nesting. It is not permitted to start a bracketing construct within a string processed under control of a `~?` directive and end the construct at some point after the `~?` construct in the string containing that construct, or vice versa. For example, this situation is invalid:

```
 (format nil "~@?ghi~)" "abc~@(def") ;Invalid!
```

This notation is invalid because the `~?` and `~(...~)` constructs are not properly nested.

22.3.10.2 Missing and Additional FORMAT Arguments

The consequences are undefined if no *arg* remains for a directive requiring an argument. However, it is permissible for one or more *args* to remain unprocessed by a directive; such *args* are ignored.

22.3.10.3 Additional FORMAT Parameters

The consequences are undefined if a format directive is given more parameters than it is described here as accepting.

22.3.10.4 Undefined FORMAT Modifier Combinations

The consequences are undefined if *colon* or *at-sign* modifiers are given to a directive in a combination not specifically described here as being meaningful.

22.3.11 Examples of FORMAT

```
 (format nil "foo") → "foo"
 (setq x 5) → 5
 (format nil "The answer is ~D." x) → "The answer is 5."
 (format nil "The answer is ~3D." x) → "The answer is   5."
 (format nil "The answer is ~3,'0D." x) → "The answer is 005."
 (format nil "The answer is ~:D." (expt 47 x))
→ "The answer is 229,345,007."
 (setq y "elephant") → "elephant"
 (format nil "Look at the ~A!" y) → "Look at the elephant!"
 (setq n 3) → 3
 (format nil "~D item~:P found." n) → "3 items found."
 (format nil "~R dog~:[s are~; is~] here." n (= n 1))
→ "three dogs are here."
 (format nil "~R dog~:*~[s are~; is~:;s are~] here." n)
→ "three dogs are here."
 (format nil "Here ~[are~;is~:;are~] ~:*~R pupp~:@P." n)
→ "Here are three puppies."

 (defun foo (x)
   (format nil "~6,2F|~6,2,1,'*F|~6,2,,'?F|~6F|~,2F|~F"
           x x x x x x)) → FOO
 (foo 3.14159)  → "  3.14| 31.42|  3.14|3.1416|3.14|3.14159"
 (foo -3.14159) → " -3.14|-31.42| -3.14|-3.142|-3.14|-3.14159"
 (foo 100.0)    → "100.00|******|100.00| 100.0|100.00|100.0"
 (foo 1234.0)   → "1234.00|******|??????|1234.0|1234.00|1234.0"
 (foo 0.006)    → "  0.01|  0.06|  0.01| 0.006|0.01|0.006"

 (defun foo (x)
    (format nil
           "~9,2,1,,'*E|~10,3,2,2,'?,,'$E|~
            ~9,3,2,-2,'%@E|~9,2E"
           x x x x))
 (foo 3.14159)  → "  3.14E+0| 31.42$-01|+.003E+03|  3.14E+0"
 (foo -3.14159) → " -3.14E+0|-31.42$-01|-.003E+03| -3.14E+0"
 (foo 1100.0)   → "  1.10E+3| 11.00$+02|+.001E+06|  1.10E+3"
 (foo 1100.0L0) → "  1.10L+3| 11.00$+02|+.001L+06|  1.10L+3"
 (foo 1.1E13)   → "*********| 11.00$+12|+.001E+16| 1.10E+13"
 (foo 1.1L120)  → "*********|??????????|%%%%%%%%%|1.10L+120"
 (foo 1.1L1200) → "*********|??????????|%%%%%%%%%|1.10L+1200"
```

As an example of the effects of varying the scale factor, the code

```
 (dotimes (k 13)
   (format t "~%Scale factor ~2D: |~13,6,2,VE|"
```

```
          (- k 5) (- k 5) 3.14159))
```

produces the following output:

```
Scale factor -5: | 0.000003E+06|
Scale factor -4: | 0.000031E+05|
Scale factor -3: | 0.000314E+04|
Scale factor -2: | 0.003142E+03|
Scale factor -1: | 0.031416E+02|
Scale factor  0: | 0.314159E+01|
Scale factor  1: | 3.141590E+00|
Scale factor  2: | 31.41590E-01|
Scale factor  3: | 314.1590E-02|
Scale factor  4: | 3141.590E-03|
Scale factor  5: | 31415.90E-04|
Scale factor  6: | 314159.0E-05|
Scale factor  7: | 3141590.E-06|
```

```
(defun foo (x)
  (format nil "~9,2,1,,'*G|~9,3,2,3,'?,,'$G|~9,3,2,0,'%G|~9,2G"
         x x x x))
(foo 0.0314159) → "  3.14E-2|314.2$-04|0.314E-01|  3.14E-2"
(foo 0.314159)  → "  0.31   |0.314    |0.314    | 0.31    "
(foo 3.14159)   → "   3.1   | 3.14    | 3.14    |  3.1    "
(foo 31.4159)   → "   31.   | 31.4    | 31.4    |  31.    "
(foo 314.159)   → "  3.14E+2| 314.    | 314.    |  3.14E+2"
(foo 3141.59)   → "  3.14E+3|314.2$+01|0.314E+04|  3.14E+3"
(foo 3141.59L0) → "  3.14L+3|314.2$+01|0.314L+04|  3.14L+3"
(foo 3.14E12)   → "*********|314.0$+10|0.314E+13| 3.14E+12"
(foo 3.14L120)  → "*********|?????????|%%%%%%%%%|3.14L+120"
(foo 3.14L1200) → "*********|?????????|%%%%%%%%%|3.14L+1200"
```

```
(format nil "~10<foo~;bar~>")   → "foo    bar"
(format nil "~10:<foo~;bar~>")  → "  foo  bar"
(format nil "~10<foobar~>")     → "    foobar"
(format nil "~10:<foobar~>")    → "    foobar"
(format nil "~10:@<foo~;bar~>") → "  foo bar "
(format nil "~10@<foobar~>")    → "foobar    "
(format nil "~10:@<foobar~>")   → "  foobar  "
```

```
 (FORMAT NIL "Written to ~A." #P"foo.bin")
 → "Written to foo.bin."
```

22.3.12 Notes about FORMAT

Formatted output is performed not only by **format**, but by certain other functions that accept a *format control* the way **format** does. For example, error-signaling functions such as **cerror** accept *format controls*.

Note that the meaning of **nil** and **t** as destinations to **format** are different than those of **nil** and **t** as *stream designators*.

The `~^` should appear only at the beginning of a `~<` clause, because it aborts the entire clause in which it appears (as well as all following clauses).

copy-pprint-dispatch *Function*

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.

Exceptional Situations:

Should signal an error of *type* **type-error** if *table* is not a *pprint dispatch table*.

formatter *Macro*

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
```

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

pprint-dispatch *Function*

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*.

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**.

Notes:

```
(let ((*print-pretty* t))
  (write object :stream s))
≡ (funcall (pprint-dispatch object) s object)
```

pprint-exit-if-list-exhausted *Local Macro*

Syntax:

pprint-exit-if-list-exhausted ⟨*no arguments*⟩ → **nil**

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.

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**.

pprint-fill, pprint-linear, pprint-tabular *Function*

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 tabsize* → **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*.

tabsize—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 *tabsize* 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.

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 `~:T` format directive nested within an iteration over a list.

pprint-indent *Function*

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, 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.

Exceptional Situations:

An error is signaled if *relative-to* is any *object* other than `:block` or `:current`.

See Also:

Section 22.3.5.3 (Tilde I: Indent)

pprint-logical-block *Macro*

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*.

pprint-logical-block

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 `: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 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***.

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*.)

pprint-newline *Function*

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:

pprint-newline

`: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:

Section 22.3.5.1 (Tilde Underscore: Conditional Newline), Section 22.2.2 (Examples of using the Pretty Printer)

pprint-pop *Local Macro*

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.

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**.

pprint-tab *Function*

Syntax:

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).

Exceptional Situations:

An error is signaled if *kind* is not one of `:line`, `:section`, `:line-relative`, or `:section-relative`.

See Also:

pprint-logical-block

print-object *Standard Generic Function*

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*.

print-object

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*.

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)

print-unreadable-object *Macro*

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>"
```

Exceptional Situations:

If ***print-readably*** is *true*, **print-unreadable-object** signals an error of *type* **print-not-readable** without printing anything.

set-pprint-dispatch *Function*

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 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.

Exceptional Situations:

An error is signaled if *priority* is not a *real*.

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**.

write, prin1, print, pprint, princ *Function*

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*

write, prin1, print, pprint, princ

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 Figure 22–7, the corresponding *printer control variable* is dynamically bound to its *value* while printing goes on; for each *keyword parameter* in Figure 22–7 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*.

write, prin1, print, pprint, princ

Parameter	Corresponding Dynamic Variable
array	***print-array***
base	***print-base***
case	***print-case***
circle	***print-circle***
escape	***print-escape***
gensym	***print-gensym***
length	***print-length***
level	***print-level***
lines	***print-lines***
miser-width	***print-miser-width***
pprint-dispatch	***print-pprint-dispatch***
pretty	***print-pretty***
radix	***print-radix***
readably	***print-readably***
right-margin	***print-right-margin***

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***.

See Also:

readtable-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)
```

write-to-string, prin1-to-string, princ-to-string

Function

Syntax:

write-to-string *object* `&key` *array base case circle escape gensym*
length level lines miser-width pprint-dispatch
pretty radix readably right-margin

$\rightarrow$ *string*

prin1-to-string *object* $\rightarrow$ *string*

princ-to-string *object* $\rightarrow$ *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"
```

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***.

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))
```

∗print-array∗ *Variable*

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.

Affected By:

The *implementation*.

See Also:

Section 2.4.8.3 (Sharpsign Left-Parenthesis), Section 2.4.8.20 (Sharpsign Less-Than-Sign)

∗print-base∗, ∗print-radix∗ *Variable*

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
 (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)))       ;3, 8, 10, 16
▷ #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**

∗print-case∗ *Variable*

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:

```
 (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-THAT |And-something-elSE|
▷ this-and-that a\n\d-\s\o\m\e\t\h\i\n\g-\e\lse
▷ This-And-That A\n\d-\s\o\m\e\t\h\i\n\g-\e\lse
→ NIL
 (test-print-case) ;Implementation B
▷ THIS-AND-THAT |And-something-elSE|
▷ this-and-that a|nd-something-el|se
▷ This-And-That A|nd-something-el|se
→ NIL
```

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.

If ***print-escape*** is *true*, lowercase characters in the *name* of a *symbol* are always printed in lowercase, and are preceded by a single escape character or enclosed by multiple escape characters; uppercase characters in the *name* of a *symbol* are printed in upper case, in lower case, or in mixed case so as to capitalize words, according to the value of ***print-case***. The convention for what constitutes a "word" is the same as for **string-capitalize**.

print-circle *Variable*

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
```

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.

∗print-escape∗ *Variable*

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, **readtable-case**

Notes:

princ effectively binds ***print-escape*** to *false*. **prin1** effectively binds ***print-escape*** to *true*.

print-gensym *Variable*

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
```

See Also:

write, ***print-escape***

∗print-level∗, ∗print-length∗

∗print-level∗, ∗print-length∗ *Variable*

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 an 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
```

See Also:

write

∗print-lines∗ *Variable*

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⟩
```

Notes:

The "`..`" notation is intentionally different than the "`...`" notation used for level abbreviation, so that the two different situations can be visually distinguished.

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.

∗print-miser-width∗ *Variable*

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*.

print-pprint-dispatch *Variable*

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*.

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-pprint-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).

print-pretty *Variable*

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
```

See Also:

write

∗print-readably∗ *Variable*

Value Type:

a *generalized boolean*.

print-readably

Initial Value:

false.

Description:

If ***print-readably*** is *true*, some special rules for printing *objects* go into effect. Specifically, printing any *object* O_1 produces a printed representation that, when seen by the *Lisp reader* while the *standard readtable* is in effect, will produce an *object* O_2 that is *similar* to O_1. 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 readably* 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)
       (*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>
```

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**.

∗print-right-margin∗ *Variable*

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.

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.

print-not-readable *Condition Type*

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 `:object` initialization argument to **make-condition**, and is *accessed* by the *function* **print-not-readable-object**.

See Also:

print-not-readable-object

print-not-readable-object *Function*

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*.

See Also:

print-not-readable, Chapter 9 (Conditions)

format *Function*

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:

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).

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)

Programming Language—Common Lisp

23. Reader

23.1 Reader Concepts

23.1.1 Dynamic Control of the Lisp Reader

Various aspects of the *Lisp reader* can be controlled dynamically. See Section 2.1.1 (Readtables) and Section 2.1.2 (Variables that affect the Lisp Reader).

23.1.2 Effect of Readtable Case on the Lisp Reader

The *readtable case* of the *current readtable* affects the *Lisp reader* in the following ways:

`:upcase`

When the *readtable case* is `:upcase`, unescaped constituent *characters* are converted to *uppercase*, as specified in Section 2.2 (Reader Algorithm).

`:downcase`

When the *readtable case* is `:downcase`, unescaped constituent *characters* are converted to *lowercase*.

`:preserve`

When the *readtable case* is `:preserve`, the case of all *characters* remains unchanged.

`:invert`

When the *readtable case* is `:invert`, then if all of the unescaped letters in the extended token are of the same *case*, those (unescaped) letters are converted to the opposite *case*.

23.1.2.1 Examples of Effect of Readtable Case on the Lisp Reader

```
(defun test-readtable-case-reading ()
  (let ((*readtable* (copy-readtable nil)))
    (format t "READTABLE-CASE  Input   Symbol-name~
             ~%-------------------~
             ~%")
    (dolist (readtable-case '(:upcase :downcase :preserve :invert))
      (setf (readtable-case *readtable*) readtable-case)
      (dolist (input '("ZEBRA" "Zebra" "zebra"))
        (format t "~&:~A~16T~A~24T~A"
                (string-upcase readtable-case)
                input
```

```
                    (symbol-name (read-from-string input)))))))
```

The output from `(test-readtable-case-reading)` should be as follows:

```
READTABLE-CASE     Input Symbol-name
-------------------
   :UPCASE         ZEBRA   ZEBRA
   :UPCASE         Zebra   ZEBRA
   :UPCASE         zebra   ZEBRA
   :DOWNCASE       ZEBRA   zebra
   :DOWNCASE       Zebra   zebra
   :DOWNCASE       zebra   zebra
   :PRESERVE       ZEBRA   ZEBRA
   :PRESERVE       Zebra   Zebra
   :PRESERVE       zebra   zebra
   :INVERT         ZEBRA   zebra
   :INVERT         Zebra   Zebra
   :INVERT         zebra   ZEBRA
```

23.1.3 Argument Conventions of Some Reader Functions

23.1.3.1 The EOF-ERROR-P argument

Eof-error-p in input function calls controls what happens if input is from a file (or any other input source that has a definite end) and the end of the file is reached. If *eof-error-p* is *true* (the default), an error of *type* **end-of-file** is signaled at end of file. If it is *false*, then no error is signaled, and instead the function returns *eof-value*.

Functions such as **read** that read the representation of an *object* rather than a single character always signals an error, 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. If a file ends in a *symbol* or a *number* immediately followed by end-of-file, **read** reads the *symbol* or *number* successfully and when called again will act according to *eof-error-p*. Similarly, the *function* **read-line** successfully reads the last line of a file even if that line is terminated by end-of-file rather than the newline character. Ignorable text, such as lines containing only *whitespace*$_2$ or comments, are not considered to begin an *object*; if **read** begins to read an *expression* but sees only such ignorable text, it does not consider the file to end in the middle of an *object*. Thus an *eof-error-p* argument controls what happens when the file ends between *objects*.

23.1.3.2 The RECURSIVE-P argument

If *recursive-p* is supplied and not **nil**, it specifies that this function call is not an outermost call to **read** but an embedded call, typically from a *reader macro function*. It is important to distinguish such recursive calls for three reasons.

1. An outermost call establishes the context within which the `#`*n*`=` and `#`*n*`#` syntax is scoped. Consider, for example, the expression

   ```
   (cons '#3=(p q r) '(x y . #3#))
   ```

   If the *single-quote reader macro* were defined in this way:

   ```
   (set-macro-character #\'       ;incorrect
      #'(lambda (stream char)
           (declare (ignore char))
           (list 'quote (read stream))))
   ```

   then each call to the *single-quote reader macro function* would establish independent contexts for the scope of **read** information, including the scope of identifications between markers like "`#3=`" and "`#3#`". However, for this expression, the scope was clearly intended to be determined by the outer set of parentheses, so such a definition would be incorrect. The correct way to define the *single-quote reader macro* uses *recursive-p*:

   ```
   (set-macro-character #\'       ;correct
      #'(lambda (stream char)
           (declare (ignore char))
           (list 'quote (read stream t nil t))))
   ```

2. A recursive call does not alter whether the reading process is to preserve *whitespace*$_2$ or not (as determined by whether the outermost call was to **read** or **read-preserving-whitespace**). Suppose again that *single-quote* were to be defined as shown above in the incorrect definition. Then a call to **read-preserving-whitespace** that read the expression `'foo`⟨*Space*⟩ would fail to preserve the space character following the symbol `foo` because the *single-quote reader macro function* calls **read**, not **read-preserving-whitespace**, to read the following expression (in this case `foo`). The correct definition, which passes the value *true* for *recursive-p* to **read**, allows the outermost call to determine whether *whitespace*$_2$ is preserved.

3. When end-of-file is encountered and the *eof-error-p* argument is not **nil**, the kind of error that is signaled may depend on the value of *recursive-p*. If *recursive-p* is *true*, then the end-of-file is deemed to have occurred within the middle of a printed representation; if *recursive-p* is *false*, then the end-of-file may be deemed to have occurred between *objects* rather than within the middle of one.

readtable *System Class*

Class Precedence List:

readtable, **t**

Description:

A *readtable* maps *characters* into *syntax types* for the *Lisp reader*; see Chapter 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:

Section 2.1.1 (Readtables), Section 22.1.3.13 (Printing Other Objects)

copy-readtable *Function*

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
```

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))) ...)
```

make-dispatch-macro-character *Function*

Syntax:

make-dispatch-macro-character *char* `&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*. 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
```

The *readtable* is altered.

See Also:

readtable, **set-dispatch-macro-character**

read, read-preserving-whitespace *Function*

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*.

read, read-preserving-whitespace

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))
```

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**, Chapter 2 (Syntax), Section 23.1 (Reader Concepts)

read-delimited-list *Function*

Syntax:

read-delimited-list *char* `&optional` *input-stream recursive-p* $\rightarrow$ *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.

read-delimited-list

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 peeks 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 readtable*.

Examples:

```
(read-delimited-list #\]) 1 2 3 4 5 6 ]
→ (1 2 3 4 5 6)
```

Suppose you wanted `#{`*a b c ... 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))
```

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***.

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.

read-from-string *Function*

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
```

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.

readtable-case *Accessor*

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).

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*.

readtablep *Function*

Syntax:

readtablep *object* → *generalized-boolean*

Arguments and Values:

object—an *object*.

generalized-boolean—a *generalized boolean*.

Description:

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

Examples:

```
(readtablep *readtable*) → true
(readtablep (copy-readtable)) → true
(readtablep '*readtable*) → false
```

Notes:

```
(readtablep object) ≡ (typep object 'readtable)
```

set-dispatch-macro-character, get-dispatch-macro-character *Function*

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 `(dollars` *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:

Section 2.1.4.4 (Macro Characters)

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:

It is necessary to use **make-dispatch-macro-character** to set up the dispatch character before specifying its sub-characters.

set-macro-character, get-macro-character *Function*

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*.

set-macro-character, get-macro-character

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))
  (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.

See Also:

readtable

set-syntax-from-char *Function*

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*.

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).

with-standard-io-syntax *Macro*

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/printer 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 Figure 23–1.

Variable	Value
package	The `CL-USER` *package*
print-array	**t**
print-base	`10`
print-case	`:upcase`
print-circle	**nil**
print-escape	**t**
print-gensym	**t**
print-length	**nil**
print-level	**nil**
print-lines	**nil**
print-miser-width	**nil**
print-pprint-dispatch	The *standard pprint dispatch table*
print-pretty	**nil**
print-radix	**nil**
print-readably	**t**
print-right-margin	**nil**
read-base	`10`
read-default-float-format	**single-float**
read-eval	**t**
read-suppress	**nil**
readtable	The *standard readtable*

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))))
```

∗read-base∗ *Variable*

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))
▷ (15 (DAD 3088 BEE 2699 123 258))
→ NIL
```

Notes:

Altering the input radix can be useful when reading data files in special formats.

∗read-default-float-format∗ *Variable*

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   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.
```

∗read-eval∗ *Variable*

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**.

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**.

