7. Numbers

Zetalisp includes several types of numbers, with different characteristics. Most numeric functions accept any type of numbers as arguments and do the right thing. That is to say, they are *generic*. In Maclisp, there are generic numeric functions (like **plus**) and there are specific numeric functions (like +) which only operate on a certain type of number, but are much more efficient. In Zetalisp, this distinction does not exist: both function names exist for compatibility but they are identical. The microprogrammed structure of the machine makes it possible to have only the generic functions without loss of efficiency.

The types of numbers in Zetalisp are:

fixnum	Fixnums are 25-bit twos-complement binary integers. These are the preferred, most efficient type of number.
bignum	Bignums are arbitrary-precision binary integers.
ratio	Ratios represent rational numbers exactly as the quotient of two integers, each of which can be a fixnum or a bignum. Ratios with a denominator of one are not normally created, as an integer is returned instead.
single-float or full-size float	Full size floats are floating-point numbers. They have a mantissa of 31 bits and an exponent of 11 bits, providing a precision of about 9 digits and a range of about 10↑300. Stable rounding is employed.
short-float	Short floats are another form of floating-point number, with a mantissa of 17 bits and an exponent of 8 bits, providing a precision of about 5 digits and a range of about 10↑38. Stable rounding is employed. Short floats are useful because, like fixnums, and unlike full-size floats, they don't require any storage. Computing with short floats is more efficient than with full-size floats because the operations are faster and consing overhead is eliminated.
complexnum	Complexnums represent complex numbers with a real part and an imaginary part, which can be any type of number except complexnums. (They must be both rational or both floats of the same type). It is impossible to make a complexnum whose real part is rational and whose imaginary part is zero; it is always changed into a real number. However, it *is* possible to create complexnums with an imaginary part of 0.0, and such numbers may result from calculations involving complexnums. In fact, 5.0 and 5.0+0.0i are *always* distinct; they are not **eql**, and arithmetic operations will never canonicalize a complexnum with zero imaginary part into a real number.

Generally, Lisp objects have a unique identity; each exists, independent of any other, and you can use the **eq** predicate to determine whether two references are to the same object or not. Numbers are the exception to this rule; they don't work this way. The following function may return either **t** or **nil**. Its behavior is considered undefined; as this manual is written, it returns **t** when interpreted but **nil** when compiled.

```
(defun foo ()
  (let ((x (float 5)))
    (eq x (car (cons x nil)))))
```

This is very strange from the point of view of Lisp's usual object semantics, but the implementation works this way, in order to gain efficiency, and on the grounds that identity testing of numbers is not really an interesting thing to do. So the rule is that the result of applying **eq** to numbers is undefined, and may return either **t** or **nil** on what appear to be two pointers to the same numeric object. The only reasonable ways to compare numbers are **=** (see page 139) and **eql** (page 69), and other things (**equal** or **equalp**) based on them.

Conversely, fixnums and short floats have the unusual property that they are always **eq** if they are equal in value. This is because they do not point to storage: the "pointer" field of a fixnum is actually its numeric value, and likewise for short floats. Stylisticly it is better to avoid depending on this, by using **eql** rather than **eq**. Also, comparing floats of any sort for exact equality, even with **=** which is guaranteed to consider only the numeric values, is usually unwise since round-off error can make the answer unpredictable and meaningless.

The distinction between fixnums and bignums is largely transparent to the user. The user simply computes with integers, and the system represents some as fixnums and the rest (less efficiently) as bignums. The system automatically converts back and forth between fixnums and bignums based solely on the size of the integer. There are a few low level functions which only work on fixnums; this fact is noted in their documentation. Also, when using **eq** on numbers the user needs to be aware of the fixnum/bignum distinction.

Integer computations cannot overflow, except for division by zero, since bignums can be of arbitrary size. Floating-point computations can get exponent overflow or underflow, if the result is too large or small to be represented. Exponent overflow always signals an error. Exponent underflow normally signals an error, and assumes **0.0** as the answer if the user says to proceed from the error. However, if the value of the variable **zunderflow** is non-nil, the error is skipped and computation proceeds with **0.0** in place of the result that was too small.

When an arithmetic function of more than one argument is given arguments of different numeric types, uniform *coercion rules* are followed to convert the arguments to a common type, which is also the type of the result (for functions which return a number). When an integer meets a ratio, the result is a ratio. When an integer or ratio meets a float, the result is a float of the same sort. When a short-float meets a full-size float, the result is a full-size float.

If any argument of the arithmetic function is complex, the other arguments are converted to complex. The components of a complex result must be both full-size floats, both small-floats, or both rational; if they differ, the one whose type comes last in that list is converted to match the other. Finally, if the components of the result are rational and the imaginary part is zero, the result is simply the real part. If, however, the components are floats, the value is always complex even if the imaginary part is zero.

Thus if the constants in a numerical algorithm are written as short floats (assuming this provides adequate precision), and if the input is a short float, the computation is done with short floats and the result is a short float, while if the input is a full-size float the computation is done in full precision and the result is a full-size float.

Zetalisp never automatically converts between full-size floats and short floats in the same way as it automatically converts between fixnums and bignums since this would lead either to inefficiency or to unexpected numerical inaccuracies. (When a short float meets a full-size float, the result is a full-size float, but if you use only one type, all the results are of the same type too.) This means that a short float computation can get an exponent overflow error even when the result could have been represented as a full-size float.

Floating-point numbers retain only a certain number of bits of precision; therefore, the results of computations are only approximate. Full-size floats have 31 bits and short floats have 17 bits, not counting the sign. The method of approximation is "stable rounding". The result of an arithmetic operation is the float which is closest to the exact value. If the exact result falls precisely halfway between two representable floats, the result is rounded down if the least-significant bit is 0, or up if the least-significant bit is 1. This choice is arbitrary but insures that no systematic bias is introduced.

Unlike Maclisp, Zetalisp does not have number declarations in the compiler. Note that because fixnums and short floats require no associated storage they are as efficient as declared numbers in Maclisp. Bignums and full-size floats are less efficient; however, bignum and float intermediate results are garbage-collected in a special way that avoids the overhead of the full garbage collector.

