Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**Programming Language—Common Lisp**

**12. Numbers**

Numbers **i**

Version 15.17R, X3J13/94-101R. Fri 12-Aug-1994 6:35pm EDT

**ii** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**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.**

Numbers **12–1**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

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

**12–2** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

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.**

Numbers **12–3**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**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.

**12–4** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**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 mathe matical 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 imple mentation 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)).

Numbers **12–5**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

| **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.

**12–6** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**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*.

Numbers **12–7**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**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*

**12–8** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**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*.

Numbers **12–9**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

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**

**12–10** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**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 |* **\***])

Numbers **12–11**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**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*]])

**12–12** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**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 · f · be−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 *bp−*1 and *bp −* 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*.

Numbers **12–13**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**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* log2*b*) and the exponent size (also measured in “bits,” computed as log2(*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:

**12–14** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

*•* 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*.

Numbers **12–15**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**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) **12–16** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**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.

Numbers **12–17**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**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**.

**12–18** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**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).

Numbers **12–19**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**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*. **12–20** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**=, /=,** *<***,** *>***,** *<***=,** *>***=**

<

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.

Numbers **12–21**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

| (= 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*

**12–22** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**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

Numbers **12–23**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

(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*.

**12–24** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**zerop** *Function*

**Syntax:**

**zerop** *number → generalized-boolean*

**Pronunciation:**

[ **z—e( )r—o( )p—e** ]

**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*

Numbers **12–25**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**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·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

**12–26** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

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

has the form *integer*+12), 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,

Numbers **12–27**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

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)

**12–28** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**asin, acos, atan**

**asin, acos, atan** *Function*

**Syntax:**

**asin** *number → radians*

**acos** *number → radians*

**atan** *number1* &optional *number2 → 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* log *ix* +1 *− x*2  Arc cosine (*π/*2) *−* arcsin *x*  Arc tangent *−i* log (1 + *ix*)p1*/*(1 + *x*2) |

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

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

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

For a *real number1*, the result is a *real* and lies between *−π/*2 and *π/*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:

Numbers **12–29**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**asin, acos, atan**

arcsin *z* = *−i* log

*iz* +p1 *− z*2

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

The following definition for arc cosine determines the range and branch cuts: arccos *z* =*π*2*−* arcsin *z*

or, which are equivalent,

arccos *z* = *−i* log

*z* + *i*p1 *− z*2

arccos *z* =2 log p(1 + *z*)*/*2 + *i*p(1 *− z*)*/*2

*i*

The branch cut for the arc cosine function is in two pieces: one along the negative real axis to the left of *−*1 (inclusive), continuous with quadrant II, and one along the positive real axis to the right of 1 (inclusive), continuous with quadrant IV. This is the same branch cut as for arc sine. The range is that strip of the complex plane containing numbers whose real part is between 0 and *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:

arctan *z* =log (1 + *iz*) *−* log (1 *− iz*)

2*i*

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 *−π/*2 and *π/*2. A number with real part equal to *−π/*2 is in the range if and only if its imaginary part is strictly positive; a number with real part equal to *π/*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 *−π/*2 and *π/*2 excluded. **12–30** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**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 < *π/*2 *y* > 0 *x* = 0 Positive y-axis *π/*2  *y* > 0 *x* < 0 Quadrant II *π/*2 < result < *π y* = 0 *x* < 0 Negative x-axis *π*  \* *y* = +0 *x* < 0 Negative x-axis +*π*  \* *y* = *−*0 *x* < 0 Negative x-axis *−π*  *y* < 0 *x* < 0 Quadrant III *−π* < result < *−π/*2 *y* < 0 *x* = 0 Negative y-axis *−π/*2  *y* < 0 *x* > 0 Quadrant IV *−π/*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 +*π*  \* *y* = *−*0 *x* = *−*0 Origin *−π* |

**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)

Numbers **12–31**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**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**.