∗read-suppress∗ *Variable*

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 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 NIL)
```

See Also:

read, Chapter 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.

∗readtable∗ *Variable*

Value Type:

a *readtable*.

Initial Value:

A *readtable* that conforms to the description of Common Lisp syntax in Chapter 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* **readtable-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)

reader-error *Condition Type*

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)

Programming Language—Common Lisp

24. System Construction

24.1 System Construction Concepts

24.1.1 Loading

To **load** a *file* is to treat its contents as *code* and *execute* that *code*. The *file* may contain ***source code*** or ***compiled code***.

A *file* containing *source code* is called a ***source file***. *Loading* a *source file* is accomplished essentially by sequentially *reading*$_2$ the *forms* in the file, *evaluating* each immediately after it is *read*.

A *file* containing *compiled code* is called a ***compiled file***. *Loading* a *compiled file* is similar to *loading* a *source file*, except that the *file* does not contain text but rather an *implementation-dependent* representation of pre-digested *expressions* created by the *compiler*. Often, a *compiled file* can be *loaded* more quickly than a *source file*. See Section 3.2 (Compilation).

The way in which a *source file* is distinguished from a *compiled file* is *implementation-dependent*.

24.1.2 Features

A ***feature*** is an aspect or attribute of Common Lisp, of the *implementation*, or of the *environment*. A *feature* is identified by a *symbol*.

A *feature* is said to be ***present*** in a *Lisp image* if and only if the *symbol* naming it is an *element* of the *list* held by the *variable* ***features***, which is called the ***features list***.

24.1.2.1 Feature Expressions

Boolean combinations of *features*, called ***feature expressions***, are used by the `#+` and `#-` *reader macros* in order to direct conditional *reading* of *expressions* by the *Lisp reader*.

The rules for interpreting a *feature expression* are as follows:

feature

If a *symbol* naming a *feature* is used as a *feature expression*, the *feature expression* succeeds if that *feature* is *present*; otherwise it fails.

`(not` *feature-conditional*`)`

A **not** *feature expression* succeeds if its argument *feature-conditional* fails; otherwise, it succeeds.

`(and {`*feature-conditional*`}*)`

An **and** *feature expression* succeeds if all of its argument *feature-conditionals* succeed; otherwise, it fails.

`(or {feature-conditional}*)`

An **or** *feature expression* succeeds if any of its argument *feature-conditionals* succeed; otherwise, it fails.

24.1.2.1.1 Examples of Feature Expressions

For example, suppose that in *implementation* A, the *features* `spice` and `perq` are *present*, but the *feature* `lispm` is not *present*; in *implementation* B, the feature `lispm` is *present*, but the *features* `spice` and `perq` are not *present*; and in *implementation* C, none of the features `spice`, *lispm*, or `perq` are *present*. Figure 24–1 shows some sample *expressions*, and how they would be *read*$_2$ in these *implementations*.

```
(cons #+spice "Spice" #-spice "Lispm" x)
  in implementation A ...          (CONS "Spice" X)
  in implementation B ...          (CONS "Lispm" X)
  in implementation C ...          (CONS "Lispm" X)

(cons #+spice "Spice" #+LispM "Lispm" x)
  in implementation A ...          (CONS "Spice" X)
  in implementation B ...          (CONS "Lispm" X)
  in implementation C ...          (CONS X)

(setq a '(1 2 #+perq 43 #+(not perq) 27))
  in implementation A ...          (SETQ A '(1 2 43))
  in implementation B ...          (SETQ A '(1 2 27))
  in implementation C ...          (SETQ A '(1 2 27))

(let ((a 3) #+(or spice lispm) (b 3)) (foo a))
  in implementation A ...          (LET ((A 3) (B 3)) (FOO A))
  in implementation B ...          (LET ((A 3) (B 3)) (FOO A))
  in implementation C ...          (LET ((A 3)) (FOO A))

(cons #+Lispm "#+Spice" #+Spice "foo" #-(or Lispm Spice) 7 x)
  in implementation A ...          (CONS "foo" X)
  in implementation B ...          (CONS "#+Spice" X)
  in implementation C ...          (CONS 7 X)
```

Figure 24–1. Features examples

compile-file *Function*

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**.

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

compile-file

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).

Affected By:

error-output, ***standard-output***, ***compile-verbose***, ***compile-print***

The computer's file system.

Exceptional Situations:

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)

compile-file-pathname *Function*

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**.

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)

load *Function*

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.)

load

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 *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)

with-compilation-unit *Macro*

Syntax:

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.

See Also:

compile, **compile-file**

features

features *Variable*

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*.

Most *features* have *implementation-dependent* meanings; The following meanings have been assigned to feature names:

`:cltl1`

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.

`:cltl2`

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 `:cltl1` or `:cltl2`, 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.

See Also:

Section 1.5.2.1.1 (Use of Read-Time Conditionals), Section 2.4 (Standard Macro Characters)

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*.

∗compile-file-pathname∗, ∗compile-file-truename∗

Variable

Value Type:

The *value* of ***compile-file-pathname*** must always be a *pathname* or **nil**. The *value* of ***compile-file-truename*** 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-truename*** is *bound* to the *truename* 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**.

The consequences are unspecified if an attempt is made to *assign* or *bind* either of these *variables*.

Affected By:

The *file system*.

See Also:

compile-file

load-pathname, *load-truename* *Variable*

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*.

Affected By:

The *file system*.

See Also:

load

compile-print, *compile-verbose* *Variable*

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**.

See Also:

compile-file

load-print, *load-verbose* *Variable*

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**.

See Also:

load

modules *Variable*

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*.

Affected By:

provide

See Also:

provide, **require**

Notes:

The variable ***modules*** is deprecated.

provide, require *Function*

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
```

provide, require

```
;;; must be initialized elsewhere.

(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.

Programming Language—Common Lisp

25. Environment

25.1 The External Environment

25.1.1 Top level loop

The top level loop is the Common Lisp mechanism by which the user normally interacts with the Common Lisp system. This loop is sometimes referred to as the *Lisp read-eval-print loop* because it typically consists of an endless loop that reads an expression, evaluates it and prints the results.

The top level loop is not completely specified; thus the user interface is *implementation-defined*. The top level loop prints all values resulting from the evaluation of a *form*. Figure 25–1 lists variables that are maintained by the *Lisp read-eval-print loop*.

*	+	/	-
**	++	//	
***	+++	///	

Figure 25–1. Variables maintained by the Read-Eval-Print Loop

25.1.2 Debugging Utilities

Figure 25–2 shows *defined names* relating to debugging.

debugger-hook	**documentation**	**step**
apropos	**dribble**	**time**
apropos-list	**ed**	**trace**
break	**inspect**	**untrace**
describe	**invoke-debugger**	

Figure 25–2. Defined names relating to debugging

25.1.3 Environment Inquiry

Environment inquiry *defined names* provide information about the hardware and software configuration on which a Common Lisp program is being executed.

Figure 25–3 shows *defined names* relating to environment inquiry.

features	**machine-instance**	**short-site-name**
lisp-implementation-type	**machine-type**	**software-type**
lisp-implementation-version	**machine-version**	**software-version**
long-site-name	**room**	

Figure 25–3. Defined names relating to environment inquiry.

25.1.4 Time

Time is represented in four different ways in Common Lisp: *decoded time*, *universal time*, *internal time*, and seconds. *Decoded time* and *universal time* are used primarily to represent calendar time, and are precise only to one second. *Internal time* is used primarily to represent measurements of computer time (such as run time) and is precise to some *implementation-dependent* fraction of a second called an *internal time unit*, as specified by **internal-time-units-per-second**. An *internal time* can be used for either *absolute* and *relative time* measurements. Both a *universal time* and a *decoded time* can be used only for *absolute time* measurements. In the case of one function, **sleep**, time intervals are represented as a non-negative *real* number of seconds.

Figure 25–4 shows *defined names* relating to *time*.

decode-universal-time	**get-internal-run-time**
encode-universal-time	**get-universal-time**
get-decoded-time	**internal-time-units-per-second**
get-internal-real-time	**sleep**

Figure 25–4. Defined names involving Time.

25.1.4.1 Decoded Time

A ***decoded time*** is an ordered series of nine values that, taken together, represent a point in calendar time (ignoring *leap seconds*):

Second

An *integer* between 0 and 59, inclusive.

Minute

An *integer* between 0 and 59, inclusive.

Hour

An *integer* between 0 and 23, inclusive.

Date

An *integer* between 1 and 31, inclusive (the upper limit actually depends on the month and year, of course).

Month

An *integer* between 1 and 12, inclusive; 1 means January, 2 means February, and so on; 12 means December.

Year

An *integer* indicating the year A.D. However, if this *integer* is between 0 and 99, the "obvious" year is used; more precisely, that year is assumed that is equal to the *integer* modulo 100 and within fifty years of the current year (inclusive backwards and exclusive forwards). Thus, in the year 1978, year 28 is 1928 but year 27 is 2027. (Functions that return time in this format always return a full year number.)

Day of week

An *integer* between 0 and 6, inclusive; 0 means Monday, 1 means Tuesday, and so on; 6 means Sunday.

Daylight saving time flag

A *generalized boolean* that, if *true*, indicates that daylight saving time is in effect.

Time zone

A *time zone*.

Figure 25–5 shows *defined names* relating to *decoded time*.

decode-universal-time	**get-decoded-time**

Figure 25–5. Defined names involving time in Decoded Time.

25.1.4.2 Universal Time

Universal time is an *absolute time* represented as a single non-negative *integer*—the number of seconds since midnight, January 1, 1900 GMT (ignoring *leap seconds*). Thus the time 1 is 00:00:01 (that is, 12:00:01 a.m.) on January 1, 1900 GMT. Similarly, the time 2398291201 corresponds to time 00:00:01 on January 1, 1976 GMT. Recall that the year 1900 was not a leap year; for the purposes of Common Lisp, a year is a leap year if and only if its number is divisible by 4, except that years divisible by 100 are not leap years, except that years divisible by 400 are leap years.

Therefore the year 2000 will be a leap year. Because *universal time* must be a non-negative *integer*, times before the base time of midnight, January 1, 1900 GMT cannot be processed by Common Lisp.

decode-universal-time	**get-universal-time**
encode-universal-time	

Figure 25–6. Defined names involving time in Universal Time.

25.1.4.3 Internal Time

Internal time represents time as a single *integer*, in terms of an *implementation-dependent* unit called an *internal time unit*. Relative time is measured as a number of these units. Absolute time is relative to an arbitrary time base.

Figure 25–7 shows *defined names* related to *internal time*.

get-internal-real-time	**internal-time-units-per-second**
get-internal-run-time	

Figure 25–7. Defined names involving time in Internal Time.

25.1.4.4 Seconds

One function, **sleep**, takes its argument as a non-negative *real* number of seconds. Informally, it may be useful to think of this as a *relative universal time*, but it differs in one important way: *universal times* are always non-negative *integers*, whereas the argument to **sleep** can be any kind of non-negative *real*, in order to allow for the possibility of fractional seconds.

sleep

Figure 25–8. Defined names involving time in Seconds.

decode-universal-time *Function*

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.

See Also:

encode-universal-time, **get-universal-time**, Section 25.1.4 (Time)

encode-universal-time *function*

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 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
```

See Also:

decode-universal-time, **get-decoded-time**

get-universal-time, get-decoded-time *Function*

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
```

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.

sleep *Function*

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.
 (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*.

apropos, apropos-list *Function*

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*.

Affected By:

The set of *symbols* which are currently *interned* in any *packages* being searched.

apropos is also affected by ***standard-output***.

describe *Function*

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.

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*.

See Also:

inspect, **describe-object**

describe-object *Standard Generic Function*

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) ())

 (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⟩
```

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.

trace, untrace

trace, untrace *Macro*

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:

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*.

step *Macro*

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.

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.

time *Macro*

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.

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.*

See Also:

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*.

internal-time-units-per-second *Constant Variable*

Constant Value:

A positive *integer*, the magnitude of which is *implementation-dependent*.

Description:

The number of *internal time units* in one second.

See Also:

get-internal-run-time, **get-internal-real-time**

Notes:

These units form the basis of the Internal Time format representation.

get-internal-real-time *Function*

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.

Affected By:

Time of day (*i.e.*, the passage of time). The time base affects the result magnitude.

See Also:

internal-time-units-per-second

get-internal-run-time *Function*

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.

Affected By:

The *implementation*, the time of day (*i.e.*, the passage of time).

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.

disassemble *Function*

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
```

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*.

documentation, (setf documentation) *Standard Generic Function*

Syntax:

documentation *x doc-type* → *documentation*

(setf documentation) *new-value x doc-type* → *new-value*

documentation, (setf documentation)

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)`)

(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, (setf documentation)

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* **standard-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*.

documentation, (setf documentation)

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*.

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.

room *Function*

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.

Side Effects:

Output to *standard output*.

Affected By:

standard-output.

ed *Function*

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*.

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)

inspect *Function*

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.

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.

dribble *Function*

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.

Affected By:

The *implementation*.

Exceptional Situations:

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.

dribble is intended primarily for interactive debugging; its effect cannot be relied upon when used in a program.

— *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:

+ (*variable*), * (*variable*), / (*variable*), Section 25.1.1 (Top level loop)

+, ++, +++ *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)

*, **, *** *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.

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 ///)
```

/, //, /// *Variable*

Value Type:

a *proper list*.

Initial Value:

implementation-dependent.

Description:

The *variables* /, //, and /// 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 / is a *list* of the most recent *values* that were printed, the *value* of // is the previous value of /, and the *value* of /// is the previous value of //.

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:

```
(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)

lisp-implementation-type, lisp-implementation-version *Function*

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"
```

short-site-name, long-site-name *Function*

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)
→ "MIT Artificial Intelligence Laboratory"
or→ "CMU Computer Science Department"
```

Affected By:

The implementation, the location of the computer hardware, and the installation/configuration process.

machine-instance *Function*

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"
```

Affected By:

The machine instance, and the *implementation*.

See Also:

machine-type, **machine-version**

machine-type *Function*

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"
```

Affected By:

The machine type. The implementation.

See Also:

machine-version

machine-version *Function*

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"
```

Affected By:

The machine version, and the *implementation*.

See Also:

machine-type, **machine-instance**

software-type, software-version *Function*

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"
```

Affected By:

Operating system environment.

Notes:

This information should be of use to maintainers of the *implementation*.

user-homedir-pathname

user-homedir-pathname *Function*

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).

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
```

Affected By:

The host computer's file system, and the *implementation*.

Programming Language—Common Lisp

26. Glossary

26.1 Glossary

Each entry in this glossary has the following parts:

- the term being defined, set in boldface.
- optional pronunciation, enclosed in square brackets and set in boldface, as in the following example: [**ˈaˌlist**]. The pronunciation key follows *Webster's Third New International Dictionary the English Language, Unabridged*, except that "ϵ" is used to notate the schwa (upside-down "e") character.
- the part or parts of speech, set in italics. If a term can be used as several parts of speech, there is a separate definition for each part of speech.
- one or more definitions, organized as follows:
  - an optional number, present if there are several definitions. Lowercase letters might also be used in cases where subdefinitions of a numbered definition are necessary.
  - an optional part of speech, set in italics, present if the term is one of several parts of speech.
  - an optional discipline, set in italics, present if the term has a standard definition being repeated. For example, "*Math.*"
  - an optional context, present if this definition is meaningful only in that context. For example, "(of a *symbol*)".
  - the definition.
  - an optional example sentence. For example, "This is an example of an example."
  - optional cross references.

In addition, some terms have idiomatic usage in the Common Lisp community which is not shared by other communities, or which is not technically correct. Definitions labeled "*Idiom.*" represent such idiomatic usage; these definitions are sometimes followed by an explanatory note.

Words in *this font* are words with entries in the glossary. Words in example sentences do not follow this convention.

When an ambiguity arises, the longest matching substring has precedence. For example, "*complex float*" refers to a single glossary entry for "*complex float*" rather than the combined meaning of the glossary terms "*complex*" and "*float*."

Subscript notation, as in "*something*$_n$" means that the nth definition of "*something*" is intended. This notation is used only in situations where the context might be insufficient to disambiguate.

The following are abbreviations used in the glossary:

Abbreviation	**Meaning**
adj.	adjective
adv.	adverb
ANSI	compatible with one or more ANSI standards
Comp.	computers
Idiom.	idiomatic
IEEE	compatible with one or more IEEE standards
ISO	compatible with one or more ISO standards
Math.	mathematics
Trad.	traditional
n.	noun
v.	verb
v.t.	transitive verb

Non-alphabetic

() [ˈ**nil**], *n.* an alternative notation for writing the symbol **nil**, used to emphasize the use of *nil* as an *empty list*.

A

absolute *adj.* 1. (of a *time*) representing a specific point in time. 2. (of a *pathname*) representing a specific position in a directory hierarchy. See *relative*.

access *n.*, *v.t.* 1. *v.t.* (a *place*, or *array*) to *read*$_1$ or *write*$_1$ the *value* of the *place* or an *element* of the *array*. 2. *n.* (of a *place*) an attempt to *access*$_1$ the *value* of the *place*.

accessibility *n.* the state of being *accessible*.

accessible *adj.* 1. (of an *object*) capable of being *referenced*. 2. (of *shared slots* or *local slots* in an *instance* of a *class*) having been defined by the *class* of the *instance* or *inherited* from a *superclass* of that *class*. 3. (of a *symbol* in a *package*) capable of being *referenced* without a *package prefix* when that *package* is current, regardless of whether the *symbol* is *present* in that *package* or is *inherited*.

accessor *n.* an *operator* that performs an *access*. See *reader* and *writer*.

active *adj.* 1. (of a *handler*, a *restart*, or a *catch tag*) having been *established* but not yet *disestablished*. 2. (of an *element* of an *array*) having an index that is greater than or equal to zero, but less than the *fill pointer* (if any). For an *array* that has no *fill pointer*, all *elements* are considered *active*.

actual adjustability *n.* (of an *array*) a *generalized boolean* that is associated with the *array*, representing whether the *array* is *actually adjustable*. See also *expressed adjustability* and **adjustable-array-p**.

actual argument *n. Trad.* an *argument*.

actual array element type *n.* (of an *array*) the *type* for which the *array* is actually specialized, which is the *upgraded array element type* of the *expressed array element type* of the *array*. See the *function* **array-element-type**.

actual complex part type *n.* (of a *complex*) the *type* in which the real and imaginary parts of the *complex* are actually represented, which is the *upgraded complex part type* of the *expressed complex part type* of the *complex*.

actual parameter *n. Trad.* an *argument*.

actually adjustable *adj.* (of an *array*) such that **adjust-array** can adjust its characteristics by direct modification. A *conforming program* may depend on an *array* being *actually adjustable* only if either that *array* is known to have been *expressly adjustable* or if that *array* has been explicitly tested by **adjustable-array-p**.

adjustability *n.* (of an *array*) 1. *expressed adjustability*. 2. *actual adjustability*.

adjustable *adj.* (of an *array*) 1. *expressly adjustable*. 2. *actually adjustable*.

after method *n.* a *method* having the *qualifier* `:after`.

alist [ˈā͟ˌlist], *n.* an *association list*.

alphabetic *n.*, *adj.* 1. *adj.* (of a *character*) being one of the *standard characters* `A` through `Z` or `a` through `z`, or being any *implementation-defined* character that has *case*, or being some other *graphic character* defined by the *implementation* to be *alphabetic*$_1$. 2. a. *n.* one of several possible *constituent traits* of a *character*. For details, see Section 2.1.4.1 (Constituent Characters) and Section 2.2 (Reader Algorithm). b. *adj.* (of a *character*) being a *character* that has *syntax type constituent* in the *current readtable* and that has the *constituent trait alphabetic*$_{2a}$. See Figure 2–8.

alphanumeric *adj.* (of a *character*) being either an *alphabetic*$_1$ *character* or a *numeric* character.

ampersand *n.* the *standard character* that is called "ampersand" (`&`). See Figure 2–5.

anonymous *adj.* 1. (of a *class* or *function*) having no *name* 2. (of a *restart*) having a *name* of **nil**.

apparently uninterned *adj.* having a *home package* of **nil**. (An *apparently uninterned symbol* might or might not be an *uninterned symbol*. *Uninterned symbols* have a *home package* of **nil**, but *symbols* which have been *uninterned* from their *home package* also have a *home package* of **nil**, even though they might still be *interned* in some other *package*.)

applicable *adj.* 1. (of a *handler*) being an *applicable handler*. 2. (of a *method*) being an *applicable method*. 3. (of a *restart*) being an *applicable restart*.

applicable handler *n.* (for a *condition* being *signaled*) an *active handler* for which the associated type contains the *condition*.

applicable method *n.* (of a *generic function* called with *arguments*) a *method* of the *generic function* for which the *arguments* satisfy the *parameter specializers* of that *method*. See Section 7.6.6.1.1 (Selecting the Applicable Methods).

applicable restart *n.* 1. (for a *condition*) an *active handler* for which the associated test returns *true* when given the *condition* as an argument. 2. (for no particular *condition*) an *active handler* for which the associated test returns *true* when given **nil** as an argument.

apply *v.t.* (a *function* to a *list*) to *call* the *function* with arguments that are the *elements* of the *list*. "Applying the function + to a list of integers returns the sum of the elements of that list."

argument *n.* 1. (of a *function*) an *object* which is offered as data to the *function* when it is *called*. 2. (of a *format control*) a *format argument*.

argument evaluation order *n.* the order in which *arguments* are evaluated in a function call. "The argument evaluation order for Common Lisp is left to right." See Section 3.1 (Evaluation).

argument precedence order *n.* the order in which the *arguments* to a *generic function* are considered when sorting the *applicable methods* into precedence order.

around method *n.* a *method* having the *qualifier* `:around`.

array *n.* an *object* of *type* **array**, which serves as a container for other *objects* arranged in a Cartesian coordinate system.

array element type *n.* (of an *array*) 1. a *type* associated with the *array*, and of which all *elements* of the *array* are constrained to be members. 2. the *actual array element type* of the *array*. 3. the *expressed array element type* of the *array*.

array total size *n.* the total number of *elements* in an *array*, computed by taking the product of the *dimensions* of the *array*. (The size of a zero-dimensional *array* is therefore one.)

assign *v.t.* (a *variable*) to change the *value* of the *variable* in a *binding* that has already been *established*. See the *special operator* **setq**.

association list *n.* a *list* of *conses* representing an association of *keys* with *values*, where the *car* of each *cons* is the *key* and the *cdr* is the *value* associated with that *key*.

asterisk *n.* the *standard character* that is variously called "asterisk" or "star" (`*`). See Figure 2–5.

at-sign *n.* the *standard character* that is variously called "commercial at" or "at sign" (`@`). See Figure 2–5.

atom *n.* any *object* that is not a *cons*. "A vector is an atom."

atomic *adj.* being an *atom*. "The number 3, the symbol `foo`, and **nil** are atomic."

atomic type specifier *n.* a *type specifier* that is *atomic*. For every *atomic type specifier*, *x*, there is an equivalent *compound type specifier* with no arguments supplied, `(`*x*`)`.

attribute *n.* (of a *character*) a program-visible aspect of the *character*. The only *standardized attribute* of a *character* is its *code*$_2$, but *implementations* are permitted to have additional *implementation-defined attributes*. See Section 13.1.3 (Character Attributes). "An implementation that support fonts might make font information an attribute of a character, while others might represent font information separately from characters."

aux variable *n.* a *variable* that occurs in the part of a *lambda list* that was introduced by **&aux**. Unlike all other *variables* introduced by a *lambda-list*, *aux variables* are not *parameters*.

auxiliary method *n.* a member of one of two sets of *methods* (the set of *primary methods* is the other) that form an exhaustive partition of the set of *methods* on the *method*'s *generic function*. How these sets are determined is dependent on the *method combination* type; see Section 7.6.2 (Introduction to Methods).