The different types of numbers can be distinguished by their printed representations. If a number has an exponent separated by 's', it is a short float. If a number has an exponent separated by 'f', it is a full-size float. A leading or embedded (but *not* trailing) decimal point, and/or an exponent separated by 'e', indicates a float; which kind is controlled by the variable *read-default-float-format*, which is usually set to specify full-size floats. Short floats require a special indicator so that naive users will not accidentally compute with the lesser precision. Fixnums and bignums have similar printed representations since there is no numerical value that has a choice of whether to be a fixnum or a bignum; an integer is a bignum if and only if its magnitude is too big for a fixnum. See the examples on page 518, in the description of what the reader understands.

zunderflow *Variable*

When this is nil, floating point exponent underflow is an error. When this is t, exponent underflow proceeds, returning zero as the value. The same thing could be accomplished with a condition handler. However, zunderflow is useful for Maclisp compatibility, and is also faster.

sys:floating-exponent-overflow (sys:arithmetic-error error) *Condition*
sys:floating-exponent-underflow (sys:arithmetic-error error) *Condition*

sys:floating-exponent-overflow is signaled when the result of an arithmetic operation should be a floating point number, but the exponent is too large to be represented in the format to be used for the value. sys:floating-exponent-underflow is signaled when the exponent is too small.

The condition instance provides two additional operations: :function, which returns the arithmetic function that was called, and :small-float-p, which is t if the result was supposed to be a short float.

sys:floating-exponent-overflow provides the **:new-value** proceed type. It expects one argument, a new value.

sys:floating-exponent-underflow provides the **:use-zero** proceed type, which expects no argument.

Unfortunately, it is not possible to make the arguments to the operation available. Perhaps someday they will be.

7.1 Numeric Predicates

zerop *x*

Returns **t** if *x* is zero. Otherwise it returns **nil**. If *x* is not a number, **zerop** causes an error. For floats, this only returns **t** for exactly **0.0** or **0.0s0**. For complex numbers, it returns **t** if both real and imaginary parts are zero.

plusp *x*

Returns **t** if its argument is a positive number, strictly greater than zero. Otherwise it returns **nil**. If *x* is not a number, **plusp** causes an error.

minusp *x*

Returns **t** if its argument is a negative number, strictly less than zero. Otherwise it returns **nil**. If *x* is not a number, **minusp** causes an error.

oddp *number*

Returns **t** if *number* is odd, otherwise **nil**. If *number* is not a fixnum or a bignum, **oddp** causes an error.

evenp *number*

Returns **t** if *number* is even, otherwise **nil**. If *number* is not a fixnum or a bignum, **evenp** causes an error.

signp *test x* *Special form*

Tests the sign of a number. **signp** is present only for Maclisp compatibility and is not recommended for use in new programs. **signp** returns **t** if *x* is a number which satisfies the *test*, **nil** if it is not a number or does not meet the test. *test* is not evaluated, but *x* is. *test* can be one of the following:

l	$x < 0$
le	$x \le 0$
e	$x = 0$
n	$x \ne 0$
ge	$x \ge 0$
g	$x > 0$

Examples:

```
        (signp ge 12) => t
        (signp le 12) => nil
        (signp n 0) => nil
        (signp g 'foo) => nil
```

See also the data-type predicates **integerp**, **rationalp**, **realp**, **complexp**, **floatp**, **bigp**, **small-floatp**, and **numberp** (page 12).

7.2 Numeric Comparisons

All of these functions require that their arguments be numbers; they signal an error if given a non-number. Equality tests work on all types of numbers, automatically performing any required coercions (as opposed to Maclisp in which generally only the spelled-out names work for all kinds of numbers). Ordering comparisons allow only real numbers, since they are meaningless on complex numbers.

= &rest *numbers*

Returns **t** if all the arguments are numerically equal. They need not be of the same type; 1 and 1.0 are considered equal. Character objects are also allowed, and in effect coerced to integers for comparison.

See also **eql**, page 69, which insists that both the type and the value match when its arguments are numbers.

> &rest *numbers*
greaterp &rest *numbers*

> compares each pair of successive arguments. If any argument is not greater than the next, **>** returns **nil**. But if the arguments are monotonically strictly decreasing, the result is **t**. Zero arguments are always monotonically decreasing, and so is a single argument.
Examples:

```
        (>) => t
        (> 3) => t
        (> 4 3) => t
        (> 4 3 2 1 0) => t
        (> 4 3 1 2 0) => nil
```

greaterp is the Maclisp name for this function.

>= &rest *numbers*
≥ &rest *numbers*

≥ compares each pair of successive arguments. If any argument is less than the next, **≥** returns **nil**. But if the arguments are monotonically decreasing or equal, the result is **t**.

>= is the Common Lisp name for this function.

< &rest *numbers*
lessp &rest *numbers*

< compares each pair of successive arguments. If any argument is not less than the next, < returns nil. But if the arguments are monotonically strictly increasing, the result is t.
Examples:

```
(<) => t
(< 3) => t
(< 3 4) => t
(< 1 1) => nil
(< 0 1 2 3 4) => t
(< 0 1 3 2 4) => nil
```

lessp is the Maclisp name for this function.

<= &rest *numbers*
≤ &rest *numbers*

≤ compares its arguments from left to right. If any argument is greater than the next, ≤ returns nil. But if the arguments are monotonically increasing or equal, the result is t.

<= is the Common Lisp name for this function.

≠ &rest *numbers*
//= &rest *numbers*

t if no two arguments are numerically equal. This is the same as (not (= ...)) when there are two arguments, but not when there are more than two.

With zero or one argument, the value is always t, since there is no pair of arguments that fail to be equal.

//= is the Common Lisp name for this function. In Common Lisp syntax, it would be written /=.

max &rest *one-or-more-args*

Returns the largest of its arguments, which must not be complex.
Example:

```
(max 1 3 2) => 3
```

max requires at least one argument.

min &rest *one-or-more-args*

Returns the smallest of its arguments, which must not be complex.
Example:

```
(min 1 3 2) => 1
```

min requires at least one argument.

7.3 Arithmetic

All of these functions require that their arguments be numbers, and signal an error if given a non-number. They work on all types of numbers, automatically performing any required coercions (as opposed to Maclisp, in which generally only the spelled-out versions work for all kinds of numbers, and the '$' versions are needed for floats).