**12–32** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

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

**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 (*ex − e−x*)*/*2  Hyperbolic cosine (*ex* + *e−x*)*/*2  Hyperbolic tangent (*ex − e−x*)*/*(*ex* + *e−x*)  Hyperbolic arc sine log (*x* +*√*1 + *x*~~2~~)  Hyperbolic arc cosine 2 log (p(*x* + 1)*/*2 + p(*x −* 1)*/*2)  Hyperbolic arc tangent (log (1 + *x*) *−* log (1 *− x*))*/*2 |

**Figure 12–16. Mathematical definitions for hyperbolic functions**

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

arccosh *z* = 2 log

p(*z* + 1)*/*2 + p(*z −* 1)*/*2*.*

The branch cut for the inverse hyperbolic cosine function lies along the real axis to the left of 1 (inclusive), extending indefinitely along the negative real axis, continuous with quadrant II and (between 0 and 1) with quadrant I. The range is that half-strip of the complex plane containing

numbers whose real part is non-negative and whose imaginary part is between *−π* (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).

Numbers **12–33**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

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

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

arcsinh *z* = log

*z* +p1 + *z*2 *.*

The branch cut for the inverse hyperbolic sine function is in two pieces: one along the positive imaginary axis above *i* (inclusive), continuous with quadrant I, and one along the negative imaginary axis below *−i* (inclusive), continuous with quadrant III. The range is that strip of the complex plane containing numbers whose imaginary part is between *−π/*2 and *π/*2. A number with imaginary part equal to *−π/*2 is in the range if and only if its real part is non-positive; a number with imaginary part equal to *π/*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: arctanh *z* =log (1 + *z*) *−* log (1 *− z*)

2*.*

Note that:

*i* arctan *z* = 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 *−π/*2 and *π/*2. A number with imaginary part equal to *−π/*2 is in the range if and only if its real part is strictly negative; a number with imaginary part equal to *π/*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 *−πi/*2 and *πi/*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.

**12–34** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

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)

Numbers **12–35**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**+** *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.

**12–36** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

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*. Numbers **12–37**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

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*.

**12–38** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**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

Numbers **12–39**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

(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* .

**12–40** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**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 *bx = exlogb*. 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, 00=1. If *b*=0 and the real part of *x* is strictly positive, then *bx*=0. For all other values of *x*, 0*x*is an error.

Numbers **12–41**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

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* .

**12–42** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

*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* .

Numbers **12–43**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**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.

**12–44** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

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.

Numbers **12–45**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**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 *−π* (exclusive) and *π* (inclusive) if minus zero is not supported, or *−π* (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.

**12–46** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**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

Numbers **12–47**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**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*

**12–48** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**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)

Numbers **12–49**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

(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)

**12–50** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**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\***

Numbers **12–51**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**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. **12–52** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**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.

Numbers **12–53**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**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*.

**12–54** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**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 *ei· 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)

Numbers **12–55**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**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)

**12–56** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**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. Numbers **12–57**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**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 *−π* (exclusive) if minus zero is not supported, or *−π* (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 *−π* (exclusive) and *π* (inclusive).

The mathematical definition of **phase** is as follows:

(phase *x*) = (atan (imagpart *x*) (realpart *x*))

**Examples:**

(phase 1) *→* 0.0s0

**12–58** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

(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**

Numbers **12–59**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**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*

**12–60** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

(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:**

**/**

Numbers **12–61**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**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**. **12–62** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**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)

Numbers **12–63**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**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·*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*))

**12–64** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**integer-length**

**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).

Numbers **12–65**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**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*.

**12–66** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

*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 #*n*R, 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* .

Numbers **12–67**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**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

**12–68** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**boole**

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

Numbers **12–69**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

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**

**12–70** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

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

**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.

Numbers **12–71**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**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

**12–72** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

*.* 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*.

Numbers **12–73**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**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

**12–74** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

(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*

Numbers **12–75**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

(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−*1through 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**

**12–76** Programming Language—Common Lisp

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**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*))

Numbers **12–77**

Version 15.17R, X3J13/94-101R.

Fri 12-Aug-1994 6:35pm EDT

**deposit-field** is to **mask-field** as **dpb** is to **ldb**.

**dpb** *Function*

**Syntax:**

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

**Pronunciation:**

[ **d****pib** ] or [ **d****p****b** ] or [ **d—e p—e b—e** ]

**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,

**12–78** Programming Language—Common Lisp