B

backquote *n.* the *standard character* that is variously called "grave accent" or "backquote" (`). See Figure 2–5.

backslash *n.* the *standard character* that is variously called "reverse solidus" or "backslash" (\). See Figure 2–5.

base character *n.* a *character* of *type* **base-char**.

base string *n.* a *string* of *type* **base-string**.

before method *n.* a *method* having the *qualifier* `:before`.

bidirectional *adj.* (of a *stream*) being both an *input stream* and an *output stream*.

binary *adj.* 1. (of a *stream*) being a *stream* that has an *element type* that is a *subtype* of *type* **integer**. The most fundamental operation on a *binary input stream* is **read-byte** and on a *binary output stream* is **write-byte**. See *character*. 2. (of a *file*) having been created by opening a *binary stream*. (It is *implementation-dependent* whether this is an detectable aspect of the *file*, or whether any given *character file* can be treated as a *binary file*.)

bind *v.t.* (a *variable*) to establish a *binding* for the *variable*.

binding *n.* an association between a *name* and that which the *name* denotes. "A lexical binding is a lexical association between a name and its value." When the term *binding* is qualified by the name of a *namespace*, such as "variable" or "function," it restricts the binding to the indicated namespace, as in: "**let** establishes variable bindings." or "**let** establishes bindings of variables."

bit *n.* an *object* of *type* **bit**; that is, the *integer* `0` or the *integer* `1`.

bit array *n.* a specialized *array* that is of *type* `(array bit)`, and whose elements are of *type* **bit**.

bit vector *n.* a specialized *vector* that is of *type* **bit-vector**, and whose elements are of *type* **bit**.

bit-wise logical operation specifier *n.* an *object* which names one of the sixteen possible bit-wise logical operations that can be performed by the **boole** function, and which is the *value* of exactly one of the *constant variables* **boole-clr**, **boole-set**, **boole-1**, **boole-2**, **boole-c1**, **boole-c2**, **boole-and**, **boole-ior**, **boole-xor**, **boole-eqv**, **boole-nand**, **boole-nor**, **boole-andc1**, **boole-andc2**, **boole-orc1**, or **boole-orc2**.

block *n.* a named lexical *exit point*, *established* explicitly by **block** or implicitly by *operators* such as **loop**, **do** and **prog**, to which control and values may be transfered by using a **return-from** *form* with the name of the *block*.

block tag *n.* the *symbol* that, within the *lexical scope* of a **block** *form*, names the *block established* by that **block** *form*. See **return** or **return-from**.

boa lambda list *n.* a *lambda list* that is syntactically like an *ordinary lambda list*, but that is processed in "**b**y **o**rder of **a**rgument" style. See Section 3.4.6 (Boa Lambda Lists).

body parameter *n.* a *parameter* available in certain *lambda lists* which from the point of view of *conforming programs* is like a *rest parameter* in every way except that it is introduced by **&body** instead of **&rest**. (*Implementations* are permitted to provide extensions which distinguish *body parameters* and *rest parameters*—*e.g.*, the *forms* for *operators* which were defined using a *body parameter* might be pretty printed slightly differently than *forms* for *operators* which were defined using *rest parameters*.)

boolean *n.* an *object* of *type* **boolean**; that is, one of the following *objects*: the symbol **t** (representing *true*), or the symbol **nil** (representing *false*). See *generalized boolean*.

boolean equivalent *n.* (of an *object* O_1) any *object* O_2 that has the same truth value as O_1 when both O_1 and O_2 are viewed as *generalized booleans*.

bound *adj.*, *v.t.* 1. *adj.* having an associated denotation in a *binding*. "The variables named by a **let** are bound within its body." See *unbound*. 2. *adj.* having a local *binding* which *shadows*$_2$ another. "The variable ***print-escape*** is bound while in the **princ** function." 3. *v.t.* the past tense of *bind*.

bound declaration *n.* a *declaration* that refers to or is associated with a *variable* or *function* and that appears within the *special form* that *establishes* the *variable* or *function*, but before the body of that *special form* (specifically, at the head of that *form*'s body). (If a *bound declaration* refers to a *function binding* or a *lexical variable binding*, the *scope* of the *declaration* is exactly the *scope* of that *binding*. If the *declaration* refers to a *dynamic variable binding*, the *scope* of the *declaration* is what the *scope* of the *binding* would have been if it were lexical rather than dynamic.)

bounded *adj.* (of a *sequence* S, by an ordered pair of *bounding indices* i_{start} and i_{end}) restricted to a subrange of the *elements* of S that includes each *element* beginning with (and including) the one indexed by i_{start} and continuing up to (but not including) the one indexed by i_{end}.

bounding index *n.* (of a *sequence* with *length* n) either of a conceptual pair of *integers*, i_{start} and i_{end}, respectively called the "lower bounding index" and "upper bounding index", such that $0 \leq i_{start} \leq i_{end} \leq n$, and which therefore delimit a subrange of the *sequence bounded* by i_{start} and i_{end}.

bounding index designator (for a *sequence*) one of two *objects* that, taken together as an ordered pair, behave as a *designator* for *bounding indices* of the *sequence*; that is, they denote *bounding indices* of the *sequence*, and are either: an *integer* (denoting itself) and **nil** (denoting the *length* of the *sequence*), or two *integers* (each denoting themselves).

break loop *n.* A variant of the normal *Lisp read-eval-print loop* that is recursively entered, usually because the ongoing *evaluation* of some other *form* has been suspended for the purpose of debugging. Often, a *break loop* provides the ability to exit in such a way as to continue the suspended computation. See the *function* **break**.

broadcast stream *n.* an *output stream* of *type* **broadcast-stream**.

built-in class *n.* a *class* that is a *generalized instance* of *class* **built-in-class**.

built-in type *n.* one of the *types* in Figure 4–2.

byte *n.* 1. adjacent bits within an *integer*. (The specific number of bits can vary from point to point in the program; see the *function* **byte**.) 2. an integer in a specified range. (The specific range can vary from point to point in the program; see the *functions* **open** and **write-byte**.)

byte specifier *n.* An *object* of *implementation-dependent* nature that is returned by the *function* **byte** and that specifies the range of bits in an *integer* to be used as a *byte* by *functions* such as **ldb**.

C

cadr [ˈ**ka**ˌ**dɛr**], *n.* (of an *object*) the *car* of the *cdr* of that *object*.

call *v.t.*, *n.* 1. *v.t.* (a *function* with *arguments*) to cause the *code* represented by that *function* to be *executed* in an *environment* where *bindings* for the *values* of its *parameters* have been *established* based on the *arguments*. "Calling the function + with the arguments `5` and `1` yields a value of `6`." 2. *n.* a *situation* in which a *function* is called.

captured initialization form *n.* an *initialization form* along with the *lexical environment* in which the *form* that defined the *initialization form* was *evaluated*. "Each newly added shared slot is set to the result of evaluating the captured initialization form for the slot that was specified in the **defclass** form for the new class."

car *n.* 1. a. (of a *cons*) the component of a *cons* corresponding to the first *argument* to **cons**; the other component is the *cdr*. "The function **rplaca** modifies the car of a cons." b. (of a *list*) the first *element* of the *list*, or **nil** if the *list* is the *empty list*. 2. the *object* that is held in the car_1. "The function **car** returns the car of a cons."

case *n.* (of a *character*) the property of being either *uppercase* or *lowercase*. Not all *characters* have *case*. "The characters `#\A` and `#\a` have case, but the character `#\$` has no case." See Section 13.1.4.3 (Characters With Case) and the *function* **both-case-p**.

case sensitivity mode *n.* one of the *symbols* `:upcase`, `:downcase`, `:preserve`, or `:invert`.

catch *n.* an *exit point* which is *established* by a **catch** *form* within the *dynamic scope* of its body, which is named by a *catch tag*, and to which control and *values* may be *thrown*.

catch tag *n.* an *object* which names an *active catch*. (If more than one *catch* is active with the same *catch tag*, it is only possible to *throw* to the innermost such *catch* because the outer one is $shadowed_2$.)

cddr [ˈku̲dϵˌdϵr] or [ˈkϵˌdu̲dϵr], *n.* (of an *object*) the *cdr* of the *cdr* of that *object*.

cdr [ˈku̲ˌdϵr], *n.* 1. a. (of a *cons*) the component of a *cons* corresponding to the second *argument* to **cons**; the other component is the *car*. "The function **rplacd** modifies the cdr of a cons." b. (of a *list* L_1) either the *list* L_2 that contains the *elements* of L_1 that follow after the first, or else **nil** if L_1 is the *empty list*. 2. the *object* that is held in the cdr_1. "The function **cdr** returns the cdr of a cons."

cell *n. Trad.* (of an *object*) a conceptual *slot* of that *object*. The *dynamic variable* and global *function bindings* of a *symbol* are sometimes referred to as its *value cell* and *function cell*, respectively.

character *n.*, *adj.* 1. *n.* an *object* of *type* **character**; that is, an *object* that represents a unitary token in an aggregate quantity of text; see Section 13.1 (Character Concepts). 2. *adj.* a. (of a *stream*) having an *element type* that is a *subtype* of *type* **character**. The most fundamental operation on a *character input stream* is **read-char** and on a *character output stream* is **write-char**. See *binary*. b. (of a *file*) having been created by opening a *character stream*. (It is *implementation-dependent* whether this is an inspectable aspect of the *file*, or whether any given *binary file* can be treated as a *character file*.)

character code *n.* 1. one of possibly several *attributes* of a *character*. 2. a non-negative *integer* less than the *value* of **char-code-limit** that is suitable for use as a *character* $code_1$.

character designator *n.* a *designator* for a *character*; that is, an *object* that denotes a *character* and that is one of: a *designator* for a *string* of *length* one (denoting the *character* that is its only *element*), or a *character* (denoting itself).

circular *adj.* 1. (of a *list*) a *circular list*. 2. (of an arbitrary *object*) having a *component*, *element*, *constituent*$_2$, or *subexpression* (as appropriate to the context) that is the *object* itself.

circular list *n.* a chain of *conses* that has no termination because some *cons* in the chain is the *cdr* of a later *cons*.

class *n.* 1. an *object* that uniquely determines the structure and behavior of a set of other *objects* called its *direct instances*, that contributes structure and behavior to a set of other *objects* called its *indirect instances*, and that acts as a *type specifier* for a set of objects called its *generalized instances*. "The class **integer** is a subclass of the class **number**." (Note that the phrase "the *class* `foo`" is often substituted for the more precise phrase "the *class* named `foo`"—in both cases, a *class object* (not a *symbol*) is denoted.) 2. (of an *object*) the uniquely determined *class* of which the *object* is a *direct instance*. See the *function* **class-of**. "The class of the object returned by **gensym** is **symbol**." (Note that with this usage a phrase such as "its *class* is `foo`" is often substituted for the more precise phrase "its *class* is the *class* named `foo`"—in both cases, a *class object* (not a *symbol*) is denoted.)

class designator *n.* a *designator* for a *class*; that is, an *object* that denotes a *class* and that is one of: a *symbol* (denoting the *class* named by that *symbol*; see the *function* **find-class**) or a *class* (denoting itself).

class precedence list *n.* a unique total ordering on a *class* and its *superclasses* that is consistent with the *local precedence orders* for the *class* and its *superclasses*. For detailed information, see Section 4.3.5 (Determining the Class Precedence List).

close *v.t.* (a *stream*) to terminate usage of the *stream* as a source or sink of data, permitting the *implementation* to reclaim its internal data structures, and to free any external resources which might have been locked by the *stream* when it was opened.

closed *adj.* (of a *stream*) having been *closed* (see close). Some (but not all) operations that are valid on *open streams* are not valid on *closed streams*. See Section 21.1.1.1.2 (Open and Closed Streams).

closure *n.* a *lexical closure*.

coalesce *v.t.* (*literal objects* that are *similar*) to consolidate the identity of those *objects*, such that they become the *same object*. See Section 3.2.1 (Compiler Terminology).

code *n.* 1. *Trad.* any representation of actions to be performed, whether conceptual or as an actual *object*, such as *forms*, *lambda expressions*, *objects* of *type function*, text in a *source file*, or instruction sequences in a *compiled file*. This is a generic term; the specific nature of the representation depends on its context. 2. (of a *character*) a *character code*.

coerce *v.t.* (an *object* to a *type*) to produce an *object* from the given *object*, without modifying that *object*, by following some set of coercion rules that must be specifically stated for any context in which this term is used. The resulting *object* is necessarily of the indicated *type*, except when that type is a *subtype* of *type* **complex**; in that case, if a *complex rational* with an imaginary part of zero would result, the result is a *rational* rather than a *complex*—see Section 12.1.5.3 (Rule of Canonical Representation for Complex Rationals).

colon *n.* the *standard character* that is called "colon" (:). See Figure 2–5.

comma *n.* the *standard character* that is called "comma" (,). See Figure 2–5.

compilation *n.* the process of *compiling code* by the *compiler*.

compilation environment *n.* 1. An *environment* that represents information known by the *compiler* about a *form* that is being *compiled*. See Section 3.2.1 (Compiler Terminology). 2. An *object* that represents the *compilation* $environment_1$ and that is used as a second argument to a *macro function* (which supplies a *value* for any **&environment** *parameter* in the *macro function*'s definition).

compilation unit *n.* an interval during which a single unit of compilation is occurring. See the *macro* **with-compilation-unit**.

compile *v.t.* 1. (*code*) to perform semantic preprocessing of the *code*, usually optimizing one or more qualities of the code, such as run-time speed of *execution* or run-time storage usage. The minimum semantic requirements of compilation are that it must remove all macro calls and arrange for all *load time values* to be resolved prior to run time. 2. (a *function*) to produce a new *object* of *type* **compiled-function** which represents the result of *compiling* the *code* represented by the *function*. See the *function* **compile**. 3. (a *source file*) to produce a *compiled file* from a *source file*. See the *function* **compile-file**.

compile time *n.* the duration of time that the *compiler* is processing *source code*.

compile-time definition *n.* a definition in the *compilation environment*.

compiled code *n.* 1. *compiled functions*. 2. *code* that represents *compiled functions*, such as the contents of a *compiled file*.

compiled file *n.* a *file* which represents the results of *compiling* the *forms* which appeared in a corresponding *source file*, and which can be *loaded*. See the *function* **compile-file**.

compiled function *n.* an *object* of *type* **compiled-function**, which is a *function* that has been *compiled*, which contains no references to *macros* that must be expanded at run time, and which contains no unresolved references to *load time values*.

compiler *n.* a facility that is part of Lisp and that translates *code* into an *implementation-dependent* form that might be represented or *executed* efficiently. The functions **compile** and **compile-file** permit programs to invoke the *compiler*.

compiler macro *n.* an auxiliary macro definition for a globally defined *function* or *macro* which might or might not be called by any given *conforming implementation* and which must preserve the semantics of the globally defined *function* or *macro* but which might perform some additional optimizations. (Unlike a *macro*, a *compiler macro* does not extend the syntax of Common Lisp; rather, it provides an alternate implementation strategy for some existing syntax or functionality.)

compiler macro expansion *n.* 1. the process of translating a *form* into another *form* by a *compiler macro*. 2. the *form* resulting from this process.

compiler macro form *n.* a *function form* or *macro form* whose *operator* has a definition as a *compiler macro*, or a **funcall** *form* whose first *argument* is a **function** *form* whose *argument* is the *name* of a *function* that has a definition as a *compiler macro*.

compiler macro function *n.* a *function* of two arguments, a *form* and an *environment*, that implements *compiler macro expansion* by producing either a *form* to be used in place of the original argument *form* or else **nil**, indicating that the original *form* should not be replaced. See Section 3.2.2.1 (Compiler Macros).

complex *n.* an *object* of *type* **complex**.

complex float *n.* an *object* of *type* **complex** which has a *complex part type* that is a *subtype* of **float**. A *complex float* is a *complex*, but it is not a *float*.

complex part type *n.* (of a *complex*) 1. the *type* which is used to represent both the real part and the imaginary part of the *complex*. 2. the *actual complex part type* of the *complex*. 3. the *expressed complex part type* of the *complex*.

complex rational *n.* an *object* of *type* **complex** which has a *complex part type* that is a *subtype* of **rational**. A *complex rational* is a *complex*, but it is not a *rational*. No *complex rational* has an imaginary part of zero because such a number is always represented by Common Lisp as an *object* of *type* **rational**; see Section 12.1.5.3 (Rule of Canonical Representation for Complex Rationals).

complex single float *n.* an *object* of *type* **complex** which has a *complex part type* that is a *subtype* of **single-float**. A *complex single float* is a *complex*, but it is not a *single float*.

composite stream *n.* a *stream* that is composed of one or more other *streams*. "**make-synonym-stream** creates a composite stream."

compound form *n.* a *non-empty list* which is a *form*: a *special form*, a *lambda form*, a *macro form*, or a *function form*.

compound type specifier *n.* a *type specifier* that is a *cons*; *i.e.*, a *type specifier* that is not an *atomic type specifier*. "`(vector single-float)` is a compound type specifier."

concatenated stream *n.* an *input stream* of *type* **concatenated-stream**.

condition *n.* 1. an *object* which represents a *situation*—usually, but not necessarily, during *signaling*. 2. an *object* of *type* **condition**.

condition designator *n.* one or more *objects* that, taken together, denote either an existing *condition object* or a *condition object* to be implicitly created. For details, see Section 9.1.2.1 (Condition Designators).

condition handler *n.* a *function* that might be invoked by the act of *signaling*, that receives the *condition* being signaled as its only argument, and that is permitted to *handle* the *condition* or to *decline*. See Section 9.1.4.1 (Signaling).

condition reporter *n.* a *function* that describes how a *condition* is to be printed when the *Lisp printer* is invoked while ***print-escape*** is *false*. See Section 9.1.3 (Printing Conditions).

conditional newline *n.* a point in output where a *newline* might be inserted at the discretion of the *pretty printer*. There are four kinds of *conditional newlines*, called "linear-style," "fill-style," "miser-style," and "mandatory-style." See the *function* **pprint-newline** and Section 22.2.1.1 (Dynamic Control of the Arrangement of Output).

conformance *n.* a state achieved by proper and complete adherence to the requirements of this specification. See Section 1.5 (Conformance).

conforming code *n.* *code* that is all of part of a *conforming program*.

conforming implementation *n.* an *implementation*, used to emphasize complete and correct adherance to all conformance criteria. A *conforming implementation* is capable of accepting a *conforming program* as input, preparing that *program* for *execution*, and executing the prepared *program* in accordance with this specification. An *implementation* which has been extended may still be a *conforming implementation* provided that no extension interferes with the correct function of any *conforming program*.

conforming processor *n. ANSI* a *conforming implementation*.

conforming program *n.* a *program*, used to emphasize the fact that the *program* depends for its correctness only upon documented aspects of Common Lisp, and can therefore be expected to run correctly in any *conforming implementation*.

congruent *n.* conforming to the rules of *lambda list* congruency, as detailed in Section 7.6.4 (Congruent Lambda-lists for all Methods of a Generic Function).

cons *n.v.* 1. *n.* a compound data *object* having two components called the *car* and the *cdr*. 2. *v.* to create such an *object*. 3. *v. Idiom.* to create any *object*, or to allocate storage.

constant *n.* 1. a *constant form*. 2. a *constant variable*. 3. a *constant object*. 4. a *self-evaluating object*.

constant form *n.* any *form* for which *evaluation* always *yields* the same *value*, that neither affects nor is affected by the *environment* in which it is *evaluated* (except that it is permitted to refer to the names of *constant variables* defined in the *environment*), and that neither affects nor is affected by the state of any *object* except those *objects* that are *otherwise inaccessible parts* of *objects* created by the *form* itself. "A **car** form in which the argument is a **quote** form is a constant form."

constant object *n.* an *object* that is constrained (*e.g.*, by its context in a *program* or by the source from which it was obtained) to be *immutable*. "A literal object that has been processed by **compile-file** is a constant object."

constant variable *n.* a *variable*, the *value* of which can never change; that is, a *keyword*$_1$ or a *named constant*. "The symbols **t**, **nil**, `:direction`, and **most-positive-fixnum** are constant variables."