+ &rest *args*
plus &rest *args*
+$ &rest *args*

Returns the sum of its arguments. If there are no arguments, it returns 0, which is the identity for this operation.

plus and $+ are Maclisp names, supported for compatibility.

- *arg* &rest *args*
-$ *arg* &rest *args*

With only one argument, - returns the negative of its argument. With more than one argument, - returns its first argument minus all of the rest of its arguments.

Examples:

```
(- 1) => -1
(- -3.0) => 3.0
(- 3 1) => 2
(- 9 2 1) => 6
```

-$ is a Maclisp name, supported for compatibility.

minus *x*

Returns the negative of *x*, just like - with one argument.

difference *arg* &rest *args*

Returns its first argument minus all of the rest of its arguments. If there are at least two arguments, difference is equivalent to -.

abs *x*

Returns |*x*|, the absolute value of the number *x*. abs for real numbers could have been defined as

```
(defun abs (x)
    (cond ((minusp x) (minus x))
          (t x)))
```

abs of a complex number could be computed, though imprecisely, as

```
(sqrt (^ (realpart x) 2) (^ (imagpart x) 2))
```

***** &rest *args*
times &rest *args*
***$** &rest *args*

Returns the product of its arguments. If there are no arguments, it returns 1, which is the identity for this operation.

times and *$ are Maclisp names, supported for compatibility.

// *arg* &rest *args*
//$ *arg* &rest *args*

With more than one argument, // it returns the first argument divided by all of the rest of its arguments. With only one argument, (// *x*) is the same as (// 1 *x*).

The name of this function is written // rather than / because / is the escape character in traditional Lisp syntax and must be escaped in order to suppress that significance. //$ is a Maclisp name, supported for compatibility.

// of two integers returns an integer even if the mathematically correct value is not an integer. More precisely, the value is the same as the first value returned by truncate (see below). This will eventually be changed, and then the value will be a ratio if necessary so that the it is mathematically correct. All code that relies on // to return an integer value rather than a ratio should be converted to use truncate (or floor or ceiling, which may simplify the code further). In the mean time, use the function cli:// if you want a rational result.

Examples:
```
(// 3 2) => 1           ;Fixnum division truncates.
(// 3 -2) => -1
(// -3 2) => -1
(// -3 -2) => 1
(// 3 2.0) => 1.5
(// 3 2.0s0) => 1.5s0
(// 4 2) => 2
(// 12. 2. 3.) => 2
(// 4.0) => .25
```

quotient *arg* &rest *args*

Returns the first argument divided by all of the rest of its arguments. When there are two or more arguments, quotient is equivalent to //.

cli:// *number* &rest *numbers*

This is the Common Lisp division function. It is like // except that it uses exact rational division when the arguments are integers.

// will someday be changed to divide integers exactly. Then there will no longer be a distinct function cli://; that name will become equivalent to //.

Note that in Common Lisp syntax you would write just / rather than cli://.

There are four functions for "integer division", the sort which produces a quotient and a remainder. They differ in how they round the quotient to an integer, and therefore also in the sign of the remainder. The arguments must be real, since ordering is needed to compute the value. The quotient is always an integer, but the arguments and remainder need not be.

floor *x* &optional (*y* 1)

floor's first value is the largest integer less than or equal to the quotient of *x* divided by *y*.

The second value is the remainder, *x* minus *y* times the first value. This has the same sign as *y* (or may be zero), regardless of the sign of *x*.

With one argument, floor's first value is the largest integer less than or equal to the argument.

ceiling *x* &optional (*y* 1)

ceiling's first value is the smallest integer greater than or equal to the quotient of *x* divided by *y*.

The second value is the remainder, *x* minus *y* times the first value. This has the opposite sign from *y* (or may be zero), regardless of the sign of *x*.

With one argument, ceiling's first value is the smallest integer greater than or equal to the argument.

truncate *x* &optional (*y* 1)

truncate is the same as floor if the arguments have the same sign, ceiling if they have opposite signs. truncate is the function that the divide instruction on most computers implements.

truncate's first value is the nearest integer, in the direction of zero, to the quotient of *x* divided by *y*.

The second value is the remainder, *x* minus *y* times the first value. This has the same sign as *x* (or may be zero).

round *x* &optional (*y* 1)

round's first value is the nearest integer to the quotient of *x* divided by *y*. If the quotient is midway between two integers, the even integer of the two is used.

The second value is the remainder, *x* minus *y* times the first value. The sign of this remainder cannot be predicted from the signs of the arguments alone.

With one argument, round's first value is the integer nearest to the argument.

Here is a table which clarifies the meaning of floor, ceiling, truncate and round with one argument:

	floor	ceiling	truncate	round
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
-2.5	-3	-2	-2	-2
-2.6	-3	-2	-2	-3

There are two kinds of remainder function, which differ in the treatment of negative numbers. The remainder can also be obtained as the second value of one of the integer division functions above, but if only the remaineder is desired it is simpler to use these functions.

\ *x y*
remainder *x y*
cli:rem *x y*

Returns the remainder of *x* divided by *y*. *x* and *y* must be integers (fixnums or bignums). This is the same as the second value of (**truncate** *x y*). Only the absolute value of the divisor is relevant.

```
(\ 3 2) => 1
(\ -3 2) => -1
(\ 3 -2) => 1
(\ -3 -2) => -1
```

Common Lisp gives this function the name **rem**, but since **rem** in traditional Zetalisp is a function to remove elements from lists (see page 105), the name **rem** is defined to mean remainder only in Common Lisp programs. Note that the name \ would have to be written as \\ in Common Lisp syntax; but the function \ is not standard Common Lisp.

mod *number divisor*

Returns the root of *number* modulo *divisor*. This is a number between 0 and *divisor*, or possibly 0, whose difference from *number* is a multiple of *divisor*. It is the same as the second value of (**floor** *number divisor*). Examples:

```
(mod 2 5)  =>  2
(mod -2 5)  =>  3
(mod -2 -5)  =>  -2
(mod 2 -5)  =>  -3
```

There are four "floating point integer division" functions. These produce a result which is a floating point number whose value is exactly integral.

ffloor *x* &optional (*y* 1)
fceiling *x* &optional (*y* 1)
ftruncate *x* &optional (*y* 1)
fround *x* &optional (*y* 1)

Like floor, ceiling, truncate and round except that the first value is converted from an integer to a float. If *x* is a float, then the result is the same type of float as *x*.

sys:divide-by-zero (sys:arithmetic-error error) *Condition*

Dividing by zero, using any of the above division functions, signals this condition. The :function operation on the condition instance returns the name of the division function. The :dividend operation may be available to return the number that was divided.

