12. Numbers

12.1 Number Concepts

12.1.1 Numeric Operations

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

The next figure shows *operators* relating to arithmetic operations.

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

Figure 12-1. Operators relating to Arithmetic.

The next figure 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.

The next figure shows *operators* relating to numeric comparison and predication.

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

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

The next figure shows *defined names* relating to numeric type manipulation and coercion.

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

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

12.1.1.1 Associativity and Commutativity in Numeric Operations

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

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

12.1.1.1.1 Examples of Associativity and Commutativity in Numeric Operations

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

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

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

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

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

In any case, all three strategies are legitimate.

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

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

12.1.1.2 Contagion in Numeric Operations

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

12.1.1.3 Viewing Integers as Bits and Bytes

12.1.1.3.1 Logical Operations on Integers

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

The next figure shows *defined names* relating to logical operations on numbers.

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

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

12.1.1.3.2 Byte Operations on Integers

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

The next figure 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

The next figure shows *defined names* relating to *implementation-dependent* details about *numbers*.

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

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

12.1.3 Rational Computations

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

12.1.3.1 Rule of Unbounded Rational Precision

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

12.1.3.2 Rule of Canonical Representation for Rationals

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

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

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

12.1.3.3 Rule of Float Substitutability

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

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

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

```
Function  Sample Results
abs       (abs #c(3 4)) =>  5 or 5.0
acos      (acos 1) =>  0 or 0.0
acosh     (acosh 1) =>  0 or 0.0
asin      (asin 0) =>  0 or 0.0
asinh     (asinh 0) =>  0 or 0.0
atan      (atan 0) =>  0 or 0.0
atanh     (atanh 0) =>  0 or 0.0
cis       (cis 0) =>  1 or #c(1.0 0.0)
cos       (cos 0) =>  1 or 1.0
cosh      (cosh 0) =>  1 or 1.0
exp       (exp 0) =>  1 or 1.0
expt      (expt 8 1/3) =>  2 or 2.0
log       (log 1) =>  0 or 0.0
          (log 8 2) =>  3 or 3.0
phase     (phase 7) =>  0 or 0.0
signum    (signum #c(3 4)) =>  #c(3/5 4/5) or #c(0.6 0.8)
sin       (sin 0) =>  0 or 0.0
```

```
sinh      (sinh 0) =>  0 or 0.0
sqrt      (sqrt 4) =>  2 or 2.0
          (sqrt 9/16) =>  3/4 or 0.75
tan       (tan 0) =>  0 or 0.0
tanh      (tanh 0) =>  0 or 0.0
```

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

12.1.4 Floating-point Computations

The following rules apply to floating point computations.

12.1.4.1 Rule of Float and Rational Contagion

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

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

12.1.4.1.1 Examples of Rule of Float and Rational Contagion

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

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

12.1.4.3 Rule of Float Underflow and Overflow

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

12.1.4.4 Rule of Float Precision Contagion

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

12.1.5 Complex Computations

The following rules apply to *complex* computations:

12.1.5.1 Rule of Complex Substitutability

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

12.1.5.2 Rule of Complex Contagion

12.1.5.3 Rule of Canonical Representation for Complex Rationals

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

12.1.5.3.1 Examples of Rule of Canonical Representation for Complex Rationals

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

12.1.5.4 Principal Values and Branch Cuts

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

The next figure 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 the next figure.

[Quadrants image]

Figure 12-10. Quadrant Numbering for Branch Cuts

12.1.6 Interval Designators

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

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

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

a *singleton list* whose *element* is a number *M* of *type T*
: This denotes a lower exclusive bound of *M*. That is, *elements* of the *subtype* of *T* will be greater than *M*.

the symbol *****
: This denotes the absence of a lower bound on the interval.

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

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

a *singleton list* whose *element* is a number *M* of *type T*
: This denotes an upper exclusive bound of *M*. That is, *elements* of the *subtype* of *T* will be less than *M*.

the symbol *****
: This denotes the absence of an upper bound on the interval.

12.1.7 Random-State Operations

The next figure 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