constituent *n.*, *adj.* 1. a. *n.* the *syntax type* of a *character* that is part of a *token*. For details, see Section 2.1.4.1 (Constituent Characters). b. *adj.* (of a *character*) having the *constituent*$_{1a}$ *syntax type*$_2$. c. *n.* a *constituent*$_{1b}$ *character*. 2. *n.* (of a *composite stream*) one of possibly several *objects* that collectively comprise the source or sink of that *stream*.

constituent trait *n.* (of a *character*) one of several classifications of a *constituent character* in a *readtable*. See Section 2.1.4.1 (Constituent Characters).

constructed stream *n.* a *stream* whose source or sink is a Lisp *object*. Note that since a *stream* is another Lisp *object*, *composite streams* are considered *constructed streams*. "A string stream is a constructed stream."

contagion *n.* a process whereby operations on *objects* of differing *types* (*e.g.*, arithmetic on mixed *types* of *numbers*) produce a result whose *type* is controlled by the dominance of one *argument*'s *type* over the *types* of the other *arguments*. See Section 12.1.1.2 (Contagion in Numeric Operations).

continuable *n.* (of an *error*) an *error* that is *correctable* by the `continue` restart.

control form *n.* 1. a *form* that establishes one or more places to which control can be transferred. 2. a *form* that transfers control.

copy *n.* 1. (of a *cons* C) a *fresh cons* with the *same car* and *cdr* as C. 2. (of a *list* L) a *fresh list* with the *same elements* as L. (Only the *list structure* is *fresh*; the *elements* are the *same*.) See the *function* **copy-list**. 3. (of an *association list* A with *elements* A_i) a *fresh list* B with *elements* B_i, each of which is **nil** if A_i is **nil**, or else a *copy* of the *cons* A_i. See the *function* **copy-alist**. 4. (of a *tree* T) a *fresh tree* with the *same leaves* as T. See the *function* **copy-tree**. 5. (of a *random state* R) a *fresh random state* that, if used as an argument to to the *function* **random** would produce the same series of "random" values as R would produce. 6. (of a *structure* S) a *fresh structure* that has the same *type* as S, and that has slot values, each of which is the *same* as the corresponding slot value of S. (Note that since the difference between a *cons*, a *list*, and a *tree* is a matter of "view" or "intention," there can be no general-purpose *function* which, based solely on the *type* of an *object*, can determine which of these distinct meanings is intended. The distinction rests solely on the basis of the text description within this document. For example, phrases like "a *copy* of the given *list*" or "copy of the *list* `x`" imply the second definition.)

correctable *adj.* (of an *error*) 1. (by a *restart* other than **abort** that has been associated with the *error*) capable of being corrected by invoking that *restart*. "The function **cerror** signals an error that is correctable by the **continue** *restart*." (Note that correctability is not a property of an *error object*, but rather a property of the *dynamic environment* that is in effect when the *error* is *signaled*. Specifically, the *restart* is "associated with" the *error condition object*. See Section 9.1.4.2.4 (Associating a Restart with a Condition).) 2. (when no specific *restart* is mentioned) *correctable*$_1$ by at least one *restart*. "**import** signals a correctable error of *type* **package-error** if any of the imported symbols has the same name as some distinct symbol already accessible in the package."

current input base *n.* (in a *dynamic environment*) the *radix* that is the *value* of ***read-base*** in that *environment*, and that is the default *radix* employed by the *Lisp reader* and its related *functions*.

current logical block *n.* the context of the innermost lexically enclosing use of **pprint-logical-block**.

current output base *n.* (in a *dynamic environment*) the *radix* that is the *value* of

print-base in that *environment*, and that is the default *radix* employed by the *Lisp printer* and its related *functions*.

current package *n.* (in a *dynamic environment*) the *package* that is the *value* of ***package*** in that *environment*, and that is the default *package* employed by the *Lisp reader* and *Lisp printer*, and their related *functions*.

current pprint dispatch table *n.* (in a *dynamic environment*) the *pprint dispatch table* that is the *value* of ***print-pprint-dispatch*** in that *environment*, and that is the default *pprint dispatch table* employed by the *pretty printer*.

current random state *n.* (in a *dynamic environment*) the *random state* that is the *value* of ***random-state*** in that *environment*, and that is the default *random state* employed by **random**.

current readtable *n.* (in a *dynamic environment*) the *readtable* that is the *value* of ***readtable*** in that *environment*, and that affects the way in which *expressions*$_2$ are parsed into *objects* by the *Lisp reader*.

D

data type *n. Trad.* a *type*.

debug I/O *n.* the *bidirectional stream* that is the *value* of the *variable* ***debug-io***.

debugger *n.* a facility that allows the *user* to handle a *condition* interactively. For example, the *debugger* might permit interactive selection of a *restart* from among the *active restarts*, and it might perform additional *implementation-defined* services for the purposes of debugging.

declaration *n.* a *global declaration* or *local declaration*.

declaration identifier *n.* one of the *symbols* **declaration**, **dynamic-extent**, **ftype**, **function**, **ignore**, **inline**, **notinline**, **optimize**, **special**, or **type**; or a *symbol* which is the *name* of a *type*; or a *symbol* which has been *declared* to be a *declaration identifier* by using a **declaration** *declaration*.

declaration specifier *n.* an *expression* that can appear at top level of a **declare** expression or a **declaim** form, or as the argument to **proclaim**, and which has a *car* which is a *declaration identifier*, and which has a *cdr* that is data interpreted according to rules specific to the *declaration identifier*.

declare *v.* to *establish* a *declaration*. See **declare**, **declaim**, or **proclaim**.

decline *v.* (of a *handler*) to return normally without having *handled* the *condition* being *signaled*, permitting the signaling process to continue as if the *handler* had not been present.

decoded time *n.* *absolute time*, represented as an ordered series of nine *objects* which, taken together, form a description of a point in calendar time, accurate to the nearest second (except that *leap seconds* are ignored). See Section 25.1.4.1 (Decoded Time).

default method *n.* a *method* having no *parameter specializers* other than the *class* **t**. Such a *method* is always an *applicable method* but might be *shadowed*$_2$ by a more specific *method*.

defaulted initialization argument list *n.* a *list* of alternating initialization argument *names* and *values* in which unsupplied initialization arguments are defaulted, used in the protocol for initializing and reinitializing *instances* of *classes*.

define-method-combination arguments lambda list *n.* a *lambda list* used by the `:arguments` option to **define-method-combination**. See Section 3.4.10 (Define-method-combination Arguments Lambda Lists).

define-modify-macro lambda list *n.* a *lambda list* used by **define-modify-macro**. See Section 3.4.9 (Define-modify-macro Lambda Lists).

defined name *n.* a *symbol* the meaning of which is defined by Common Lisp.

defining form *n.* a *form* that has the side-effect of *establishing* a definition. "**defun** and **defparameter** are defining forms."

defsetf lambda list *n.* a *lambda list* that is like an *ordinary lambda list* except that it does not permit **&aux** and that it permits use of **&environment**. See Section 3.4.7 (Defsetf Lambda Lists).

deftype lambda list *n.* a *lambda list* that is like a *macro lambda list* except that the default *value* for unsupplied *optional parameters* and *keyword parameters* is the *symbol* ***** (rather than **nil**). See Section 3.4.8 (Deftype Lambda Lists).

denormalized *adj.*, *ANSI*, *IEEE* (of a *float*) conforming to the description of "denormalized" as described by *IEEE Standard for Binary Floating-Point Arithmetic*. For example, in an *implementation* where the minimum possible exponent was `-7` but where `0.001` was a valid mantissa, the number `1.0e-10` might be representable as `0.001e-7` internally even if the *normalized* representation would call for it to be represented instead as `1.0e-10` or `0.1e-9`. By their nature, *denormalized floats* generally have less precision than *normalized floats*.

derived type *n.* a *type specifier* which is defined in terms of an expansion into another *type specifier*. **deftype** defines *derived types*, and there may be other *implementation-defined operators* which do so as well.

derived type specifier *n.* a *type specifier* for a *derived type*.

designator *n.* an *object* that denotes another *object*. In the dictionary entry for an *operator* if a *parameter* is described as a *designator* for a *type*, the description of the *operator* is written in a way that assumes that appropriate coercion to that *type* has already occurred; that is, that the *parameter* is already of the denoted *type*. For more detailed information, see Section 1.4.1.5 (Designators).

destructive *adj.* (of an *operator*) capable of modifying some program-visible aspect of one or more *objects* that are either explicit *arguments* to the *operator* or that can be obtained directly or indirectly from the *global environment* by the *operator*.

destructuring lambda list *n.* an *extended lambda list* used in **destructuring-bind** and nested within *macro lambda lists*. See Section 3.4.5 (Destructuring Lambda Lists).

different *adj.* not the *same* "The strings `"FOO"` and `"foo"` are different under **equal** but not under **equalp**."

digit *n.* (in a *radix*) a *character* that is among the possible digits (`0` to `9`, `A` to `Z`, and `a` to `z`) and that is defined to have an associated numeric weight as a digit in that *radix*. See Section 13.1.4.6 (Digits in a Radix).

dimension *n.* 1. a non-negative *integer* indicating the number of *objects* an *array* can hold along one axis. If the *array* is a *vector* with a *fill pointer*, the *fill pointer* is ignored. "The second dimension of that array is 7." 2. an axis of an array. "This array has six dimensions."

direct instance *n.* (of a *class* C) an *object* whose *class* is C itself, rather than some *subclass* of C. "The function **make-instance** always returns a direct instance of the class which is (or is named by) its first argument."

direct subclass *n.* (of a *class* C_1) a *class* C_2, such that C_1 is a *direct superclass* of C_2.

direct superclass *n.* (of a *class* C_1) a *class* C_2 which was explicitly designated as a *superclass* of C_1 in the definition of C_1.

disestablish *v.t.* to withdraw the *establishment* of an *object*, a *binding*, an *exit point*, a *tag*, a *handler*, a *restart*, or an *environment*.

disjoint *n.* (of *types*) having no *elements* in common.

dispatching macro character *n.* a *macro character* that has an associated table that specifies the *function* to be called for each *character* that is seen following the *dispatching macro character*. See the *function* **make-dispatch-macro-character**.

displaced array *n.* an *array* which has no storage of its own, but which is instead indirected to the storage of another *array*, called its *target*, at a specified offset, in such a way that any attempt to *access* the *displaced array* implicitly references the *target array*.

distinct *adj.* not *identical*.

documentation string *n.* (in a defining *form*) A *literal string* which because of the context in which it appears (rather than because of some intrinsically observable aspect of the *string*) is taken as documentation. In some cases, the *documentation string* is saved in such a way that it can later be obtained by supplying either an *object*, or by supplying a *name* and a "kind" to the *function* **documentation**. "The body of code in a **defmacro** form can be preceded by a documentation string of kind **function**."

dot *n.* the *standard character* that is variously called "full stop," "period," or "dot" (`.`). See Figure 2–5.

dotted list *n.* a *list* which has a terminating *atom* that is not **nil**. (An *atom* by itself is not a *dotted list*, however.)

dotted pair *n.* 1. a *cons* whose *cdr* is a *non-list*. 2. any *cons*, used to emphasize the use of the *cons* as a symmetric data pair.

double float *n.* an *object* of *type* **double-float**.

double-quote *n.* the *standard character* that is variously called "quotation mark" or "double quote" (`"`). See Figure 2–5.

dynamic binding *n.* a *binding* in a *dynamic environment*.

dynamic environment *n.* that part of an *environment* that contains *bindings* with *dynamic extent*. A *dynamic environment* contains, among other things: *exit points* established by **unwind-protect**, and *bindings* of *dynamic variables*, *exit points* established by **catch**, *condition handlers*, and *restarts*.

dynamic extent *n.* an *extent* whose duration is bounded by points of *establishment* and *disestablishment* within the execution of a particular *form*. See *indefinite extent*. "Dynamic variable bindings have dynamic extent."

dynamic scope *n.* *indefinite scope* along with *dynamic extent*.

dynamic variable *n.* a *variable* the *binding* for which is in the *dynamic environment*. See **special**.

E

echo stream *n.* a *stream* of *type* **echo-stream**.

effective method *n.* the combination of *applicable methods* that are executed when a *generic function* is invoked with a particular sequence of *arguments*.

element *n.* 1. (of a *list*) an *object* that is the *car* of one of the *conses* that comprise the *list*. 2. (of an *array*) an *object* that is stored in the *array*. 3. (of a *sequence*) an *object* that is an *element* of the *list* or *array* that is the *sequence*. 4. (of a *type*) an *object* that is a member of the set of *objects* designated by the *type*. 5. (of an *input stream*) a *character* or *number* (as appropriate to the *element type* of the *stream*) that is among the ordered series of *objects* that can be read from the *stream* (using **read-char** or **read-byte**, as appropriate to the *stream*). 6. (of an *output stream*) a *character* or *number* (as appropriate to the *element type* of the *stream*) that is among the ordered series of *objects* that has been or will be written to the *stream* (using **write-char** or **write-byte**, as appropriate to the *stream*). 7. (of a *class*) a *generalized instance* of the *class*.

element type *n.* 1. (of an *array*) the *array element type* of the *array*. 2. (of a *stream*) the *stream element type* of the *stream*.

em *n.* *Trad.* a context-dependent unit of measure commonly used in typesetting, equal to the displayed width of of a letter "M" in the current font. (The letter "M" is traditionally chosen because it is typically represented by the widest *glyph* in the font, and other characters' widths are typically fractions of an *em*. In implementations providing non-Roman characters with wider characters than "M," it is permissible for another character to be the *implementation-defined* reference character for this measure, and for "M" to be only a fraction of an *em* wide.) In a fixed width font, a line with *n* characters is *n ems* wide; in a variable width font, *n ems* is the expected upper bound on the width of such a line.

empty list *n.* the *list* containing no *elements*. See *()*.

empty type *n.* the *type* that contains no *elements*, and that is a *subtype* of all *types* (including itself). See *nil*.

end of file *n.* 1. the point in an *input stream* beyond which there is no further data. Whether or not there is such a point on an *interactive stream* is *implementation-defined*. 2. a *situation* that occurs upon an attempt to obtain data from an *input stream* that is at the *end of file*$_1$.

environment *n.* 1. a set of *bindings*. See Section 3.1.1 (Introduction to Environments). 2. an *environment object*. "**macroexpand** takes an optional environment argument."

environment object *n.* an *object* representing a set of *lexical bindings*, used in the processing of a *form* to provide meanings for *names* within that *form*. "**macroexpand** takes an optional environment argument." (The *object* **nil** when used as an *environment object* denotes the *null lexical environment*; the *values* of *environment parameters* to *macro functions* are *objects* of *implementation-dependent* nature which represent the *environment*$_1$ in which the corresponding *macro form* is to be expanded.) See Section 3.1.1.4 (Environment Objects).

environment parameter *n.* A *parameter* in a *defining form* f for which there is no corresponding *argument*; instead, this *parameter* receives as its value an *environment object* which corresponds to the *lexical environment* in which the *defining form* f appeared.

error *n.* 1. (only in the phrase "is an error") a *situation* in which the semantics of a program are not specified, and in which the consequences are undefined. 2. a *condition* which represents an *error situation*. See Section 1.4.2 (Error Terminology). 3. an *object* of *type* **error**.

error output *n.* the *output stream* which is the *value* of the *dynamic variable* ***error-output***.

escape *n.*, *adj.* 1. *n.* a *single escape* or a *multiple escape*. 2. *adj. single escape* or *multiple escape*.

establish *v.t.* to build or bring into being a *binding*, a *declaration*, an *exit point*, a *tag*, a *handler*, a *restart*, or an *environment*. "**let** establishes lexical bindings."

evaluate *v.t.* (a *form* or an *implicit progn*) to *execute* the *code* represented by the *form* (or the series of *forms* making up the *implicit progn*) by applying the rules of *evaluation*, returning zero or more values.

evaluation *n.* a model whereby *forms* are *executed*, returning zero or more values. Such execution might be implemented directly in one step by an interpreter or in two steps by first *compiling* the *form* and then *executing* the *compiled code*; this choice is dependent both on context and the nature of the *implementation*, but in any case is not in general detectable by any program. The evaluation model is designed in such a way that a *conforming implementation* might legitimately have only a compiler and no interpreter, or vice versa. See Section 3.1.2 (The Evaluation Model).

evaluation environment *n.* a *run-time environment* in which macro expanders and code specified by **eval-when** to be evaluated are evaluated. All evaluations initiated by the *compiler* take place in the *evaluation environment*.

execute *v.t. Trad.* (*code*) to perform the imperative actions represented by the *code*.

execution time *n.* the duration of time that *compiled code* is being *executed*.

exhaustive partition *n.* (of a *type*) a set of *pairwise disjoint types* that form an *exhaustive union*.

exhaustive union *n.* (of a *type*) a set of *subtypes* of the *type*, whose union contains all *elements* of that *type*.

exit point *n.* a point in a *control form* from which (*e.g.*, **block**), through which (*e.g.*, **unwind-protect**), or to which (*e.g.*, **tagbody**) control and possibly *values* can be transferred both actively by using another *control form* and passively through the normal control and data flow of *evaluation*. "**catch** and **block** establish bindings for exit points to which **throw** and **return-from**, respectively, can transfer control and values; **tagbody** establishes a binding for an exit point with lexical extent to which **go** can transfer control; and **unwind-protect** establishes an exit point through which control might be transferred by operators such as **throw**, **return-from**, and **go**."

explicit return *n.* the act of transferring control (and possibly *values*) to a *block* by using **return-from** (or **return**).

explicit use *n.* (of a *variable* V in a *form* F) a reference to V that is directly apparent in the normal semantics of F; *i.e.*, that does not expose any undocumented details of the *macro expansion* of the *form* itself. References to V exposed by expanding *subforms* of F are, however, considered to be *explicit uses* of V.

exponent marker *n.* a character that is used in the textual notation for a *float* to separate the mantissa from the exponent. The characters defined as *exponent markers* in the *standard readtable* are shown in Figure 26–1. For more information, see Section 2.1 (Character Syntax). "The exponent marker 'd' in '3.0d7' indicates that this number is to be represented as a double float."

Marker	Meaning
`D` or `d`	**double-float**
`E` or `e`	**float** (see ***read-default-float-format***)
`F` or `f`	**single-float**
`L` or `l`	**long-float**
`S` or `s`	**short-float**

Figure 26–1. Exponent Markers

export *v.t.* (a *symbol* in a *package*) to add the *symbol* to the list of *external symbols* of the *package*.

exported *adj.* (of a *symbol* in a *package*) being an *external symbol* of the *package*.

expressed adjustability *n.* (of an *array*) a *generalized boolean* that is conceptually (but not necessarily actually) associated with the *array*, representing whether the *array* is *expressly adjustable*. See also *actual adjustability*.

expressed array element type *n.* (of an *array*) the *type* which is the *array element type* implied by a *type declaration* for the *array*, or which is the requested *array element type* at its time of creation, prior to any selection of an *upgraded array element type*. (Common Lisp does not provide a way of detecting this *type* directly at run time, but an *implementation* is permitted to make assumptions about the *array*'s contents and the operations which may be performed on the *array* when this *type* is noted during code analysis, even if those assumptions would not be valid in general for the *upgraded array element type* of the *expressed array element type*.)

expressed complex part type *n.* (of a *complex*) the *type* which is implied as the *complex part type* by a *type declaration* for the *complex*, or which is the requested *complex part type* at its time of creation, prior to any selection of an *upgraded complex part type*. (Common Lisp does not provide a way of detecting this *type* directly at run time, but an *implementation* is permitted to make assumptions about the operations which may be performed on the *complex* when this *type* is noted during code analysis, even if those assumptions would not be valid in general for the *upgraded complex part type* of the *expressed complex part type*.)

expression *n.* 1. an *object*, often used to emphasize the use of the *object* to encode or represent information in a specialized format, such as program text. "The second expression in a **let** form is a list of bindings." 2. the textual notation used to notate an *object* in a source file. "The expression `'sample` is equivalent to `(quote sample)`."

expressly adjustable *adj.* (of an *array*) being *actually adjustable* by virtue of an explicit request for this characteristic having been made at the time of its creation. All *arrays* that are *expressly adjustable* are *actually adjustable*, but not necessarily vice versa.

extended character *n.* a *character* of *type* **extended-char**: a *character* that is not a *base character*.

extended function designator *n.* a *designator* for a *function*; that is, an *object* that denotes a *function* and that is one of: a *function name* (denoting the *function* it names in the *global environment*), or a *function* (denoting itself). The consequences are undefined if a *function name* is used as an *extended function designator* but it does not have a global definition as a *function*, or if it is a *symbol* that has a global definition as a *macro* or a *special form*. See also *function designator*.

extended lambda list *n.* a list resembling an *ordinary lambda list* in form and purpose, but offering additional syntax or functionality not available in an *ordinary lambda list*. "**defmacro** uses extended lambda lists."

extension *n.* a facility in an *implementation* of Common Lisp that is not specified by this standard.

extent *n.* the interval of time during which a *reference* to an *object*, a *binding*, an *exit point*, a *tag*, a *handler*, a *restart*, or an *environment* is defined.

external file format *n.* an *object* of *implementation-dependent* nature which determines one of possibly several *implementation-dependent* ways in which *characters* are encoded externally in a *character file*.

external file format designator *n.* a *designator* for an *external file format*; that is, an *object* that denotes an *external file format* and that is one of: the *symbol* `:default` (denoting an *implementation-dependent* default *external file format* that can accomodate at least the *base characters*), some other *object* defined by the *implementation* to be an *external file format designator* (denoting an *implementation-defined external file format*), or some other *object* defined by the *implementation* to be an *external file format* (denoting itself).

external symbol *n.* (of a *package*) a *symbol* that is part of the 'external interface' to the *package* and that are *inherited*$_3$ by any other *package* that *uses* the *package*. When using the *Lisp reader*, if a *package prefix* is used, the *name* of an *external symbol* is separated from the *package name* by a single *package marker* while the *name* of an *internal symbol* is separated from the *package name* by a double *package marker*; see Section 2.3.4 (Symbols as Tokens).

externalizable object *n.* an *object* that can be used as a *literal object* in *code* to be processed by the *file compiler*.