1+ *x*
add1 *x*
1+$ *x*

(1+ x) is the same as (+ x 1). The other two names are for Maclisp compatibility.

1- *x*
sub1 *x*
1-$ *x*

(1- x) is the same as (- x 1). Note that the short name may be confusing: (1- *x*) does *not* mean 1-*x*; rather, it means *x*-1. The names sub1 and 1-$ are for Maclisp compatibility.

gcd &rest *integers*
\ &rest *integers*

Returns the greatest common divisor of all its arguments, which must be integers. With one argument, the value is that argument. With no arguments, the value is zero.

In Common Lisp syntax \\ would be written as \\\\, but only the name gcd is valid in strict Common Lisp.

lcm *integer* &rest *more-integers*

Returns the least common multiple of the specified integers.

expt *x y*
^ *x y*
^$ *x y*

Returns *x* raised to the *y*'th power. The result is rational (and possibly an integer) if *x* is rational and *y* an integer. If the exponent is an integer a repeated-squaring algorithm is used; otherwise the result is (exp (* *y* (log *x*))).

If *y* is zero, the result is (+ 1 (* *x y*)); this is equal to one, but its type depends on those of *x* and *y*.

sys:zero-to-negative-power (sys:arithmetic-error error) *Condition*

This condition is signaled when expt's first argument is zero and the second argument is negative.

sqrt *x*

Returns the square root of *x*. A mathematically unavoidable discontinuity occurs for negative real arguments, for which the value returned is a positive real times i.

```
(sqrt 4) => 2
(sqrt -4) => 0+2i
(sqrt -4+.0001i) => .00005+2i (approximately)
(sqrt -4-.0001i) => .00005-2i (approximately)
```

isqrt *x*

Integer square-root. *x* must be an integer; the result is the greatest integer less than or equal to the exact square root of *x*.

***dif** *x y*
***plus** *x y*
***quo** *x y*
***times** *x y*

These are the internal microcoded arithmetic functions. There is no reason why anyone should need to write code with these explicitly, since the compiler knows how to generate the appropriate code for **plus**, **+**, etc. These names are only here for Maclisp compatibility.

%div *dividend divisor*

The internal division function used by **cli://**, it was available before **cli://** was and may therefore be used in some programs. It takes exactly two arguments. Uses of **%div** should be changed to use **cli://**.

7.4 Complex Number Functions

See also the predicates **realp** and **complexp** (page 12).

complex *x* &optional *y*

Returns the complex number whose real part is *x* and whose imaginary part is *y*.

If *x* is rational and *y* is zero or omitted, the value is *x*, and not a complex number at all. If *x* is a float and *y* is zero or omitted, of if *y* is a floating zero, the result is a complexnum whose imaginary part is zero.

realpart *z*

Returns the real part of the number *z*. If *z* is real, this is the same as *z*.

imagpart *z*

Returns the imaginary part of the number *z*. If *z* is real, this is zero.

conjugate *z*

Returns the complex conjugate of the number *z*. If *z* is real, this is the same as *z*.

phase *z*

Returns the phase angle of the complex number *z* in its polar form. This is the angle from the positive *x* axis to the ray from the origin through *z*. The value is always in the interval (-π, π].

```
(phase -4) => π
(phase -4-.0001i) is just over -π.
(phase 0) => 0 (an arbitrary choice)
```

cis *angle*

Returns the complex number of unit magnitude whose phase is *angle*. This is equal to (complex (cos *angle*) (sin *angle*)). *angle* must be real.

signum *z*

Returns a number with unit magnitude and the same type and phase as *z*. If *z* is zero, the value is zero.

If *z* is real, the value is = to 1 or -1; it may be a float, however.

7.5 Transcendental Functions

These functions are only for floating-point arguments; if given an integer they convert it to a float. If given a short float, they return a short float.

pi *Constant*

The value of π, as a full-size float.

exp *x*

Returns *e* raised to the *x*'th power, where *e* is the base of natural logarithms.

log *x* &optional *base*

Returns the logarithm of *x* to base *base*. *base* defaults to *e*. When *base* is *e*, the imaginary part of the value is in the interval (-π, π]; for negative real *x*, the value has imaginary part π.

If *base* is specified, the result is

```
(// (log x) (log base))
```

sys:zero-log (sys:arithmetic-error error) *Condition*

This is signaled when the argument to log is zero.

sin *x*
cos *x*
tan *x*

Return, respectively, the sine, cosine and tangent of *x*, where *x* is expressed in radians. *x* may be complex.

sind *x*
cosd *x*
tand *x*

Return, respectively, the sine, cosine and tangent of *x*, where *x* is expressed in degrees.

asin *x*
acos *x*

Returns the angle (in radians) whose sine (respectively, cosine) is *x*. The real part of the result of asin is between -π/2 and π/2; acos and asin of any given argument always add up to π/2.

atan *y* &optional *x*

If only *y* is given, the value is the angle, in radians, whose tangent is *y*. The real part of the result is between zero and -π.

If *x* is also given, both arguments must be real, and the value is an angle, in radians, whose tangent is *y*/*x*. However, the signs of the two arguments are used to choose between two angles which differ by π and have the same tangent. The one chosen is the angle from the *x*-axis counterclockwise to the line from the origin to the point (*x*, *y*).

atan always returns a non-negative number between zero and 2π.

atan2 *y* &optional *x*
cli:atan *y* &optional *x*

Like atan but always returns a value whose real part is between -π/2 and π/2. The value is either the same as the value of atan or differs from it by π.

atan2 is the traditional name of this function. In Common Lisp it is called atan; it is documented as cli:atan since the name atan has a different meaning in traditional syntax.

sinh *x*
cosh *x*
tanh *x*
asinh *x*
acosh *x*
atanh *x*

The hyperbolic and inverse hyperbolic functions.

7.6 Numeric Type Conversions

These functions are provided to allow specific conversions of data types to be forced, when desired.

`float` *number* &optional *float*

Converts *number* to a floating point number and returns it.

If *float* is specified, the result is of the same floating point format as *float*. If *number* is a float of a different format then it is converted.

If *float* is omitted, then *number* is converted to a single-float unless it is already a floating point number.

A complex number is converted to one whose real and imaginary parts are full-size floats unless they are already both floats.

`small-float` *x*
`short-float` *x*

Converts any kind of real number to a short-float. A complex number is converted to one whose real and imaginary parts are short floats. The two names are synonymous.

`numerator` *x*

Returns the numerator of the rational number *x*. If *x* is an integer, the value equals *x*. If *x* is not an integer or ratio, an error is signaled.

`denominator` *x*

Returns the denominator of the rational number *x*. If *x* is an integer, the value is `1`. If *x* is not an integer or ratio, an error is signaled.

`rational` *x*

Converts *x* to a rational number. If *x* is an integer or a ratio, it is returned unchanged. If it is a floating point number, it is regarded as an exact fraction whose numerator is the mantissa and whose denominator is a power of two. For any other argument, an error is signaled.

`rationalize` *x* &optional *precision*

Returns a rational approximation to *x*.

If there is only one argument, and it is an integer or a ratio, it is returned unchanged. If the argument is a floating point number, a rational number is returned which, if converted to a floating point number, would produce the original argument. Of all such rational numbers, the one chosen has the smallest numerator and denominator.

If there are two arguments, the second one specifies how much precision of the first argument should be considered significant. *precision* can be a positive integer (the number of bits to use), a negative integer (the number of bits to drop at the end), or a floating point number (minus its exponent is the number of bits to use).

If there are two arguments and the first is rational, the value is a "simpler" rational which approximates it.

fix *x*

Converts *x* from a float or ratio to an integer, truncating towards negative infinity. The result is a fixnum or a bignum as appropriate. If *x* is already a fixnum or a bignum, it is returned unchanged.

fix is the same as floor except that floor returns an additional value. fix is semi-obsolete, since the functions floor, ceiling, truncate and round provide four different ways of converting numbers to integers with different kinds of rounding.

fixr *x*

fixr is the same as round except that round returns an additional value. fixr is considered obsolete.

7.7 Floating Point Numbers

decode-float *float*

Returns three values which describe the value of *float*.

The first value is a positive float of the same format having the same mantissa, but with an exponent chosen to make it between 1/2 and 1, less than 1.

The second value is the exponent of *float*: the power of 2 by which the first value needs to be scaled in order to get *float* back.

The third value expresses the sign of *float*. It is a float of the same format as *float*, whose value is either 1 or -1. Example:

```
(decode-float 38.2)
  =>  0.596875   6   1.0
```

integer-decode-float *float*

Like decode-float except that the first value is scaled so as to make it an integer, and the second value is modified by addition of a constant to compensate.

```
(integer-decode-float 38.2)
  =>  #o11431463146   -25.   1.0
```

scale-float *float integer*

Multiplies *float* by 2 raised to the *integer* power. *float* can actually be an integer also; it is converted to a float and then scaled.

```
(scale-float 0.596875 6)  =>  38.2
(scale-float #o11431463146 -25.)  =>  38.2
```

float-sign *float1* &optional *float2*

Returns a float whose sign matches that of *float1* and whose magnitude and format are those of *float2*. If *float2* is omitted, 1.0 is used as the magnitude and *float1* 's format is used.

```
        (float-sign -1.0s0 35.3)  =>  -35.3
        (float-sign -1.0s0 35.3s0)  =>  -35.3s0
```

float-radix *float*

Defined by Common Lisp to return the radix used for the exponent in the format used for *float*. On the Lisp Machine, floating point exponents are always powers of 2, so float-radix ignores its argument and always returns 2.

float-digits *float*

Returns the number of bits of mantissa in the floating point format which float is an example of. It is 17 for short floats and 31 for full size ones.

float-precision *float*

Returns the number of significant figures present in in the mantissa of *float*. This is always the same as (float-digits *float*) for normalized numbers, and on the Lisp Machine all floats are normalized, so the two functions are the same.

7.8 Logical Operations on Numbers

Except for lsh and rot, these functions operate on both fixnums and bignums. lsh and rot have an inherent word-length limitation and hence only operate on 25-bit fixnums. Negative numbers are operated on in their 2's-complement representation.

logior &rest *integers*

Returns the bit-wise logical *inclusive or* of its arguments. With no arguments, the value is zero, which is the identity for this operation.
Example (in octal):

```
        (logior #o4002 #o67) => #o4067
```

logand &rest *integers*

Returns the bit-wise logical *and* of its arguments. With no arguments, the value is -1, which is the identity for this operation.
Examples (in octal):

```
        (logand #o3456 #o707) => #o406
        (logand #o3456 #o-100) => #o3400
```

logxor &rest *integers*

Returns the bit-wise logical *exclusive or* of its arguments. With no arguments, the value is zero, which is the identity for this operation.
Example (in octal):

```
        (logxor #o2531 #o7777) => #o5246
```

logeqv &rest *integers*

Combines the *integers* together bitwise using the equivalence operation, which, for two arguments, is defined to result in 1 if the two argument bits are equal. This operation is asociative. With no args, the value is -1, which is an identity for the equivalence operation.
Example (in octal):
```
        (logeqv #o2531 #o7707) => #o-5237 = ...77772541
```

Non-associative bitwise operations take only two arguments:

lognand *integer1 integer2*

Returns the bitwise-nand of the two arguments. A bit of the result is 1 if at least one of the corresponding argument bits is 0.

lognor *integer1 integer2*

Returns the bitwise-nor of the two arguments. A bit of the result is 1 if both of the corresponding argument bits are 0.

logorc1 *integer1 integer2*

Returns the bitwise-or of *integer2* with the complement of *integer1*.