F

false *n.* the *symbol* **nil**, used to represent the failure of a *predicate* test.

fbound [ˈefˌbau̲nd] *adj.* (of a *function name*) *bound* in the *function namespace*. (The *names* of *macros* and *special operators* are *fbound*, but the nature and *type* of the *object* which is their *value* is *implementation-dependent*. Further, defining a *setf expander* *F* does not cause the *setf function* `(setf F)` to become defined; as such, if there is a such a definition of a *setf expander* *F*, the *function* `(setf F)` can be *fbound* if and only if, by design or coincidence, a function binding for `(setf F)` has been independently established.) See the *functions* **fboundp** and **symbol-function**.

feature *n.* 1. an aspect or attribute of Common Lisp, of the *implementation*, or of the *environment*. 2. a *symbol* that names a *feature*$_1$. See Section 24.1.2 (Features). "The `:ansi-cl` feature is present in all conforming implementations."

feature expression *n.* A boolean combination of *features* used by the `#+` and `#-` *reader macros* in order to direct conditional *reading* of *expressions* by the *Lisp reader*. See Section 24.1.2.1 (Feature Expressions).

features list *n.* the *list* that is the *value* of ***features***.

file *n.* a named entry in a *file system*, having an *implementation-defined* nature.

file compiler *n.* any *compiler* which *compiles source code* contained in a *file*, producing a *compiled file* as output. The **compile-file** function is the only interface to such a *compiler* provided by Common Lisp, but there might be other, *implementation-defined* mechanisms for invoking the *file compiler*.

file position *n.* (in a *stream*) a non-negative *integer* that represents a position in the *stream*. Not all *streams* are able to represent the notion of *file position*; in the description of any *operator* which manipulates *file positions*, the behavior for *streams* that don't have this notion must be explicitly stated. For *binary streams*, the *file position* represents the number of preceding *bytes* in the *stream*. For *character streams*, the constraint is more relaxed: *file positions* must increase monotonically, the amount of the increase between *file positions* corresponding to any two successive characters in the *stream* is *implementation-dependent*.

file position designator *n.* (in a *stream*) a *designator* for a *file position* in that *stream*; that is, the symbol `:start` (denoting `0`, the first *file position* in that *stream*), the symbol `:end` (denoting the last *file position* in that *stream*; *i.e.*, the position following the last *element* of the *stream*), or a *file position* (denoting itself).

file stream *n.* an *object* of *type* **file-stream**.

file system *n.* a facility which permits aggregations of data to be stored in named *files* on some medium that is external to the *Lisp image* and that therefore persists from *session* to *session*.

filename *n.* a handle, not necessarily ever directly represented as an *object*, that can be used to refer to a *file* in a *file system*. *Pathnames* and *namestrings* are two kinds of *objects* that substitute for *filenames* in Common Lisp.

fill pointer *n.* (of a *vector*) an *integer* associated with a *vector* that represents the index above which no *elements* are *active*. (A *fill pointer* is a non-negative *integer* no larger than the total number of *elements* in the *vector*. Not all *vectors* have *fill pointers*.)

finite *adj.* (of a *type*) having a finite number of *elements*. "The type specifier `(integer 0 5)` denotes a finite type, but the type specifiers **integer** and `(integer 0)` do not."

fixnum *n.* an *integer* of *type* **fixnum**.

float *n.* an *object* of *type* **float**.

for-value *adj.* (of a *reference* to a *binding*) being a *reference* that $reads_1$ the *value* of the *binding*.

form *n.* 1. any *object* meant to be *evaluated*. 2. a *symbol*, a *compound form*, or a *self-evaluating object*. 3. (for an *operator*, as in "⟪*operator*⟫ *form*") a *compound form* having that *operator* as its first element. "A **quote** form is a constant form."

formal argument *n. Trad.* a *parameter*.

formal parameter *n. Trad.* a *parameter*.

format *v.t.* (a *format control* and *format arguments*) to perform output as if by **format**, using the *format string* and *format arguments*.

format argument *n.* an *object* which is used as data by functions such as **format** which interpret *format controls*.

format control *n.* a *format string*, or a *function* that obeys the *argument* conventions for a *function* returned by the **formatter** *macro*. See Section 22.2.1.3 (Compiling Format Strings).

format directive *n.* 1. a sequence of *characters* in a *format string* which is introduced by a *tilde*, and which is specially interpreted by *code* which processes *format strings* to mean that some special operation should be performed, possibly involving data supplied by the *format arguments* that accompanied the *format string*. See the *function* **format**. "In `"~D base 10 = ~8R"`, the character sequences '`~D`' and '`~8R`' are format directives." 2. the conceptual category of all $\textit{format directives}_1$ which use the same dispatch character. "Both `"~3d"` and `"~3,'0D"` are valid uses of the '`~D`' format directive."

format string *n.* a *string* which can contain both ordinary text and *format directives*, and which is used in conjunction with *format arguments* to describe how text output should be formatted by certain functions, such as **format**.

free declaration *n.* a declaration that is not a *bound declaration*. See **declare**.

fresh *adj.* 1. (of an *object yielded* by a *function*) having been newly-allocated by that *function*. (The caller of a *function* that returns a *fresh object* may freely modify the *object* without fear that such modification will compromise the future correct behavior of that *function*.) 2. (of a *binding* for a *name*) newly-allocated; not shared with other *bindings* for that *name*.

freshline *n.* a conceptual operation on a *stream*, implemented by the *function* **fresh-line** and by the *format directive* `~&`, which advances the display position to the beginning of the next line (as if a *newline* had been typed, or the *function* **terpri** had been called) unless the *stream* is already known to be positioned at the beginning of a line. Unlike *newline*, *freshline* is not a *character*.

funbound [ˈ**efunbaund**] *n.* (of a *function name*) not *fbound*.

function *n.* 1. an *object* representing code, which can be *called* with zero or more *arguments*, and which produces zero or more *values*. 2. an *object* of *type* **function**.

function block name *n.* (of a *function name*) The *symbol* that would be used as the name of an *implicit block* which surrounds the body of a *function* having that *function name*. If the *function name* is a *symbol*, its *function block name* is the *function name* itself. If the *function name* is a *list* whose *car* is **setf** and whose *cadr* is a *symbol*, its *function block name* is the *symbol* that is the *cadr* of the *function name*. An *implementation* which supports additional kinds of *function names* must specify for each how the corresponding *function block name* is computed.

function cell *n. Trad.* (of a *symbol*) The *place* which holds the *definition* of the global *function binding*, if any, named by that *symbol*, and which is *accessed* by **symbol-function**. See *cell*.

function designator *n.* a *designator* for a *function*; that is, an *object* that denotes a *function* and that is one of: a *symbol* (denoting the *function* named by that *symbol* in the *global environment*), or a *function* (denoting itself). The consequences are undefined if a *symbol* is used as a *function designator* but it does not have a global definition as a *function*, or it has a global definition as a *macro* or a *special form*. See also *extended function designator*.

function form *n.* a *form* that is a *list* and that has a first element which is the *name* of a *function* to be called on *arguments* which are the result of *evaluating* subsequent elements of the *function form*.

function name *n.* 1. (in an *environment*) A *symbol* or a *list* `(setf` *symbol*`)` that is the *name* of a *function* in that *environment*. 2. A *symbol* or a *list* `(setf` *symbol*`)`.

functional evaluation *n.* the process of extracting a *functional value* from a *function name* or a *lambda expression*. The evaluator performs *functional evaluation* implicitly when it encounters a *function name* or a *lambda expression* in the *car* of a *compound form*, or explicitly when it encounters a **function** *special form*. Neither a use of a *symbol* as a *function designator* nor a use of the *function* **symbol-function** to extract the *functional value* of a *symbol* is considered a *functional evaluation*.

functional value *n.* 1. (of a *function name* N in an *environment* E) The *value* of the *binding* named N in the *function namespace* for *environment* E; that is, the contents of the *function cell* named *N* in *environment E*. 2. (of an *fbound symbol S*) the contents of the *symbol*'s *function cell*; that is, the *value* of the *binding* named *S* in the *function namespace* of the *global environment*. (A *name* that is a *macro name* in the *global environment* or is a *special operator* might or might not be *fbound*. But if *S* is such a *name* and is *fbound*, the specific nature of its *functional value* is *implementation-dependent*; in particular, it might or might not be a *function*.)

further compilation *n. implementation-dependent* compilation beyond *minimal compilation*. Further compilation is permitted to take place at *run time*. "Block compilation and generation of machine-specific instructions are examples of further compilation."

G

general *adj.* (of an *array*) having *element type* **t**, and consequently able to have any *object* as an *element*.

generalized boolean *n.* an *object* used as a truth value, where the symbol **nil** represents *false* and all other *objects* represent *true*. See *boolean*.

generalized instance *n.* (of a *class*) an *object* the *class* of which is either that *class* itself, or some subclass of that *class*. (Because of the correspondence between types and classes, the term "generalized instance of *X*" implies "object of type *X*" and in cases where *X* is a *class* (or *class name*) the reverse is also true. The former terminology emphasizes the view of *X* as a *class* while the latter emphasizes the view of *X* as a *type specifier*.)

generalized reference *n.* a reference to a location storing an *object* as if to a *variable*. (Such a reference can be either to *read* or *write* the location.) See Section 5.1 (Generalized Reference). See also *place*.

generalized synonym stream *n.* (with a *synonym stream symbol*) 1. (to a *stream*) a *synonym stream* to the *stream*, or a *composite stream* which has as a target a *generalized synonym stream* to the *stream*. 2. (to a *symbol*) a *synonym stream* to the *symbol*, or a *composite stream* which has as a target a *generalized synonym stream* to the *symbol*.

generic function *n.* a *function* whose behavior depends on the *classes* or identities of the arguments supplied to it and whose parts include, among other things, a set of *methods*, a *lambda list*, and a *method combination* type.

generic function lambda list *n.* A *lambda list* that is used to describe data flow into a *generic function*. See Section 3.4.2 (Generic Function Lambda Lists).

gensym *n. Trad.* an *uninterned symbol*. See the *function* **gensym**.

global declaration *n.* a *form* that makes certain kinds of information about code globally available; that is, a **proclaim** *form* or a **declaim** *form*.

global environment *n.* that part of an *environment* that contains *bindings* with *indefinite scope* and *indefinite extent*.

global variable *n.* a *dynamic variable* or a *constant variable*.

glyph *n.* a visual representation. "Graphic characters have associated glyphs."

go *v.* to transfer control to a *go point*. See the *special operator* **go**.

go point one of possibly several *exit points* that are *established* by **tagbody** (or other abstractions, such as **prog**, which are built from **tagbody**).

go tag *n.* the *symbol* or *integer* that, within the *lexical scope* of a **tagbody** *form*, names an *exit point established* by that **tagbody** *form*.

graphic *adj.* (of a *character*) being a "printing" or "displayable" *character* that has a standard visual representation as a single *glyph*, such as `A` or `*` or `=`. *Space* is defined to be *graphic*. Of the *standard characters*, all but *newline* are *graphic*. See *non-graphic*.

H

handle *v.* (of a *condition* being *signaled*) to perform a non-local transfer of control, terminating the ongoing *signaling* of the *condition*.

handler *n.* a *condition handler*.

hash table *n.* an *object* of *type* **hash-table**, which provides a mapping from *keys* to *values*.

home package *n.* (of a *symbol*) the *package*, if any, which is contents of the *package cell* of the *symbol*, and which dictates how the *Lisp printer* prints the *symbol* when it is not *accessible* in the *current package*. (*Symbols* which have **nil** in their *package cell* are said to have no *home package*, and also to be *apparently uninterned*.)

I

I/O customization variable *n.* one of the *stream variables* in Figure 26–2, or some other (*implementation-defined*) *stream variable* that is defined by the *implementation* to be an *I/O customization variable*.

debug-io	*error-io*	query-io*
standard-input	*standard-output*	*trace-output*

Figure 26–2. Standardized I/O Customization Variables

identical *adj.* the *same* under **eq**.

identifier *n.* 1. a *symbol* used to identify or to distinguish *names*. 2. a *string* used the same way.

immutable *adj.* not subject to change, either because no *operator* is provided which is capable of effecting such change or because some constraint exists which prohibits the use of an *operator* that might otherwise be capable of effecting such a change. Except as explicitly indicated otherwise, *implementations* are not required to detect attempts to modify *immutable objects* or *cells*; the consequences of attempting to make such modification are undefined. "Numbers are immutable."

implementation *n.* a system, mechanism, or body of *code* that implements the semantics of Common Lisp.

implementation limit *n.* a restriction imposed by an *implementation*.

implementation-defined *adj. implementation-dependent*, but required by this specification to be defined by each *conforming implementation* and to be documented by the corresponding implementor.

implementation-dependent *adj.* describing a behavior or aspect of Common Lisp which has been deliberately left unspecified, that might be defined in some *conforming implementations* but not in others, and whose details may differ between *implementations*. A *conforming implementation* is encouraged (but not required) to document its treatment of each item in this specification which is marked *implementation-dependent*, although in some cases such documentation might simply identify the item as "undefined."

implementation-independent *adj.* used to identify or emphasize a behavior or aspect of Common Lisp which does not vary between *conforming implementations*.

implicit block *n.* a *block* introduced by a *macro form* rather than by an explicit **block** *form*.

implicit compilation *n. compilation* performed during *evaluation*.

implicit progn *n.* an ordered set of adjacent *forms* appearing in another *form*, and defined by their context in that *form* to be executed as if within a **progn**.

implicit tagbody *n.* an ordered set of adjacent *forms* and/or *tags* appearing in another *form*, and defined by their context in that *form* to be executed as if within a **tagbody**.

import *v.t.* (a *symbol* into a *package*) to make the *symbol* be *present* in the *package*.

improper list *n.* a *list* which is not a *proper list*: a *circular list* or a *dotted list*.

inaccessible *adj.* not *accessible*.

indefinite extent *n.* an *extent* whose duration is unlimited. "Most Common Lisp objects have indefinite extent."

indefinite scope *n.* *scope* that is unlimited.

indicator *n.* a *property indicator*.

indirect instance *n.* (of a *class* C_1) an *object* of *class* C_2, where C_2 is a *subclass* of C_1. "An integer is an indirect instance of the class **number**."

inherit *v.t.* 1. to receive or acquire a quality, trait, or characteristic; to gain access to a feature defined elsewhere. 2. (a *class*) to acquire the structure and behavior defined by a *superclass*. 3. (a *package*) to make *symbols exported* by another *package accessible* by using **use-package**.

initial pprint dispatch table *n.* the *value* of ***print-pprint-dispatch*** at the time the *Lisp image* is started.

initial readtable *n.* the *value* of ***readtable*** at the time the *Lisp image* is started.

initialization argument list *n.* a *property list* of initialization argument *names* and *values* used in the protocol for initializing and reinitializing *instances* of *classes*. See Section 7.1 (Object Creation and Initialization).

initialization form *n.* a *form* used to supply the initial *value* for a *slot* or *variable*. "The initialization form for a slot in a **defclass** form is introduced by the keyword `:initform`."

input *adj.* (of a *stream*) supporting input operations (*i.e.*, being a "data source"). An *input stream* might also be an *output stream*, in which case it is sometimes called a *bidirectional stream*. See the *function* **input-stream-p**.

instance *n.* 1. a *direct instance*. 2. a *generalized instance*. 3. an *indirect instance*.

integer *n.* an *object* of *type* **integer**, which represents a mathematical integer.

interactive stream *n.* a *stream* on which it makes sense to perform interactive querying. See Section 21.1.1.1.3 (Interactive Streams).

intern *v.t.* 1. (a *string* in a *package*) to look up the *string* in the *package*, returning either a *symbol* with that *name* which was already *accessible* in the *package* or a newly created *internal symbol* of the *package* with that *name*. 2. *Idiom.* generally, to observe a protocol whereby objects which are equivalent or have equivalent names under some predicate defined by the protocol are mapped to a single canonical object.

internal symbol *n.* (of a *package*) a symbol which is *accessible* in the *package*, but which is not an *external symbol* of the *package*.

internal time *n.* *time*, represented as an *integer* number of *internal time units*. *Absolute internal time* is measured as an offset from an arbitrarily chosen, *implementation-dependent* base. See Section 25.1.4.3 (Internal Time).

internal time unit *n.* a unit of time equal to $1/n$ of a second, for some *implementation-defined integer* value of n. See the *variable* **internal-time-units-per-second**.

interned *adj. Trad.* 1. (of a *symbol*) *accessible*$_3$ in any *package*. 2. (of a *symbol* in a specific *package*) *present* in that *package*.

interpreted function *n.* a *function* that is not a *compiled function*. (It is possible for there to be a *conforming implementation* which has no *interpreted functions*, but a *conforming program* must not assume that all *functions* are *compiled functions*.)

interpreted implementation *n.* an *implementation* that uses an execution strategy for *interpreted functions* that does not involve a one-time semantic analysis pre-pass, and instead uses "lazy" (and sometimes repetitious) semantic analysis of *forms* as they are encountered during execution.

interval designator *n.* (of *type* T) an ordered pair of *objects* that describe a *subtype* of T by delimiting an interval on the real number line. See Section 12.1.6 (Interval Designators).

invalid *n.*, *adj.* 1. *n.* a possible *constituent trait* of a *character* which if present signifies that the *character* cannot ever appear in a *token* except under the control of a *single escape character*. For details, see Section 2.1.4.1 (Constituent Characters). 2. *adj.* (of a *character*) being a *character* that has *syntax type constituent* in the *current readtable* and that has the *constituent trait invalid*$_1$. See Figure 2–8.

iteration form *n.* a *compound form* whose *operator* is named in Figure 26–3, or a *compound form* that has an *implementation-defined operator* and that is defined by the *implementation* to be an *iteration form*.

do	**do-external-symbols**	**dotimes**
do*	**do-symbols**	**loop**
do-all-symbols	**dolist**	

Figure 26–3. Standardized Iteration Forms

iteration variable *n.* a *variable* V, the *binding* for which was created by an *explicit use* of V in an *iteration form*.

K

key *n.* an *object* used for selection during retrieval. See *association list*, *property list*, and *hash table*. Also, see Section 17.1 (Sequence Concepts).

keyword *n.* 1. a *symbol* the *home package* of which is the `KEYWORD` *package*. 2. any *symbol*, usually but not necessarily in the `KEYWORD` *package*, that is used as an identifying marker in keyword-style argument passing. See **lambda**. 3. *Idiom.* a *lambda list keyword*.

keyword parameter *n.* A *parameter* for which a corresponding keyword *argument* is optional. (There is no such thing as a required keyword *argument*.) If the *argument* is not supplied, a default value is used. See also *supplied-p parameter*.

keyword/value pair *n.* two successive *elements* (a *keyword* and a *value*, respectively) of a *property list*.

L

lambda combination *n.* *Trad.* a *lambda form*.

lambda expression *n.* a *list* which 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*; its name derives from the fact that its first element is the *symbol* `lambda`. See **lambda**.

lambda form *n.* a *form* that is a *list* and that has a first element which is a *lambda expression* representing a *function* to be called on *arguments* which are the result of *evaluating* subsequent elements of the *lambda form*.

lambda list *n.* a *list* that specifies a set of *parameters* (sometimes called *lambda variables*) and a protocol for receiving *values* for those *parameters*; that is, an *ordinary lambda list*, an *extended lambda list*, or a *modified lambda list*.

lambda list keyword *n.* a *symbol* whose *name* begins with *ampersand* and that is specially recognized in a *lambda list*. Note that no *standardized lambda list keyword* is in the `KEYWORD` *package*.

lambda variable *n.* a *formal parameter*, used to emphasize the *variable*'s relation to the *lambda list* that *established* it.

leaf *n.* 1. an *atom* in a *tree*$_1$. 2. a terminal node of a *tree*$_2$.

leap seconds *n.* additional one-second intervals of time that are occasionally inserted into the true calendar by official timekeepers as a correction similar to "leap years." All Common Lisp *time* representations ignore *leap seconds*; every day is assumed to be exactly 86400 seconds long.

left-parenthesis *n.* the *standard character* "`(`", that is variously called "left parenthesis" or "open parenthesis" See Figure 2–5.

length *n.* (of a *sequence*) the number of *elements* in the *sequence*. (Note that if the *sequence* is a *vector* with a *fill pointer*, its *length* is the same as the *fill pointer* even though the total allocated size of the *vector* might be larger.)

lexical binding *n.* a *binding* in a *lexical environment*.

lexical closure *n.* a *function* that, when invoked on *arguments*, executes the body of a *lambda expression* in the *lexical environment* that was captured at the time of the creation of the *lexical closure*, augmented by *bindings* of the *function*'s *parameters* to the corresponding *arguments*.

lexical environment *n.* that part of the *environment* that contains *bindings* whose names have *lexical scope*. A *lexical environment* contains, among other things: ordinary *bindings* of *variable names* to *values*, lexically *established bindings* of *function names* to *functions*, *macros*, *symbol macros*, *blocks*, *tags*, and *local declarations* (see **declare**).

lexical scope *n.* *scope* that is limited to a spatial or textual region within the establishing *form*. "The names of parameters to a function normally are lexically scoped."

lexical variable *n.* a *variable* the *binding* for which is in the *lexical environment*.