logorc2 *integer1 integer2*

Returns the bitwise-or of *integer1* with the complement of *integer2*.

logandc1 *integer1 integer2*

Returns the bitwise-and of *integer2* with the complement of *integer1*.

logandc2 *integer1 integer2*

Returns the bitwise-and of *integer1* with the complement of *integer2*.

lognot *number*

Returns the logical complement of *number*. This is the same as logxor'ing *number* with -1.
Example:
```
        (lognot #o3456) => #o-3457
```

boole *fn* &rest *one-or-more-args*

boole is the generalization of logand, logior, and logxor. *fn* should be a fixnum between 0 and 17 octal inclusive; it controls the function which is computed. If the binary representation of *fn* is *abcd* (*a* is the most significant bit, *d* the least) then the truth table for the Boolean operation is as follows:

```
          y
       | 0  1
    ---------
      0| a  c
    x  |
      1| b  d
```

If **boole** has more than three arguments, it is associated left to right; thus,

```
        (boole fn x y z) = (boole fn (boole fn x y) z)
```

With two arguments, the result of **boole** is simply its second argument. At least two arguments are required.

Examples:

```
        (boole 1 x y) = (logand x y)
        (boole 6 x y) = (logxor x y)
        (boole 2 x y) = (logand (lognot x) y)
```

logand, **logior**, and so on are usually preferred over the equivalent forms of **boole**. **boole** is useful when the operation to be performed is not constant.

boole-ior *Constant*
boole-and *Constant*
boole-xor *Constant*
boole-eqv *Constant*
boole-nand *Constant*
boole-nor *Constant*
boole-orc1 *Constant*
boole-orc2 *Constant*
boole-andc1 *Constant*
boole-andc2 *Constant*

The **boole** opcodes that correspond to the functions **logior**, **logand**, etc.

boole-clr *Constant*
boole-set *Constant*
boole-1 *Constant*
boole-2 *Constant*

The **boole** opcodes for the four trivial operations. Respectively, they are those which always return zero, always return one, always return the first argument, and always return the second argument.

bit-test *x y*
logtest *x y*

bit-test is a predicate which returns **t** if any of the bits designated by the 1's in *x* are 1's in *y*. **bit-test** is implemented as a macro which expands as follows:

```
        (bit-test x y) ==> (not (zerop (logand x y)))
```

logtest is the Common Lisp name for this function.

lsh *x y*

Returns *x* shifted left *y* bits if *y* is positive or zero, or *x* shifted right |*y*| bits if *y* is negative. Zero bits are shifted in (at either end) to fill unused positions. *x* and *y* must be fixnums. (In some applications you may find **ash** useful for shifting bignums; see below.)

Examples:
```
        (lsh 4 1) => #o10
        (lsh #o14 -2) => 3
        (lsh -1 1) => -2
```

ash *x y*
Shifts *x* arithmetically left *y* bits if *y* is positive, or right -*y* bits if *y* is negative. Unused positions are filled by zeroes from the right, and by copies of the sign bit from the left. Thus, unlike **lsh**, the sign of the result is always the same as the sign of *x*. If *x* is a fixnum or a bignum, this is a shifting operation. If *x* is a float, this does scaling (multiplication by a power of two), rather than actually shifting any bits.

rot *x y*
Returns *x* rotated left *y* bits if *y* is positive or zero, or *x* rotated right |*y*| bits if *y* is negative. The rotation considers *x* as a 25-bit number (unlike Maclisp, which considers *x* to be a 36-bit number in both the pdp-10 and Multics implementations). *x* and *y* must be fixnums. (There is no function for rotating bignums.)
Examples:
```
        (rot 1 2) => 4
        (rot 1 -2) => #o20000000
        (rot -1 7) => -1
        (rot #o15 25.) => #o15
```

logcount *integer*
Returns the number of 1 bits in *integer*, if it is positive. Returns the number of 0 bits in *integer*, if it is negative. (There are infinitely many 1 bits in a negative integer.)
```
        (logcount #o15)  =>  3
        (logcount #o-15)  =>  2
```

integer-length *integer*
The minimum number of bits (aside from sign) needed to represent *integer* in two's complement. This is the same as **haulong** for positive numbers.
```
        (integer-length 0) => 0
        (integer-length 7) => 3
        (integer-length 8) => 4
        (integer-length -7) => 3
        (integer-length -8) => 3
        (integer-length -9) => 4
```

haulong *integer*
The same as **integer-length** of the absolute value of **integer**. This name exists for Maclisp compatibility only.

haipart *x n*
Returns the high *n* bits of the binary representation of |*x*|, or the low -*n* bits if *n* is negative. *x* may be a fixnum or a bignum; its sign is ignored. **haipart** could have been defined by:

```
(defun haipart (x n)
  (setq x (abs x))
  (if (minusp n)
      (logand x (1- (ash 1 (- n))))
      (ash x (min (- n (haulong x))
                  0))))
```

7.9 Byte Manipulation Functions

Several functions are provided for dealing with an arbitrary-width field of contiguous bits appearing anywhere in an integer (a fixnum or a bignum). Such a contiguous set of bits is called a *byte*. Note that we are not using the term *byte* to mean eight bits, but rather any number of bits within a number. These functions use numbers called *byte specifiers* to designate a specific byte position within any word. A byte specifier contains two pieces of information: the size of the byte, and the position of the byte. The position is expressed as the number of least significant bits which are not included in the byte. A position of zero means that the byte is at the right (least significant) end of the number.

The maximum value of the size is 24, since a byte must fit in a fixnum although bytes can be loaded from and deposited into bignums. (Bytes are always positive numbers.)

Byte specifiers are represented as fixnums whose two lowest octal digits represent the *size* of the byte, and whose higher (usually two, but sometimes more) octal digits represent the *position* of the byte within a number. For example, the byte-specifier #o0010 (i.e. 10 octal) refers to the lowest eight bits of a word, and the byte-specifier #o1010 refers to the next eight bits. The format of byte-specifiers is taken from the pdp-10 byte instructions.