Lisp image *n.* a running instantiation of a Common Lisp *implementation*. A *Lisp image* is characterized by a single address space in which any *object* can directly refer to any another in conformance with this specification, and by a single, common, *global environment*. (External operating systems sometimes call this a "core image," "fork," "incarnation," "job," or "process." Note however, that the issue of a "process" in such an operating system is technically orthogonal to the issue of a *Lisp image* being defined here. Depending on the operating system, a single "process" might have multiple *Lisp images*, and multiple "processes" might reside in a single *Lisp image*. Hence, it is the idea of a fully shared address space for direct reference among all *objects* which is the defining characteristic. Note, too, that two "processes" which have a communication area that permits the sharing of some but not all *objects* are considered to be distinct *Lisp images*.)

Lisp printer *n. Trad.* the procedure that prints the character representation of an *object* onto a *stream*. (This procedure is implemented by the *function* **write**.)

Lisp read-eval-print loop *n. Trad.* an endless loop that *reads*$_2$ a *form*, *evaluates* it, and prints (*i.e.*, *writes*$_2$) the results. In many *implementations*, the default mode of interaction with Common Lisp during program development is through such a loop.

Lisp reader *n. Trad.* the procedure that parses character representations of *objects* from a *stream*, producing *objects*. (This procedure is implemented by the *function* **read**.)

list *n.* 1. 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*. See also *proper list*, *dotted list*, or *circular list*. 2. the *type* that is the union of **null** and **cons**.

list designator *n.* a *designator* for a *list* of *objects*; that is, an *object* that denotes a *list* and that is one of: a *non-nil atom* (denoting a *singleton list* whose *element* is that *non-nil atom*) or a *proper list* (denoting itself).

list structure *n.* (of a *list*) the set of *conses* that make up the *list*. Note that while the *car*$_{1b}$ component of each such *cons* is part of the *list structure*, the *objects* that are *elements* of the *list* (*i.e.*, the *objects* that are the *cars*$_2$ of each *cons* in the *list*) are not themselves part of its *list structure*, even if they are *conses*, except in the (*circular*$_2$) case where the *list* actually contains one of its *tails* as an *element*. (The *list structure* of a *list* is sometimes redundantly referred to as its "top-level list structure" in order to emphasize that any *conses* that are *elements* of the *list* are not involved.)

literal *adj.* (of an *object*) referenced directly in a program rather than being computed by the program; that is, appearing as data in a **quote** *form*, or, if the *object* is a *self-evaluating object*, appearing as unquoted data. "In the form `(cons "one" '("two"))`, the expressions `"one"`, `("two")`, and `"two"` are literal objects."

load *v.t.* (a *file*) to cause the *code* contained in the *file* to be *executed*. See the *function* **load**.

load time *n.* the duration of time that the loader is *loading compiled code*.

load time value *n.* an *object* referred to in *code* by a **load-time-value** *form*. The *value* of such a *form* is some specific *object* which can only be computed in the run-time *environment*. In the case of *file compilation*, the *value* is computed once as part of the process of *loading* the *compiled file*, and not again. See the *special operator* **load-time-value**.

loader *n.* a facility that is part of Lisp and that *loads* a *file*. See the *function* **load**.

local declaration *n.* an *expression* which may appear only in specially designated positions of certain *forms*, and which provides information about the code contained within the containing *form*; that is, a **declare** *expression*.

local precedence order *n.* (of a *class*) a *list* consisting of the *class* followed by its *direct superclasses* in the order mentioned in the defining *form* for the *class*.

local slot *n.* (of a *class*) a *slot accessible* in only one *instance*, namely the *instance* in which the *slot* is allocated.

logical block *n.* a conceptual grouping of related output used by the *pretty printer*. See the *macro* **pprint-logical-block** and Section 22.2.1.1 (Dynamic Control of the Arrangement of Output).

logical host *n.* an *object* of *implementation-dependent* nature that is used as the representation of a "host" in a *logical pathname*, and that has an associated set of translation rules for converting *logical pathnames* belonging to that host into *physical pathnames*. See Section 19.3 (Logical Pathnames).

logical host designator *n.* a *designator* for a *logical host*; that is, an *object* that denotes a *logical host* and that is one of: a *string* (denoting the *logical host* that it names), or a *logical host* (denoting itself). (Note that because the representation of a *logical host* is *implementation-dependent*, it is possible that an *implementation* might represent a *logical host* as the *string* that names it.)

logical pathname *n.* an *object* of *type* **logical-pathname**.

long float *n.* an *object* of *type* **long-float**.

loop keyword *n. Trad.* a symbol that is a specially recognized part of the syntax of an extended **loop** *form*. Such symbols are recognized by their *name* (using **string=**), not by their identity; as such, they may be in any package. A *loop keyword* is not a *keyword*.

lowercase *adj.* (of a *character*) being among *standard characters* corresponding to the small letters `a` through `z`, or being some other *implementation-defined character* that is defined by the *implementation* to be *lowercase*. See Section 13.1.4.3 (Characters With Case).

M

macro *n.* 1. a *macro form* 2. a *macro function*. 3. a *macro name*.

macro character *n.* a *character* which, when encountered by the *Lisp reader* in its main dispatch loop, introduces a *reader* $macro_1$. (*Macro characters* have nothing to do with *macros*.)

macro expansion *n.* 1. the process of translating a *macro form* into another *form*. 2. the *form* resulting from this process.

macro form *n.* a *form* that stands for another *form* (*e.g.*, for the purposes of abstraction, information hiding, or syntactic convenience); that is, either a *compound form* whose first element is a *macro name*, or a *form* that is a *symbol* that names a *symbol macro*.

macro function *n.* a *function* of two arguments, a *form* and an *environment*, that implements *macro expansion* by producing a *form* to be evaluated in place of the original argument *form*.

macro lambda list *n.* an *extended lambda list* used in *forms* that *establish macro* definitions, such as **defmacro** and **macrolet**. See Section 3.4.4 (Macro Lambda Lists).

macro name *n.* a *name* for which **macro-function** returns *true* and which when used as the first element of a *compound form* identifies that *form* as a *macro form*.

macroexpand hook *n.* the *function* that is the *value* of ***macroexpand-hook***.

mapping *n.* 1. a type of iteration in which a *function* is successively applied to *objects* taken from corresponding entries in collections such as *sequences* or *hash tables*. 2. *Math.* a relation between two sets in which each element of the first set (the "domain") is assigned one element of the second set (the "range").

metaclass *n.* 1. a *class* whose instances are *classes*. 2. (of an *object*) the *class* of the *class* of the *object*.

Metaobject Protocol *n.* one of many possible descriptions of how a *conforming implementation* might implement various aspects of the object system. This description is beyond the scope of this document, and no *conforming implementation* is required to adhere to it except as noted explicitly in this specification. Nevertheless, its existence helps to establish normative practice, and implementors with no reason to diverge from it are encouraged to consider making their *implementation* adhere to it where possible. It is described in detail in *The Art of the Metaobject Protocol*.

method *n.* an *object* that is part of a *generic function* and which provides information about how that *generic function* should behave when its *arguments* are *objects* of certain *classes* or with certain identities.

method combination *n.* 1. generally, the composition of a set of *methods* to produce an *effective method* for a *generic function*. 2. an object of *type* **method-combination**, which represents the details of how the *method combination*$_1$ for one or more specific *generic functions* is to be performed.

method-defining form *n.* a *form* that defines a *method* for a *generic function*, whether explicitly or implicitly. See Section 7.6.1 (Introduction to Generic Functions).

method-defining operator *n.* an *operator* corresponding to a *method-defining form*. See Figure 7–1.

minimal compilation *n.* actions the *compiler* must take at compile time. See Section 3.2.2 (Compilation Semantics).

modified lambda list *n.* a list resembling an *ordinary lambda list* in form and purpose, but which deviates in syntax or functionality from the definition of an *ordinary lambda list*. See *ordinary lambda list*. "**deftype** uses a modified lambda list."

most recent *adj.* innermost; that is, having been *established* (and not yet *disestablished*) more recently than any other of its kind.

multiple escape *n.*, *adj.* 1. *n.* the *syntax type* of a *character* that is used in pairs to indicate that the enclosed *characters* are to be treated as *alphabetic*$_2$ *characters* with their *case* preserved. For details, see Section 2.1.4.5 (Multiple Escape Characters). 2. *adj.* (of a *character*) having the *multiple escape syntax type*. 3. *n.* a *multiple escape*$_2$ *character*. (In the *standard readtable*, *vertical-bar* is a *multiple escape character*.)

multiple values *n.* 1. more than one *value*. "The function **truncate** returns multiple values." 2. a variable number of *values*, possibly including zero or one. "The function **values** returns multiple values." 3. a fixed number of values other than one. "The macro **multiple-value-bind** is among the few operators in Common Lisp which can detect and manipulate multiple values."

N

name *n.*, *v.t.* 1. *n.* an *identifier* by which an *object*, a *binding*, or an *exit point* is referred to by association using a *binding*. 2. *v.t.* to give a *name* to. 3. *n.* (of an *object* having a name component) the *object* which is that component. "The string which is a symbol's name is returned by **symbol-name**." 4. *n.* (of a *pathname*) a. the name component, returned by **pathname-name**. b. the entire namestring, returned by **namestring**. 5. *n.* (of a *character*) a *string* that names the *character* and that has *length* greater than one. (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*.)

named constant *n.* a *variable* that is defined by Common Lisp, by the *implementation*, or by user code (see the *macro* **defconstant**) to always *yield* the same *value* when *evaluated*. "The value of a named constant may not be changed by assignment or by binding."

namespace *n.* 1. *bindings* whose denotations are restricted to a particular kind. "The bindings of names to tags is the tag namespace." 2. any *mapping* whose domain is a set of *names*. "A package defines a namespace."

namestring *n.* a *string* that represents a *filename* using either the *standardized* notation for naming *logical pathnames* described in Section 19.3.1 (Syntax of Logical Pathname Namestrings), or some *implementation-defined* notation for naming a *physical pathname*.

newline *n.* the *standard character* ⟨*Newline*⟩, notated for the *Lisp reader* as `#\Newline`.

next method *n.* the next *method* to be invoked with respect to a given *method* for a particular set of arguments or argument *classes*. See Section 7.6.6.1.3 (Applying method combination to the sorted list of applicable methods).

nickname *n.* (of a *package*) one of possibly several *names* that can be used to refer to the *package* but that is not the primary *name* of the *package*.

nil *n.* the *object* that is at once the *symbol* named `"NIL"` in the `COMMON-LISP` *package*, the *empty list*, the *boolean* (or *generalized boolean*) representing *false*, and the *name* of the *empty type*.

non-atomic *adj.* being other than an *atom*; *i.e.*, being a *cons*.

non-constant variable *n.* a *variable* that is not a *constant variable*.

non-correctable *adj.* (of an *error*) not intentionally *correctable*. (Because of the dynamic nature of *restarts*, it is neither possible nor generally useful to completely prohibit an *error* from being *correctable*. This term is used in order to express an intent that no special effort should be made by *code* signaling an *error* to make that *error correctable*; however, there is no actual requirement on *conforming programs* or *conforming implementations* imposed by this term.)

non-empty *adj.* having at least one *element*.

non-generic function *n.* a *function* that is not a *generic function*.

non-graphic *adj.* (of a *character*) not *graphic*. See Section 13.1.4.1 (Graphic Characters).

non-list *n.*, *adj.* other than a *list*; *i.e.*, a *non-nil atom*.

non-local exit *n.* a transfer of control (and sometimes *values*) to an *exit point* for reasons other than a *normal return*. "The operators **go**, **throw**, and **return-from** cause a non-local exit."

non-nil *n.*, *adj.* not **nil**. Technically, any *object* which is not **nil** can be referred to as *true*, but that would tend to imply a unique view of the *object* as a *generalized boolean*. Referring to such an *object* as *non-nil* avoids this implication.

non-null lexical environment *n.* a *lexical environment* that has additional information not present in the *global environment*, such as one or more *bindings*.

non-simple *adj.* not *simple*.

non-terminating *adj.* (of a *macro character*) being such that it is treated as a constituent *character* when it appears in the middle of an extended token. See Section 2.2 (Reader Algorithm).

non-top-level form *n.* a *form* that, by virtue of its position as a *subform* of another *form*, is not a *top level form*. See Section 3.2.3.1 (Processing of Top Level Forms).

normal return *n.* the natural transfer of control and *values* which occurs after the complete *execution* of a *form*.

normalized *adj.*, *ANSI*, *IEEE* (of a *float*) conforming to the description of "normalized" as described by *IEEE Standard for Binary Floating-Point Arithmetic*. See *denormalized*.

null *adj.*, *n.* 1. *adj.* a. (of a *list*) having no *elements*: empty. See *empty list*. b. (of a *string*) having a *length* of zero. (It is common, both within this document and in observed spoken behavior, to refer to an empty string by an apparent definite reference, as in "the *null string*" even though no attempt is made to *intern*$_2$ null strings. The phrase "a *null string*" is technically more correct, but is generally considered awkward by most Lisp programmers. As such, the phrase "the *null string*" should be treated as an indefinite reference in all cases except for anaphoric references.) c. (of an *implementation-defined attribute* of a *character*) An *object* to which the value of that *attribute* defaults if no specific value was requested. 2. *n.* an *object* of *type* **null** (the only such *object* being **nil**).

null lexical environment *n.* the *lexical environment* which has no *bindings*.

number *n.* an *object* of *type* **number**.

numeric *adj.* (of a *character*) being one of the *standard characters* 0 through *9*, or being some other *graphic character* defined by the *implementation* to be *numeric*.

O

object *n.* 1. any Lisp datum. "The function **cons** creates an object which refers to two other objects." 2. (immediately following the name of a *type*) an *object* which is of that *type*, used to emphasize that the *object* is not just a *name* for an object of that *type* but really an *element* of the *type* in cases where *objects* of that *type* (such as **function** or **class**) are commonly referred to by *name*. "The function **symbol-function** takes a function name and returns a function object."

object-traversing *adj.* operating in succession on components of an *object*. "The operators **mapcar**, **maphash**, **with-package-iterator** and **count** perform object-traversing operations."

open *adj.*, *v.t.* (a *file*) 1. *v.t.* to create and return a *stream* to the *file*. 2. *adj.* (of a *stream*) having been $opened_1$, but not yet *closed*.

operator *n.* 1. a *function*, *macro*, or *special operator*. 2. a *symbol* that names such a *function*, *macro*, or *special operator*. 3. (in a **function** *special form*) the *cadr* of the **function** *special form*, which might be either an $operator_2$ or a *lambda expression*. 4. (of a *compound form*) the *car* of the *compound form*, which might be either an $operator_2$ or a *lambda expression*, and which is never `(setf` *symbol*`)`.

optimize quality *n.* one of several aspects of a program that might be optimizable by certain compilers. Since optimizing one such quality might conflict with optimizing another, relative priorities for qualities can be established in an **optimize** *declaration*. The *standardized optimize qualities* are `compilation-speed` (speed of the compilation process), `debug` (ease of debugging), `safety` (run-time error checking), `space` (both code size and run-time space), and `speed` (of the object code). *Implementations* may define additional *optimize qualities*.

optional parameter *n.* A *parameter* for which a corresponding positional *argument* is optional. If the *argument* is not supplied, a default value is used. See also *supplied-p parameter*.

ordinary function *n.* a *function* that is not a *generic function*.

ordinary lambda list *n.* the kind of *lambda list* used by **lambda**. See *modified lambda list* and *extended lambda list*. "**defun** uses an ordinary lambda list."

otherwise inaccessible part *n.* (of an *object*, O_1) an *object*, O_2, which would be made *inaccessible* if O_1 were made *inaccessible*. (Every *object* is an *otherwise inaccessible part* of itself.)

output *adj.* (of a *stream*) supporting output operations (*i.e.*, being a "data sink"). An *output stream* might also be an *input stream*, in which case it is sometimes called a *bidirectional stream*. See the *function* **output-stream-p**.

P

package *n.* an *object* of *type* **package**.

package cell *n. Trad.* (of a *symbol*) The *place* in a *symbol* that holds one of possibly several *packages* in which the *symbol* is *interned*, called the *home package*, or which holds **nil** if no such *package* exists or is known. See the *function* **symbol-package**.

package designator *n.* a *designator* for a *package*; that is, an *object* that denotes a *package* and that is one of: a *string designator* (denoting the *package* that has the *string* that it designates as its *name* or as one of its *nicknames*), or a *package* (denoting itself).

package marker *n.* a character which is used in the textual notation for a symbol to separate the package name from the symbol name, and which is *colon* in the *standard readtable*. See Section 2.1 (Character Syntax).

package prefix *n.* a notation preceding the *name* of a *symbol* in text that is processed by the *Lisp reader*, which uses a *package name* followed by one or more *package markers*, and which indicates that the symbol is looked up in the indicated *package*.

package registry *n.* A mapping of *names* to *package objects*. It is possible for there to be a *package object* which is not in this mapping; such a *package* is called an *unregistered package*. *Operators* such as **find-package** consult this mapping in order to find a *package* from its *name*. *Operators* such as **do-all-symbols**, **find-all-symbols**, and **list-all-packages** operate only on *packages* that exist in the *package registry*.

pairwise *adv.* (of an adjective on a set) applying individually to all possible pairings of elements of the set. "The types A, B, and C are pairwise disjoint if A and B are disjoint, B and C are disjoint, and A and C are disjoint."

parallel *adj. Trad.* (of *binding* or *assignment*) done in the style of **psetq**, **let**, or **do**; that is, first evaluating all of the *forms* that produce *values*, and only then *assigning* or *binding* the *variables* (or *places*). Note that this does not imply traditional computational "parallelism" since the *forms* that produce *values* are evaluated *sequentially*. See *sequential*.

parameter *n.* 1. (of a *function*) a *variable* in the definition of a *function* which takes on the *value* of a corresponding *argument* (or of a *list* of corresponding arguments) to that *function* when it is called, or which in some cases is given a default value because there is no corresponding *argument*. 2. (of a *format directive*) an *object* received as data flow by a *format directive* due to a prefix notation within the *format string* at the *format directive*'s point of use. See Section 22.3 (Formatted Output). "In `"~3,'0D"`, the number `3` and the character `#\0` are parameters to the `~D` format directive."

parameter specializer *n.* 1. (of a *method*) an *expression* which constrains the *method* to be applicable only to *argument* sequences in which the corresponding *argument* matches the *parameter specializer*. 2. a *class*, or a *list* `(eql` *object*`)`.

parameter specializer name *n.* 1. (of a *method* definition) an expression used in code to name a *parameter specializer*. See Section 7.6.2 (Introduction to Methods). 2. a *class*, a *symbol* naming a *class*, or a *list* `(eql` *form*`)`.

pathname *n.* an *object* of *type* **pathname**, which is a structured representation of the name of a *file*. A *pathname* has six components: a "host," a "device," a "directory," a "name," a "type," and a "version."

pathname designator *n.* a *designator* for a *pathname*; that is, an *object* that denotes a *pathname* and that is one of: a *pathname namestring* (denoting the corresponding *pathname*), a *stream associated with a file* (denoting the *pathname* used to open the *file*; this may be, but is not required to be, the actual name of the *file*), or a *pathname* (denoting itself). See Section 21.1.1.1.2 (Open and Closed Streams).

physical pathname *n.* a *pathname* that is not a *logical pathname*.

place *n.* 1. a *form* which is suitable for use as a *generalized reference*. 2. the conceptual location referred to by such a *place*$_1$.

plist [ˈ**pē͵list**] *n.* a *property list*.

portable *adj.* (of *code*) required to produce equivalent results and observable side effects in all *conforming implementations*.

potential copy *n.* (of an *object* O_1 subject to constriants) an *object* O_2 that if the specified constraints are satisfied by O_1 without any modification might or might not be *identical* to O_1, or else that must be a *fresh object* that resembles a *copy* of O_1 except that it has been modified as necessary to satisfy the constraints.

potential number *n.* A textual notation that might be parsed by the *Lisp reader* in some *conforming implementation* as a *number* but is not required to be parsed as a *number*. No *object* is a *potential number*—either an *object* is a *number* or it is not. See Section 2.3.1.1 (Potential Numbers as Tokens).

pprint dispatch table *n.* an *object* that can be the *value* of ***print-pprint-dispatch*** and hence can control how *objects* are printed when ***print-pretty*** is *true*. See Section 22.2.1.4 (Pretty Print Dispatch Tables).

predicate *n.* a *function* that returns a *generalized boolean* as its first value.

present *n.* 1. (of a *feature* in a *Lisp image*) a state of being that is in effect if and only if the *symbol* naming the *feature* is an *element* of the *features list*. 2. (of a *symbol* in a *package*) being accessible in that *package* directly, rather than being inherited from another *package*.

pretty print *v.t.* (an *object*) to invoke the *pretty printer* on the *object*.

pretty printer *n.* the procedure that prints the character representation of an *object* onto a *stream* when the *value* of ***print-pretty*** is *true*, and that uses layout techniques (*e.g.*, indentation) that tend to highlight the structure of the *object* in a way that makes it easier for human readers to parse visually. See the *variable* ***print-pprint-dispatch*** and Section 22.2 (The Lisp Pretty Printer).

pretty printing stream *n.* a *stream* that does pretty printing. Such streams are created by the *function* **pprint-logical-block** as a link between the output stream and the logical block.

primary method *n.* a member of one of two sets of *methods* (the set of *auxiliary methods* is the other) that form an exhaustive partition of the set of *methods* on the *method*'s *generic function*. How these sets are determined is dependent on the *method combination* type; see Section 7.6.2 (Introduction to Methods).

primary value *n.* (of *values* resulting from the *evaluation* of a *form*) the first *value*, if any, or else **nil** if there are no *values*. "The primary value returned by **truncate** is an integer quotient, truncated toward zero."

principal *adj.* (of a value returned by a Common Lisp *function* that implements a mathematically irrational or transcendental function defined in the complex domain) of possibly many (sometimes an infinite number of) correct values for the mathematical function, being the particular *value* which the corresponding Common Lisp *function* has been defined to return.

print name *n.* *Trad.* (usually of a *symbol*) a *name*$_3$.

printer control variable *n.* a *variable* whose specific purpose is to control some action of the *Lisp printer*; that is, one of the *variables* in Figure 22–1, or else some *implementation-defined variable* which is defined by the *implementation* to be a *printer control variable*.

printer escaping *n.* The combined state of the *printer control variables* ***print-escape*** and ***print-readably***. If the value of either ***print-readably*** or ***print-escape*** is *true*, then ***printer escaping*** is "enabled"; otherwise (if the values of both ***print-readably*** and ***print-escape*** are *false*), then *printer escaping* is "disabled".

printing *adj.* (of a *character*) being a *graphic character* other than *space*.

process *v.t.* (a *form* by the *compiler*) to perform *minimal compilation*, determining the time of evaluation for a *form*, and possibly *evaluating* that *form* (if required).

processor *n.*, *ANSI* an *implementation*.

proclaim *v.t.* (a *proclamation*) to *establish* that *proclamation*.

proclamation *n.* a *global declaration*.

prog tag *n. Trad.* a *go tag*.

program *n. Trad.* Common Lisp *code*.

programmer *n.* an active entity, typically a human, that writes a *program*, and that might or might not also be a *user* of the *program*.

programmer code *n. code* that is supplied by the programmer; that is, *code* that is not *system code*.

proper list *n.* A *list* terminated by the *empty list*. (The *empty list* is a *proper list*.) See *improper list*.

proper name *n.* (of a *class*) a *symbol* that *names* the *class* whose *name* is that *symbol*. See the *functions* **class-name** and **find-class**.

proper sequence *n.* a *sequence* which is not an *improper list*; that is, a *vector* or a *proper list*.

proper subtype *n.* (of a *type*) a *subtype* of the *type* which is not the *same type* as the *type* (*i.e.*, its *elements* are a "proper subset" of the *type*).

property *n.* (of a *property list*) 1. a conceptual pairing of a *property indicator* and its associated *property value* on a *property list*. 2. a *property value*.

property indicator *n.* (of a *property list*) the *name* part of a *property*, used as a *key* when looking up a *property value* on a *property list*.

property list *n.* 1. a *list* containing an even number of *elements* that are alternating *names* (sometimes called *indicators* or *keys*) and *values* (sometimes called *properties*). When there is more than one *name* and *value* pair with the *identical name* in a *property list*, the first such pair determines the *property*. 2. (of a *symbol*) the component of the *symbol* containing a *property list*.

property value *n.* (of a *property indicator* on a *property list*) the *object* associated with the *property indicator* on the *property list*.

purports to conform *v.* makes a good-faith claim of conformance. This term expresses intention to conform, regardless of whether the goal of that intention is realized in practice. For example, language implementations have been known to have bugs, and while an *implementation* of this specification with bugs might not be a *conforming implementation*, it can still *purport to conform*. This is an important distinction in certain specific cases; *e.g.*, see the *variable* ***features***.

Q

qualified method *n.* a *method* that has one or more *qualifiers*.

qualifier *n.* (of a *method* for a *generic function*) one of possibly several *objects* used to annotate the *method* in a way that identifies its role in the *method combination*. The *method combination type* determines how many *qualifiers* are permitted for each *method*, which *qualifiers* are permitted, and the semantics of those *qualifiers*.

query I/O *n.* the *bidirectional stream* that is the *value* of the *variable* ***query-io***.

quoted object *n.* an *object* which is the second element of a **quote** *form*.

R

radix *n.* an *integer* between 2 and 36, inclusive, which can be used to designate a base with respect to which certain kinds of numeric input or output are performed. (There are *n* valid digit characters for any given *radix n*, and those digits are the first *n* digits in the sequence `0`, `1`, ..., `9`, `A`, `B`, ..., `Z`, which have the weights `0`, `1`, ..., `9`, `10`, `11`, ..., `35`, respectively. Case is not significant in parsing numbers of radix greater than `10`, so "9b8a" and "9B8A" denote the same *radix* `16` number.)

random state *n.* an *object* of *type* **random-state**.

rank *n.* a non-negative *integer* indicating the number of *dimensions* of an *array*.

ratio *n.* an *object* of *type* **ratio**.

ratio marker *n.* a character which is used in the textual notation for a *ratio* to separate the numerator from the denominator, and which is *slash* in the *standard readtable*. See Section 2.1 (Character Syntax).

rational *n.* an *object* of *type* **rational**.

read *v.t.* 1. (a *binding* or *slot* or component) to obtain the *value* of the *binding* or *slot*. 2. (an *object* from a *stream*) to parse an *object* from its representation on the *stream*.

readably *adv.* (of a manner of printing an *object* O_1) in such a way as to permit the *Lisp Reader* to later *parse* the printed output into an *object* O_2 that is *similar* to O_1.

reader *n.* 1. a *function* that *reads*$_1$ a *variable* or *slot*. 2. the *Lisp reader*.

reader macro *n.* 1. a textual notation introduced by dispatch on one or two *characters* that defines special-purpose syntax for use by the *Lisp reader*, and that is implemented by a *reader macro function*. See Section 2.2 (Reader Algorithm). 2. the *character* or *characters* that introduce a *reader macro*$_1$; that is, a *macro character* or the conceptual pairing of a *dispatching macro character* and the *character* that follows it. (A *reader macro* is not a kind of *macro*.)

reader macro function *n.* a *function designator* that denotes a *function* that implements a *reader macro*$_2$. See the *functions* **set-macro-character** and **set-dispatch-macro-character**.

readtable *n.* an *object* of *type* **readtable**.

readtable case *n.* an attribute of a *readtable* whose value is a *case sensitivity mode*, and that selects the manner in which *characters* in a *symbol*'s *name* are to be treated by the *Lisp reader* and the *Lisp printer*. See Section 23.1.2 (Effect of Readtable Case on the Lisp Reader) and Section 22.1.3.3.2 (Effect of Readtable Case on the Lisp Printer).

readtable designator *n.* a *designator* for a *readtable*; that is, an *object* that denotes a *readtable* and that is one of: **nil** (denoting the *standard readtable*), or a *readtable* (denoting itself).

recognizable subtype *n.* (of a *type*) a *subtype* of the *type* which can be reliably detected to be such by the *implementation*. See the *function* **subtypep**.

reference *n.*, *v.t.* 1. *n.* an act or occurrence of referring to an *object*, a *binding*, an *exit point*, a *tag*, or an *environment*. 2. *v.t.* to refer to an *object*, a *binding*, an *exit point*, a *tag*, or an *environment*, usually by *name*.

registered package *n.* a *package object* that is installed in the *package registry*. (Every *registered package* has a *name* that is a *string*, as well as zero or more *string* nicknames. All *packages* that are initially specified by Common Lisp or created by **make-package** or **defpackage** are *registered packages*. *Registered packages* can be turned into *unregistered packages* by **delete-package**.)

relative *adj.* 1. (of a *time*) representing an offset from an *absolute time* in the units appropriate to that time. For example, a *relative internal time* is the difference between two *absolute internal times*, and is measured in *internal time units*. 2. (of a *pathname*) representing a position in a directory hierarchy by motion from a position other than the root, which might therefore vary. "The notation `#P"../foo.text"` denotes a relative pathname if the host file system is Unix." See *absolute*.

repertoire *n.*, *ISO* a *subtype* of **character**. See Section 13.1.2.2 (Character Repertoires).

report *n.* (of a *condition*) to *call* the *function* **print-object** on the *condition* in an *environment* where the *value* of ***print-escape*** is *false*.

report message *n.* the text that is output by a *condition reporter*.

required parameter *n.* A *parameter* for which a corresponding positional *argument* must be supplied when *calling* the *function*.

rest list *n.* (of a *function* having a *rest parameter*) The *list* to which the *rest parameter* is *bound* on some particular *call* to the *function*.

rest parameter *n.* A *parameter* which was introduced by **&rest**.

restart *n.* an *object* of *type* **restart**.

restart designator *n.* a *designator* for a *restart*; that is, an *object* that denotes a *restart* and that is one of: a *non-nil symbol* (denoting the most recently established *active restart* whose *name* is that *symbol*), or a *restart* (denoting itself).

restart function *n.* a *function* that invokes a *restart*, as if by **invoke-restart**. The primary purpose of a *restart function* is to provide an alternate interface. By convention, a *restart function* usually has the same name as the *restart* which it invokes. Figure 26–4 shows a list of the *standardized restart functions*.

abort	**muffle-warning**	**use-value**
continue	**store-value**	

Figure 26–4. Standardized Restart Functions

return *v.t.* (of *values*) 1. (from a *block*) to transfer control and *values* from the *block*; that is, to cause the *block* to *yield* the *values* immediately without doing any further evaluation of the *forms* in its body. 2. (from a *form*) to *yield* the *values*.

return value *n.* *Trad.* a $value_1$

right-parenthesis *n.* the *standard character* ")", that is variously called "right parenthesis" or "close parenthesis" See Figure 2–5.

run time *n.* 1. *load time* 2. *execution time*

run-time compiler *n.* refers to the **compile** function or to *implicit compilation*, for which the compilation and run-time *environments* are maintained in the same *Lisp image*.

run-time definition *n.* a definition in the *run-time environment*.

run-time environment *n.* the *environment* in which a program is *executed*.

S

safe *adj.* 1. (of *code*) processed in a *lexical environment* where the the highest **safety** level (3) was in effect. See **optimize**. 2. (of a *call*) a *safe call*.

safe call *n.* a *call* in which the *call*, the *function* being *called*, and the point of *functional evaluation* are all $safe_1$ *code*. For more detailed information, see Section 3.5.1.1 (Safe and Unsafe Calls).

same *adj.* 1. (of *objects* under a specified *predicate*) indistinguishable by that *predicate*. "The symbol `car`, the string `"car"`, and the string `"CAR"` are the `same` under **string-equal**". 2. (of *objects* if no predicate is implied by context) indistinguishable by **eql**. Note that **eq** might be capable of distinguishing some *numbers* and *characters* which **eql** cannot distinguish, but the nature of such, if any, is *implementation-dependent*. Since **eq** is used only rarely in this specification, **eql** is the default predicate when none is mentioned explicitly. "The conses returned by two successive calls to **cons** are never the same." 3. (of *types*) having the same set of *elements*; that is, each *type* is a *subtype* of the others. "The types specified by `(integer 0 1)`, `(unsigned-byte 1)`, and `bit` are the same."

satisfy the test *v.* (of an *object* being considered by a *sequence function*) 1. (for a one *argument* test) to be in a state such that the *function* which is the *predicate argument* to the *sequence function* returns *true* when given a single *argument* that is the result of calling the *sequence function*'s *key argument* on the *object* being considered. See Section 17.2.2 (Satisfying a One-Argument Test). 2. (for a two *argument* test) to be in a state such that the two-place *predicate* which is the *sequence function*'s *test* *argument* returns *true* when given a first *argument* that is the *object* being considered, and when given a second *argument* that is the result of calling the *sequence function*'s *key argument* on an *element* of the *sequence function*'s *sequence argument* which is being tested for equality; or to be in a state such that the *test-not function* returns *false* given the same *arguments*. See Section 17.2.1 (Satisfying a Two-Argument Test).

scope *n.* the structural or textual region of code in which *references* to an *object*, a *binding*, an *exit point*, a *tag*, or an *environment* (usually by *name*) can occur.

script *n. ISO* one of possibly several sets that form an *exhaustive partition* of the type **character**. See Section 13.1.2.1 (Character Scripts).

secondary value *n.* (of *values* resulting from the *evaluation* of a *form*) the second *value*, if any, or else **nil** if there are fewer than two *values*. "The secondary value returned by **truncate** is a remainder."

section *n.* a partitioning of output by a *conditional newline* on a *pretty printing stream*. See Section 22.2.1.1 (Dynamic Control of the Arrangement of Output).

self-evaluating object *n.* an *object* that is neither a *symbol* nor a *cons*. If a *self-evaluating object* is *evaluated*, it *yields* itself as its only *value*. "Strings are self-evaluating objects."

semi-standard *adj.* (of a language feature) not required to be implemented by any *conforming implementation*, but nevertheless recommended as the canonical approach in situations where an *implementation* does plan to support such a feature. The presence of *semi-standard* aspects in the language is intended to lessen portability problems and reduce the risk of gratuitous divergence among *implementations* that might stand in the way of future standardization.

semicolon *n.* the *standard character* that is called "semicolon" (;). See Figure 2–5.

sequence *n.* 1. an ordered collection of elements 2. a *vector* or a *list*.

sequence function *n.* one of the *functions* in Figure 17–1, or an *implementation-defined function* that operates on one or more *sequences*. and that is defined by the *implementation* to be a *sequence function*.

sequential *adj. Trad.* (of *binding* or *assignment*) done in the style of **setq**, **let***, or **do***; that is, interleaving the evaluation of the *forms* that produce *values* with the *assignments* or *bindings* of the *variables* (or *places*). See *parallel*.

sequentially *adv.* in a *sequential* way.

serious condition *n.* a *condition* of *type* **serious-condition**, which represents a *situation* that is generally sufficiently severe that entry into the *debugger* should be expected if the *condition* is *signaled* but not *handled*.

session *n.* the conceptual aggregation of events in a *Lisp image* from the time it is started to the time it is terminated.

set *v.t. Trad.* (any *variable* or a *symbol* that is the *name* of a *dynamic variable*) to *assign* the *variable*.

setf expander *n.* a function used by **setf** to compute the *setf expansion* of a *place*.

setf expansion *n.* a set of five *expressions*$_1$ that, taken together, describe how to store into a *place* and which *subforms* of the macro call associated with the *place* are evaluated. See Section 5.1.1.2 (Setf Expansions).

setf function *n.* a *function* whose *name* is `(setf` *symbol*`)`.

setf function name *n.* (of a *symbol* *S*) the *list* `(setf` *S*`)`.

shadow *v.t.* 1. to override the meaning of. "That binding of `X` shadows an outer one." 2. to hide the presence of. "That **macrolet** of `F` shadows the outer **flet** of `F`." 3. to replace. "That package shadows the symbol `cl:car` with its own symbol `car`."

shadowing symbol *n.* (in a *package*) an *element* of the *package*'s *shadowing symbols list*.

shadowing symbols list *n.* (of a *package*) a *list*, associated with the *package*, of *symbols* that are to be exempted from 'symbol conflict errors' detected when packages are *used*. See the *function* **package-shadowing-symbols**.

shared slot *n.* (of a *class*) a *slot accessible* in more than one *instance* of a *class*; specifically, such a *slot* is *accessible* in all *direct instances* of the *class* and in those *indirect instances* whose *class* does not *shadow*$_1$ the *slot*.

sharpsign *n.* the *standard character* that is variously called "number sign," "sharp," or "sharp sign" (`#`). See Figure 2–5.

short float *n.* an *object* of *type* **short-float**.

sign *n.* one of the *standard characters* "`+`" or "`-`".

signal *v.* to announce, using a standard protocol, that a particular situation, represented by a *condition*, has been detected. See Section 9.1 (Condition System Concepts).

signature *n.* (of a *method*) a description of the *parameters* and *parameter specializers* for the *method* which determines the *method*'s applicability for a given set of required *arguments*, and which also describes the *argument* conventions for its other, non-required *arguments*.

similar *adj.* (of two *objects*) defined to be equivalent under the *similarity* relationship.

similarity *n.* a two-place conceptual equivalence predicate, which is independent of the *Lisp image* so that two *objects* in different *Lisp images* can be understood to be equivalent under this predicate. See Section 3.2.4 (Literal Objects in Compiled Files).

simple *adj.* 1. (of an *array*) being of *type* **simple-array**. 2. (of a *character*) having no *implementation-defined attributes*, or else having *implementation-defined attributes* each of which has the *null* value for that *attribute*.

simple array *n.* an *array* of *type* **simple-array**.

simple bit array *n.* a *bit array* that is a *simple array*; that is, an *object* of *type* `(simple-array bit)`.

simple bit vector *n.* a *bit vector* of *type* **simple-bit-vector**.

simple condition *n.* a *condition* of *type* **simple-condition**.

simple general vector *n.* a *simple vector*.

simple string *n.* a *string* of *type* **simple-string**.

simple vector *n.* a *vector* of *type* **simple-vector**, sometimes called a "*simple general vector*." Not all *vectors* that are *simple* are *simple vectors*—only those that have *element type* **t**.

single escape *n.*, *adj.* 1. *n.* the *syntax type* of a *character* that indicates that the next *character* is to be treated as an *alphabetic*$_2$ *character* with its *case* preserved. For details, see Section 2.1.4.6 (Single Escape Character). 2. *adj.* (of a *character*) having the *single escape syntax type*. 3. *n.* a *single escape*$_2$ *character*. (In the *standard readtable*, *slash* is the only *single escape*.)

single float *n.* an *object* of *type* **single-float**.

single-quote *n.* the *standard character* that is variously called "apostrophe," "acute accent," "quote," or "single quote" (`'`). See Figure 2–5.

singleton *adj.* (of a *sequence*) having only one *element*. "`(list 'hello)` returns a singleton list."

situation *n.* the *evaluation* of a *form* in a specific *environment*.

slash *n.* the *standard character* that is variously called "solidus" or "slash" (`/`). See Figure 2–5.

slot *n.* a component of an *object* that can store a *value*.

slot specifier *n.* a representation of a *slot* that includes the *name* of the *slot* and zero or more *slot* options. A *slot* option pertains only to a single *slot*.

source code *n.* *code* representing *objects* suitable for *evaluation* (*e.g.*, *objects* created by **read**, by *macro expansion*, or by *compiler macro expansion*).

source file *n.* a *file* which contains a textual representation of *source code*, that can be edited, *loaded*, or *compiled*.

space *n.* the *standard character* ⟨*Space*⟩, notated for the *Lisp reader* as `#\Space`.

special form *n.* a *list*, other than a *macro form*, which is a *form* with special syntax or special *evaluation* rules or both, possibly manipulating the *evaluation environment* or control flow or both. The first element of a *special form* is a *special operator*.

special operator *n.* one of a fixed set of *symbols*, enumerated in Figure 3–2, that may appear in the *car* of a *form* in order to identify the *form* as a *special form*.

special variable *n.* *Trad.* a *dynamic variable*.

specialize *v.t.* (a *generic function*) to define a *method* for the *generic function*, or in other words, to refine the behavior of the *generic function* by giving it a specific meaning for a particular set of *classes* or *arguments*.

specialized *adj.* 1. (of a *generic function*) having *methods* which *specialize* the *generic function*. 2. (of an *array*) having an *actual array element type* that is a *proper subtype* of the *type* **t**; see Section 15.1.1 (Array Elements). "`(make-array 5 :element-type 'bit)` makes an array of length five that is specialized for bits."

specialized lambda list *n.* an *extended lambda list* used in *forms* that *establish method* definitions, such as **defmethod**. See Section 3.4.3 (Specialized Lambda Lists).

spreadable argument list designator *n.* a *designator* for a *list* of *objects*; that is, an *object* that denotes a *list* and that is a *non-null list* $L1$ of length n, whose last element is a *list* $L2$ of length m (denoting a list $L3$ of length $m+n-1$ whose *elements* are $L1_i$ for $i < n-1$ followed by $L2_j$ for $j < m$). "The list (1 2 (3 4 5)) is a spreadable argument list designator for the list (1 2 3 4 5)."

stack allocate *v.t. Trad.* to allocate in a non-permanent way, such as on a stack. Stack-allocation is an optimization technique used in some *implementations* for allocating certain kinds of *objects* that have *dynamic extent*. Such *objects* are allocated on the stack rather than in the heap so that their storage can be freed as part of unwinding the stack rather than taking up space in the heap until the next garbage collection. What *types* (if any) can have *dynamic extent* can vary from *implementation* to *implementation*. No *implementation* is ever required to perform stack-allocation.

stack-allocated *adj. Trad.* having been *stack allocated*.

standard character *n.* a *character* of *type* **standard-char**, which is one of a fixed set of 96 such *characters* required to be present in all *conforming implementations*. See Section 2.1.3 (Standard Characters).

standard class *n.* a *class* that is a *generalized instance* of *class* **standard-class**.

standard generic function a *function* of *type* **standard-generic-function**.

standard input *n.* the *input stream* which is the *value* of the *dynamic variable* ***standard-input***.

standard method combination *n.* the *method combination* named **standard**.

standard object *n.* an *object* that is a *generalized instance* of *class* **standard-object**.

standard output *n.* the *output stream* which is the *value* of the *dynamic variable* ***standard-output***.