Much old code contains byte specifiers written explicitly as octal numbers. It is cleaner to construct byte specifiers using **byte** instead. Decomposition of byte specifiers should always be done with **byte-position** and **byte-size**, as at some time in the future other kinds of byte specifiers may be created to refer to fields whose size is greater than #o77.

byte *size position*

Returns a byte specifier for the byte of *size* bits, positioned to exclude the *position* least significant bits. This byte specifier can be passed as the first argument to **ldb**, **dpb**, **%logldb**, **%logdpb**, **mask-field**, **%p-ldb**, **%p-ldb-offset**, and so on.

byte-position *byte-spec*
byte-size *byte-spec*

Return, respectively, the size and the position of *byte-spec*. It is always true that
`(byte (byte-size` *byte-spec*`) (byte-position` *byte-spec*`))`
equals *byte-spec*.

ldb *byte-spec integer*

Extracts a byte from *integer* according to *byte-soec*. The contents of this byte are returned right-justified in a fixnum. The name of the function, **ldb**, means 'load byte'. *integer* may be a fixnum or a bignum. The returned value is always a fixnum.

Example:
```
        (ldb (byte 6 3) #o4567) => #o56
```

load-byte *integer position size*
This is like ldb except that instead of using a byte specifier, the *position* and *size* are passed as separate arguments. The argument order is not analogous to that of ldb so that load-byte can be compatible with Maclisp.

ldb-test *byte-spec integer*
ldb-test is a predicate which returns t if any of the bits designated by the byte specifier *byte-spec* are 1's in *integer*. That is, it returns t if the designated field is non-zero. ldb-test is implemented as a macro which expands as follows:
```
        (ldb-test byte-spec integer) ==> (not (zerop (ldb byte-spec integer)))
```

logbitp *index integer*
t if the bit *index* up from the least significant in *integer* is a 1. This is equivalent to (ldb-test (byte *index* 1) *integer*).

mask-field *byte-spec fixnum*
This is similar to ldb; however, the specified byte of *fixnum* is positioned in the same byte of the returned value. The returned value is zero outside of that byte. *fixnum* must be a fixnum.
Example:
```
        (mask-field (byte 6 3) #o4567) => #o560
```

dpb *byte byte-spec integer*
Returns a number which is the same as *integer* except in the bits specified by *byte-spec*. The low bits of *byte*, appropriately many, are placed in those bits. *byte* is interpreted as being right-justified, as if it were the result of ldb. *integer* may be a fixnum or a bignum. The name means 'deposit byte'.
Example:
```
        (dpb #o23 (byte 6 3) #o4567) => #o4237
```

deposit-byte *integer position size byte*
This is like dpb except that instead of using a byte specifier, the *position* and *size* are passed as separate arguments. The argument order is not analogous to that of dpb so that deposit-byte can be compatible with Maclisp.

deposit-field *byte byte-spec fixnum*
This is like dpb, except that *byte* is not taken to be left-justified; the *byte-spec* bits of *byte* are used for the *byte-spec* bits of the result, with the rest of the bits taken from *fixnum*. *fixnum* must be a fixnum.
Example:
```
        (deposit-field #o230 (byte 6 3) #o4567) => #o4237
```

The behavior of the following two functions depends on the size of fixnums, and so functions using them may not work the same way on future implementations of Zetalisp. Their names start with % because they are more like machine-level subprimitives than the previous functions.

%logldb *byte-spec fixnum*

%logldb is like ldb except that it only loads out of fixnums and allows a byte size of 25, i.e. all 25 bits of the fixnum including the sign bit.

%logdpb *byte byte-spec fixnum*

%logdpb is like dpb except that it only deposits into fixnums. Using this to change the sign-bit leaves the result as a fixnum, while dpb would produce a bignum result for arithmetic correctness. %logdpb is good for manipulating fixnum bit-masks such as are used in some internal system tables and data-structures.

7.10 Random Numbers

The functions in this section provide a pseudo-random number generator facility. The basic function you use is random, which returns a new pseudo-random number each time it is called.

random &optional *number random-state*

Returns a randomly generated number. If *number* is specified, the random number is of the same type as *number* (floating if *number* is floating, etc.), nonnegative, and less than *number*.

If *number* is omitted, the result is a randomly chosen fixnum, with all fixnums being equally likely.

If *random-state* is present, it is used and updated in generating the random number. Otherwise, the default random-state (the value of *random-state*) is used (and is created if it doesn't already exist). The algorithm is executed inside a without-interrupts (see page 684) so two processes can use the same random-state without colliding.

si:random-in-range *low high*

Returns a random float in the interval [*low*, *high*). The default random-state is used.

A *random-state* is a named structure of type random-state whose contents control the future actions of the random number generator. Each time you call the function random, it uses (and updates) one random-state. One random-state exists standardly and is used by default. To have several different controllable, resettable sources of random numbers, you can create your own random-states. Random-states print as

#s(random-state ...*more data*...)

so that they can be read back in.

random-state-p *object*

t if *object* is a random-state.

random-state *Variable*

This random-state is used by default when random is called and the random-state is not explicitly specified.

make-random-state &optional *random-state*

Creates and returns a new random-state object. If *random-state* is nil, the new random-state is a copy of *random-state*. If *random-state* is a random-state, the new one is a copy of that one. If *random-state* is t, the new random-state is initialized truly randomly (based on the value of (time)).

A random-state actually consists of an array of numbers and two pointers into the array. The pointers circulate around the array; each time a random number is requested, both pointers are advanced by one, wrapping around at the end of the array. Thus, the distance forward from the first pointer to the second pointer stays the same, allowing for wraparound. Let the length of the array be *length* and the distance between the pointers be *offset*. To generate a new random number, each pointer is set to its old value plus one, modulo *length*. Then the two elements of the array addressed by the pointers are added together; the sum is stored back into the array at the location where the second pointer points, and is returned as the random number after being normalized into the right range.

This algorithm produces well-distributed random numbers if *length* and *offset* are chosen carefully, so that the polynomial *x* ^ *length* + *x* ^ *offset* + 1 is irreducible over the mod-2 integers. The system uses 71. and 35.