standard pprint dispatch table *n.* A *pprint dispatch table* that is *different* from the *initial pprint dispatch table*, that implements *pretty printing* as described in this specification, and that, unlike other *pprint dispatch tables*, must never be modified by any program. (Although the definite reference "the *standard pprint dispatch table*" is generally used within this document, it is actually *implementation-dependent* whether a single *object* fills the role of the *standard pprint dispatch table*, or whether there might be multiple such objects, any one of which could be used on any given occasion where "the *standard pprint dispatch table*" is called for. As such, this phrase should be seen as an indefinite reference in all cases except for anaphoric references.)

standard readtable *n.* A *readtable* that is *different* from the *initial readtable*, that implements the *expression* syntax defined in this specification, and that, unlike other *readtables*, must never be modified by any program. (Although the definite reference "the *standard readtable*" is generally used within this document, it is actually *implementation-dependent* whether a single *object* fills the role of the *standard readtable*, or whether there might be multiple such objects, any one of which could be used on any given occasion where "the *standard readtable*" is called for. As such, this phrase should be seen as an indefinite reference in all cases except for anaphoric references.)

standard syntax *n.* the syntax represented by the *standard readtable* and used as a reference syntax throughout this document. See Section 2.1 (Character Syntax).

standardized *adj.* (of a *name*, *object*, or definition) having been defined by Common Lisp. "All standardized variables that are required to hold bidirectional streams have "`-io*`" in their name."

startup environment *n.* the *global environment* of the running *Lisp image* from which the *compiler* was invoked.

step *v.t.*, *n.* 1. *v.t.* (an iteration *variable*) to *assign* the *variable* a new *value* at the end of an iteration, in preparation for a new iteration. 2. *n.* the *code* that identifies how the next value in an iteration is to be computed. 3. *v.t.* (*code*) to specially execute the *code*, pausing at intervals to allow user confirmation or intervention, usually for debugging.

stream *n.* 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.

stream associated with a file *n.* a *file stream*, or a *synonym stream* the *target* of which is a *stream associated with a file*. Such a *stream* cannot be created with **make-two-way-stream**, **make-echo-stream**, **make-broadcast-stream**, **make-concatenated-stream**, **make-string-input-stream**, or **make-string-output-stream**.

stream designator *n.* a *designator* for a *stream*; that is, an *object* that denotes a *stream* and that is one of: **t** (denoting the *value* of ***terminal-io***), **nil** (denoting the *value* of ***standard-input*** for *input stream designators* or denoting the *value* of ***standard-output*** for *output stream designators*), or a *stream* (denoting itself).

stream element type *n.* (of a *stream*) the *type* of data for which the *stream* is specialized.

stream variable *n.* a *variable* whose *value* must be a *stream*.

stream variable designator *n.* a *designator* for a *stream variable*; that is, a *symbol* that denotes a *stream variable* and that is one of: **t** (denoting ***terminal-io***), **nil** (denoting ***standard-input*** for *input stream variable designators* or denoting ***standard-output*** for *output stream variable designators*), or some other *symbol* (denoting itself).

string *n.* a specialized *vector* that is of *type* **string**, and whose elements are of *type* **character** or a *subtype* of *type* **character**.

string designator *n.* a *designator* for a *string*; that is, an *object* that denotes a *string* and that is one of: a *character* (denoting a *singleton string* that has the *character* as its only *element*), a *symbol* (denoting the *string* that is its *name*), or a *string* (denoting itself). The intent is that this term be consistent with the behavior of **string**; *implementations* that extend **string** must extend the meaning of this term in a compatible way.

string equal *adj.* the *same* under **string-equal**.

string stream *n.* a *stream* of *type* **string-stream**.

structure *n.* an *object* of *type* **structure-object**.

structure class *n.* a *class* that is a *generalized instance* of *class* **structure-class**.

structure name *n.* a *name* defined with **defstruct**. Usually, such a *type* is also a *structure class*, but there may be *implementation-dependent* situations in which this is not so, if the `:type` option to **defstruct** is used.

style warning *n.* a *condition* of *type* **style-warning**.

subclass *n.* a *class* that *inherits* from another *class*, called a *superclass*. (No *class* is a *subclass* of itself.)

subexpression *n.* (of an *expression*) an *expression* that is contained within the *expression*. (In fact, the state of being a *subexpression* is not an attribute of the *subexpression*, but really an attribute of the containing *expression* since the *same object* can at once be a *subexpression* in one context, and not in another.)

subform *n.* (of a *form*) an *expression* that is a *subexpression* of the *form*, and which by virtue of its position in that *form* is also a *form*. "`(f x)` and `x`, but not `exit`, are subforms of `(return-from exit (f x))`."

subrepertoire *n.* a subset of a *repertoire*.

subtype *n.* a *type* whose membership is the same as or a proper subset of the membership of another *type*, called a *supertype*. (Every *type* is a *subtype* of itself.)

superclass *n.* a *class* from which another *class* (called a *subclass*) *inherits*. (No *class* is a *superclass* of itself.) See *subclass*.

supertype *n.* a *type* whose membership is the same as or a proper superset of the membership of another *type*, called a *subtype*. (Every *type* is a *supertype* of itself.) See *subtype*.

supplied-p parameter *n.* a *parameter* which recieves its *generalized boolean* value implicitly due to the presence or absence of an *argument* corresponding to another *parameter* (such as an *optional parameter* or a *rest parameter*). See Section 3.4.1 (Ordinary Lambda Lists).

symbol *n.* an *object* of *type* **symbol**.

symbol macro *n.* a *symbol* that stands for another *form*. See the *macro* **symbol-macrolet**.

synonym stream *n.* 1. a *stream* of *type* **synonym-stream**, which is consequently 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*. See the *function* **make-synonym-stream**. 2. (to a *stream*) a *synonym stream* which has the *stream* as the *value* of its *synonym stream symbol*. 3. (to a *symbol*) a *synonym stream* which has the *symbol* as its *synonym stream symbol*.

synonym stream symbol *n.* (of a *synonym stream*) the *symbol* which names the *dynamic variable* which has as its *value* another *stream* for which the *synonym stream* is an alias.

syntax type *n.* (of a *character*) one of several classifications, enumerated in Figure 2–6, that are used for dispatch during parsing by the *Lisp reader*. See Section 2.1.4 (Character Syntax Types).

system class *n.* a *class* that may be of *type* **built-in-class** in a *conforming implementation* and hence cannot be inherited by *classes* defined by *conforming programs*.

system code *n.* *code* supplied by the *implementation* to implement this specification (*e.g.*, the definition of **mapcar**) or generated automatically in support of this specification (*e.g.*, during method combination); that is, *code* that is not *programmer code*.

T

t *n.* 1. a. the *boolean* representing true. b. the canonical *generalized boolean* representing true. (Although any *object* other than **nil** is considered *true* as a *generalized boolean*, `t` is generally used when there is no special reason to prefer one such *object* over another.) 2. the *name* of the *type* to which all *objects* belong—the *supertype* of all *types* (including itself). 3. the *name* of the *superclass* of all *classes* except itself.

tag *n.* 1. a *catch tag*. 2. a *go tag*.

tail *n.* (of a *list*) an *object* that is the *same* as either some *cons* which makes up that *list* or the *atom* (if any) which terminates the *list*. "The empty list is a tail of every proper list."

target *n.* 1. (of a *constructed stream*) a *constituent* of the *constructed stream*. "The target of a synonym stream is the value of its synonym stream symbol." 2. (of a *displaced array*) the *array* to which the *displaced array* is displaced. (In the case of a chain of *constructed streams* or *displaced arrays*, the unqualified term "*target*" always refers to the immediate *target* of the first item in the chain, not the immediate target of the last item.)

terminal I/O *n.* the *bidirectional stream* that is the *value* of the *variable* ***terminal-io***.

terminating *n.* (of a *macro character*) being such that, if it appears while parsing a token, it terminates that token. See Section 2.2 (Reader Algorithm).

tertiary value *n.* (of *values* resulting from the *evaluation* of a *form*) the third *value*, if any, or else **nil** if there are fewer than three *values*.

throw *v.* to transfer control and *values* to a *catch*. See the *special operator* **throw**.

tilde *n.* the *standard character* that is called "tilde" (`~`). See Figure 2–5.

time a representation of a point (*absolute time*) or an interval (*relative time*) on a time line. See *decoded time*, *internal time*, and *universal time*.

time zone *n.* a *rational* multiple of `1/3600` between `-24` (inclusive) and `24` (inclusive) that represents a time zone as a number of hours offset from Greenwich Mean Time. Time zone values increase with motion to the west, so Massachusetts, U.S.A. is in time zone `5`, California, U.S.A. is time zone `8`, and Moscow, Russia is time zone *-3*. (When "daylight savings time" is separately represented as an *argument* or *return value*, the *time zone* that accompanies it does not depend on whether daylight savings time is in effect.)

token *n.* a textual representation for a *number* or a *symbol*. See Section 2.3 (Interpretation of Tokens).

top level form *n.* a *form* which is processed specially by **compile-file** for the purposes of enabling *compile time evaluation* of that *form*. *Top level forms* include those *forms* which are not *subforms* of any other *form*, and certain other cases. See Section 3.2.3.1 (Processing of Top Level Forms).

trace output *n.* the *output stream* which is the *value* of the *dynamic variable* ***trace-output***.

tree *n.* 1. a binary recursive data structure made up of *conses* and *atoms*: the *conses* are themselves also *trees* (sometimes called "subtrees" or "branches"), and the *atoms* are terminal nodes (sometimes called *leaves*). Typically, the *leaves* represent data while the branches establish some relationship among that data. 2. in general, any recursive data structure that has some notion of "branches" and *leaves*.

tree structure *n.* (of a $tree_1$) the set of *conses* that make up the *tree*. Note that while the car_{1b} component of each such *cons* is part of the *tree structure*, the *objects* that are the $cars_2$ of each *cons* in the *tree* are not themselves part of its *tree structure* unless they are also *conses*.

true *n.* any *object* that is not *false* and that is used to represent the success of a *predicate* test. See t_1.

truename *n.* 1. the canonical *filename* of a *file* in the *file system*. See Section 20.1.3 (Truenames). 2. a *pathname* representing a $truename_1$.

two-way stream *n.* a *stream* of *type* **two-way-stream**, which is a *bidirectional composite stream* that receives its input from an associated *input stream* and sends its output to an associated *output stream*.

type *n.* 1. a set of *objects*, usually with common structure, behavior, or purpose. (Note that the expression "X is of type S_a" naturally implies that "X is of type S_b" if S_a is a *subtype* of S_b.) 2. (immediately following the name of a *type*) a *subtype* of that *type*. "The type **vector** is an array type."

type declaration *n.* a *declaration* that asserts that every reference to a specified *binding* within the scope of the *declaration* results in some *object* of the specified *type*.

type equivalent *adj.* (of two *types* X and Y) having the same *elements*; that is, X is a *subtype* of Y and Y is a *subtype* of X.

type expand *n.* to fully expand a *type specifier*, removing any references to *derived types*. (Common Lisp provides no program interface to cause this to occur, but the semantics of Common Lisp are such that every *implementation* must be able to do this internally, and some situations involving *type specifiers* are most easily described in terms of a fully expanded *type specifier*.)

type specifier *n.* an *expression* that denotes a *type*. "The symbol `random-state`, the list `(integer 3 5)`, the list `(and list (not null))`, and the class named `standard-class` are type specifiers."

U

unbound *adj.* not having an associated denotation in a *binding*. See *bound*.

unbound variable *n.* a *name* that is syntactically plausible as the name of a *variable* but which is not *bound* in the *variable namespace*.

undefined function *n.* a *name* that is syntactically plausible as the name of a *function* but which is not *bound* in the *function namespace*.

unintern *v.t.* (a *symbol* in a *package*) to make the *symbol* not be *present* in that *package*. (The *symbol* might continue to be *accessible* by inheritance.)

uninterned *adj.* (of a *symbol*) not *accessible* in any *package*; *i.e.*, not *interned*$_1$.

universal time *n.* *time*, represented as a non-negative *integer* number of seconds. *Absolute universal time* is measured as an offset from the beginning of the year 1900 (ignoring *leap seconds*). See Section 25.1.4.2 (Universal Time).

unqualified method *n.* a *method* with no *qualifiers*.

unregistered package *n.* a *package object* that is not present in the *package registry*. An *unregistered package* has no *name*; *i.e.*, its *name* is **nil**. See the *function* **delete-package**.

unsafe *adj.* (of *code*) not *safe*. (Note that, unless explicitly specified otherwise, if a particular kind of error checking is guaranteed only in a *safe* context, the same checking might or might not occur in that context if it were *unsafe*; describing a context as *unsafe* means that certain kinds of error checking are not reliably enabled but does not guarantee that error checking is definitely disabled.)

unsafe call *n.* a *call* that is not a *safe call*. For more detailed information, see Section 3.5.1.1 (Safe and Unsafe Calls).

upgrade *v.t.* (a declared *type* to an actual *type*) 1. (when creating an *array*) to substitute an *actual array element type* for an *expressed array element type* when choosing an appropriately *specialized array* representation. See the *function* **upgraded-array-element-type**. 2. (when creating a *complex*) to substitute an *actual complex part type* for an *expressed complex part type* when choosing an appropriately *specialized complex* representation. See the *function* **upgraded-complex-part-type**.

upgraded array element type *n.* (of a *type*) a *type* that is a *supertype* of the *type* and that is used instead of the *type* whenever the *type* is used as an *array element type* for object creation or type discrimination. See Section 15.1.2.1 (Array Upgrading).

upgraded complex part type *n.* (of a *type*) a *type* that is a *supertype* of the *type* and that is used instead of the *type* whenever the *type* is used as a *complex part type* for object creation or type discrimination. See the *function* **upgraded-complex-part-type**.

uppercase *adj.* (of a *character*) being among *standard characters* corresponding to the capital letters `A` through `Z`, or being some other *implementation-defined character* that is defined by the *implementation* to be *uppercase*. See Section 13.1.4.3 (Characters With Case).

use *v.t.* (a *package* P_1) to *inherit* the *external symbols* of P_1. (If a package P_2 uses P_1, the *external symbols* of P_1 become *internal symbols* of P_2 unless they are explicitly *exported*.) "The package `CL-USER` uses the package `CL`."

use list *n.* (of a *package*) a (possibly empty) *list* associated with each *package* which determines what other *packages* are currently being *used* by that *package*.

user *n.* an active entity, typically a human, that invokes or interacts with a *program* at run time, but that is not necessarily a *programmer*.

V

valid array dimension *n.* a *fixnum* suitable for use as an *array dimension*. Such a *fixnum* must be greater than or equal to zero, and less than the *value* of **array-dimension-limit**. When multiple *array dimensions* are to be used together to specify a multi-dimensional *array*, there is also an implied constraint that the product of all of the *dimensions* be less than the *value* of **array-total-size-limit**.

valid array index *n.* (of an *array*) a *fixnum* suitable for use as one of possibly several indices needed to name an *element* of the *array* according to a multi-dimensional Cartesian coordinate system. Such a *fixnum* must be greater than or equal to zero, and must be less than the corresponding *dimension*$_1$ of the *array*. (Unless otherwise explicitly specified, the phrase "a *list* of *valid array indices*" further implies that the *length* of the *list* must be the same as the *rank* of the *array*.) "For a `2` by `3` array, valid array indices for the first dimension are `0` and `1`, and valid array indices for the second dimension are `0`, `1` and `2`."

valid array row-major index *n.* (of an *array*, which might have any number of *dimensions*$_2$) a single *fixnum* suitable for use in naming any *element* of the *array*, by viewing the array's storage as a linear series of *elements* in row-major order. Such a *fixnum* must be greater than or equal to zero, and less than the *array total size* of the *array*.

valid fill pointer *n.* (of an *array*) a *fixnum* suitable for use as a *fill pointer* for the *array*. Such a *fixnum* must be greater than or equal to zero, and less than or equal to the *array total size* of the *array*.

valid logical pathname host *n.* a *string* that has been defined as the name of a *logical host*. See the *function* **load-logical-pathname-translations**.

valid pathname device *n.* a *string*, **nil**, `:unspecific`, or some other *object* defined by the *implementation* to be a *valid pathname device*.

valid pathname directory *n.* a *string*, a *list* of *strings*, **nil**, `:wild`, `:unspecific`, or some other *object* defined by the *implementation* to be a *valid directory component*.

valid pathname host *n.* a *valid physical pathname host* or a *valid logical pathname host*.

valid pathname name *n.* a *string*, **nil**, `:wild`, `:unspecific`, or some other *object* defined by the *implementation* to be a *valid pathname name*.

valid pathname type *n.* a *string*, **nil**, `:wild`, `:unspecific`.

valid pathname version *n.* a non-negative *integer*, or one of `:wild`, `:newest`, `:unspecific`, or **nil**. The symbols `:oldest`, `:previous`, and `:installed` are *semi-standard* special version symbols.

valid physical pathname host *n.* any of a *string*, a *list* of *strings*, or the symbol `:unspecific`, that is recognized by the implementation as the name of a host.

valid sequence index *n.* (of a *sequence*) an *integer* suitable for use to name an *element* of the *sequence*. Such an *integer* must be greater than or equal to zero, and must be less than the *length* of the *sequence*. (If the *sequence* is an *array*, the *valid sequence index* is further constrained to be a *fixnum*.)

value *n.* 1. a. one of possibly several *objects* that are the result of an *evaluation*. b. (in a situation where exactly one value is expected from the *evaluation* of a *form*) the *primary value* returned by the *form*. c. (of *forms* in an *implicit progn*) one of possibly several *objects* that result from the *evaluation* of the last *form*, or **nil** if there are no *forms*. 2. an *object* associated with a *name* in a *binding*. 3. (of a *symbol*) the *value* of the *dynamic variable* named by that symbol. 4. an *object* associated with a *key* in an *association list*, a *property list*, or a *hash table*.

value cell *n.* *Trad.* (of a *symbol*) The *place* which holds the *value*, if any, of the *dynamic variable* named by that *symbol*, and which is *accessed* by **symbol-value**. See *cell*.

variable *n.* a *binding* in the "variable" *namespace*. See Section 3.1.2.1.1 (Symbols as Forms).

vector *n.* a one-dimensional *array*.

vertical-bar *n.* the *standard character* that is called "vertical bar" (`|`). See Figure 2–5.

W

whitespace *n.* 1. one or more *characters* that are either the *graphic character* `#\Space` or else *non-graphic* characters such as `#\Newline` that only move the print position. 2. a. *n.* the *syntax type* of a *character* that is a *token* separator. For details, see Section 2.1.4.7 (Whitespace Characters). b. *adj.* (of a *character*) having the *whitespace*$_{2a}$ *syntax type*$_2$. c. *n.* a *whitespace*$_{2b}$ *character*.

wild *adj.* 1. (of a *namestring*) using an *implementation-defined* syntax for naming files, which might "match" any of possibly several possible *filenames*, and which can therefore be used to refer to the aggregate of the *files* named by those *filenames*. 2. (of a *pathname*) a structured representation of a name which might "match" any of possibly several *pathnames*, and which can therefore be used to refer to the aggregate of the *files* named by those *pathnames*. The set of *wild pathnames* includes, but is not restricted to, *pathnames* which have a component which is `:wild`, or which have a directory component which contains `:wild` or `:wild-inferors`. See the *function* **wild-pathname-p**.

write *v.t.* 1. (a *binding* or *slot* or component) to change the *value* of the *binding* or *slot*. 2. (an *object* to a *stream*) to output a representation of the *object* to the *stream*.

writer *n.* a *function* that *writes*$_1$ a *variable* or *slot*.

Y

yield *v.t.* (*values*) to produce the *values* as the result of *evaluation*. "The form `(+ 2 3)` yields `5`."

Programming Language—Common Lisp

A. Appendix

A.1 Removed Language Features

A.1.1 Requirements for removed and deprecated features

For this standard, some features from the language described in *Common Lisp: The Language* have been removed, and others have been deprecated (and will most likely not appear in future Common Lisp standards). Which features were removed and which were deprecated was decided on a case-by-case basis by the X3J13 committee.

Conforming implementations that wish to retain any removed features for compatibility must assure that such compatibility does not interfere with the correct function of *conforming programs*. For example, symbols corresponding to the names of removed functions may not appear in the the `COMMON-LISP` *package*. (Note, however, that this specification has been devised in such a way that there can be a package named `LISP` which can contain such symbols.)

Conforming implementations must implement all deprecated features. For a list of deprecated features, see Section 1.8 (Deprecated Language Features).

A.1.2 Removed Types

The *type* `string-char` was removed.

A.1.3 Removed Operators

The functions `int-char`, `char-bits`, `char-font`, `make-char`, `char-bit`, `set-char-bit`, `string-char-p`, and `commonp` were removed.

The *special operator* `compiler-let` was removed.

A.1.4 Removed Argument Conventions

The *font* argument to **digit-char** was removed. The *bits* and *font* arguments to **code-char** were removed.

A.1.5 Removed Variables

The variables `char-font-limit`, `char-bits-limit`, `char-control-bit`, `char-meta-bit`, `char-super-bit`, `char-hyper-bit`, and `*break-on-warnings*` were removed.

A.1.6 Removed Reader Syntax

The "`#,`" *reader macro* in *standard syntax* was removed.

A.1.7 Packages No Longer Required

The *packages* `LISP`, `USER`, and `SYSTEM` are no longer required. It is valid for *packages* with one or more of these names to be provided by a *conforming implementation* as extensions.