The contents of the array of numbers should be initialized to anything moderately random, to make the algorithm work. The contents get initialized by a simple random number generator, based on a number called the *seed*. The initial value of the seed is set when the random-state is created, and it can be changed.

si:random-create-array *length offset seed* &optional (*area* nil)

Creates and returns a new random-state according to precise specifications. *length* is the length of the array. *offset* is the distance between the pointers and should be an integer less than *length*. *seed* is the initial value of the seed, and should be a fixnum. This calls si:random-initialize on the random state before returning it.

si:random-initialize *random-state* &optional *new-seed*

random-state must be a random-state, such as is created by si:random-create-array. If *new-seed* is provided, it should be a fixnum, and the seed is set to it. si:random-initialize reinitializes the contents of the array from the seed (calling random changes the contents of the array and the pointers, but not the seed).

7.11 Information on Numeric Precision

Common Lisp defines some constants whose values give information in a standard way about the ranges of numbers representable in the individual Lisp implementation.

most-negative-fixnum *Constant*

Any integer smaller than this must be a bignum.

most-positive-fixnum *Constant*
Any integer larger than this must be a bignum.

most-positive-short-float *Constant*
No short float can be greater than this number.

least-positive-short-float *Constant*
No positive short float can be closer to zero than this number.

least-negative-short-float *Constant*
No negative short float can be closer to zero than this number.

most-negative-short-float *Constant*
No short float can be less than this (negative) number.

most-positive-single-float *Constant*
least-positive-single-float *Constant*
least-negative-single-float *Constant*
most-negative-single-float *Constant*
Similar to the above, but for full-size floats rather than for short floats.

most-positive-double-float *Constant*
least-positive-double-float *Constant*
least-negative-double-float *Constant*
most-negative-double-float *Constant*
most-positive-long-float *Constant*
least-positive-long-float *Constant*
least-negative-long-float *Constant*
most-negative-long-float *Constant*
These are defined by Common Lisp to be similar to the above, but for double-floats and long-floats. On the Lisp Machine, there are no distinct double and long floating formats; they are synonyms for single-floats. So these constants exist but their values are the same as those of most-positive-single-float and so on.

short-float-epsilon *Constant*
Smallest positive short float which can be added to 1.0s0 and make a difference. That is, for any short float *x* less than this, `(+ 1.0s0 x)` equals 1.0s0.

single-float-epsilon *Constant*
double-float-epsilon *Constant*
long-float-epsilon *Constant*
Smallest positive float which can be added to 1.0 and make a difference. The three names are synonyms on the Lisp Machine, for reasons explained above.

short-float-negative-epsilon *Constant*
Smallest positive short float which can be subtracted from 1.0s0 and make a difference.

single-float-negative-epsilon *Constant*
double-float-negative-epsilon *Constant*
long-float-negative-epsilon *Constant*
Smallest positive float which can be subtracted from 1.0 and make a difference.

7.12 Arithmetic Ignoring Overflow

Sometimes it is desirable to have a form of arithmetic which has no overflow checking (that would produce bignums), and truncates results to the word size of the machine.

%pointer-plus *pointer-1 pointer-2*
Returns a fixnum which is *pointer-1* plus *pointer-2*, modulo what could be stored in the size of the pointer field (currently 25 bits). Arguments other than fixnums are rarely useful, but no type checks are made.

%pointer-difference *pointer-1 pointer-2*
Returns a fixnum which is *pointer-1* minus *pointer-2*. If the arguments are fixnums, rather than true pointers, this provides subtraction modulo what can be stored in the pointer field.

%pointer-times *pointer-1 pointer-2*
Returns a fixnum which is *pointer-1* times *pointer-2*. Arguments other than fixnums are rarely useful, but no type checks are made. The two pointer fields are regarded as signed numbers.

7.13 24-Bit Arithmetic

Sometimes it is useful to have a form of truncating arithmetic with a strictly specified field width which is independent of the range of fixnums permissible on a particular machine. In Zetalisp, this is provided by the following set of functions. Their answers are correct only modulo 2↑24.

These functions should *not* be used for efficiency; they are probably less efficient than the functions which *do* check for overflow. They are intended for algorithms which require this sort of arithmetic, such as hash functions and pseudo-random number generation.

%24-bit-plus *x y*
Returns the sum of *x* and *y* modulo 2↑24. Both arguments must be fixnums.

%24-bit-difference *x y*
Returns the difference of *x* and *y* modulo 2↑24. Both arguments must be fixnums.

%24-bit-times *x y*
Returns the product of *x* and *y* modulo 2↑24. Both arguments must be fixnums.

7.14 Double-Precision Arithmetic

These peculiar functions are useful in programs that don't want to use bignums for one reason or another. They should usually be avoided, as they are difficult to use and understand, and they depend on special numbers of bits and on the use of twos-complement notation.

A double-precision number has 50 bits, of which one is the sign bit. It is represented as two fixnums. The less signficant fixnum conveys 25 signficant bits and is regarded as unsigned (that is, what is normally the sign bit is treated as an ordinary data bit); the more significant fixnum has the same sign as the double-precision number. Only **%float-double** handles negative double-precision numbers; for the other functions, the more signficant fixnum is always positive and contains only 24 bits of actual data.

%multiply-fractions *num1 num2*

Returns bits 25 through 48 (the most significant half) of the product of *num1* and *num2*, regarded as unsigned integers. If you call this and **%pointer-times** on the same arguments *num1* and *num2*, you can combine the results into a double-precision product. If *num1* and *num2* are regarded as two's-complement fractions, $-1 \le num < 1$, **%multiply-fractions** returns 1/2 of their correct product as a fraction.

[The name of this function isn't too great.]

%divide-double *dividend[25:48] dividend[0:24] divisor*

Divides the double-precision number given by the first two arguments by the third argument, and returns the single-precision quotient. Causes an error if *divisor* is zero or if the quotient won't fit in single precision.

There are only 24 bits in each half of the number, as neither sign bit is used to convey information.

%remainder-double *dividend[25:48] dividend[0:24] divisor*

Divides the double-precision number given by the first two arguments by the third argument, and returns the remainder. Causes an error if *divisor* is zero.

%float-double *high25 low25*

high25 and *low25*, which must be fixnums, are concatenated to produce a 50-bit unsigned positive integer. A full-size float containing the same value is constructed and returned. Note that only the 31 most significant bits are retained (after removal of leading zeroes.) This function is mainly for the benefit of **read**.