7. Numbers

Zetalisp includes several types of numbers, with different characteristics. Most numeric functions will 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 24-bit 2's complement binary integers. These are the "preferred, most efficient" type of number.

bignum — Bignums are arbitrary-precision binary integers.

rationalnum — Rationalnums represent rational numbers exactly as the quotient of two integers, each of which can be a fixnum or a bignum. Rationalnums with a denominator of one are not normally created, as an integer will be returned instead.

flonum — Flonums are floating-point numbers. They have a mantissa of 32 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.

small-flonum — Small flonums are another form of floating-point number, with a mantissa of 18 bits and an exponent of 7 bits, providing a precision of about 5 digits and a range of about 10↑19. Stable rounding is employed. Small flonums are useful because, like fixnums, and unlike flonums, they don't require any storage. Computing with small flonums is more efficient than with regular flonums because the operations are faster and consing overhead is eliminated.

complexnum — Complexnums represent complex numbers with a real part and an imaginary part, each of which can be any type of number except a complexnum. Complexnums whose imaginary part is zero are not normally generated, as a real number will be returned instead.

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 undefinedd; 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 at will. If you want to compare the values of two numbers, use = (see page 106) or eql (page 12).

Fixnums and small flonums are exceptions to this rule; some system code knows that **eq** works on fixnums used to represent characters or small integers, and uses **memq** or **assq** on them. **eq** works as well as = as an equality test for fixnums. Small flonums that are = tend to be **eq** also, but it is unwise to depend on this.

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 rationalnum, the result is a rationalnum. When an integer or rationalnum meets a small flonum or a flonum, the result is a small flonum or a flonum (respectively). When a small flonum meets a regular flonum, the result is a regular flonum. When a real number meets a complexnum, the result is a complexnum.

Thus if the constants in a numerical algorithm are written as small flonums (assuming this provides adequate precision), and if the input is a small flonum, the computation will be done in small-flonum mode and the result will be a small flonum, while if the input is a large flonum the computations will be done in full precision and the result will be a flonum.

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

Floating-point numbers retain only a certain number of bits of precision; therefore, the results of computations are only approximate. Large flonums have 31 bits and small flonums have 17 bits, not counting the sign. The method of approximation is "stable rounding". The result of an arithmetic operation will be the flonum which is closest to the exact value. If the exact result falls precisely halfway between two flonums, the result will be 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 small flonums require no associated storage they are as efficient as declared numbers in Maclisp. Bignums and (large) flonums are less efficient; however, bignum and flonum 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. A leading or embedded (but *not* trailing) decimal point, and/or an exponent separated by "e", indicates a flonum. If a number has an exponent separated by "s", it is a small flonum. Small flonums 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 372, 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 small flonum.

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 flonums, this only returns t for exactly 0.0 or 0.0s0; there is no "fuzz".

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*

signp is used to test the sign of a number. It 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	$x \leq 0$
e	$x = 0$
n	$x \neq 0$
ge	$x \geq 0$
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 9).

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 work only on real numbers, since they are meaningless on complex numbers.

= *x y*

Returns t if *x* and *y* are numerically equal. An integer can be = to a flonum.

eql *x y*

Returns t if *x* and *y* are both numbers and numerically equal, or if they are **eq**.

greaterp &rest *two-or-more-args*
> &rest *two-or-more-args*

greaterp compares its arguments from left to right. If any argument is not greater than the next, **greaterp** returns **nil**. But if the arguments are monotonically strictly decreasing, the result is **t**.

Examples:

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

>= &rest *two-or-more-args*
≥ &rest *two-or-more-args*

≥ compares its arguments from left to right. If any argument is less than the next, ≥ returns **nil**. But if the arguments are monotonically decreasing or equal, the result is **t**.

lessp &rest *two-or-more-args*
< &rest *two-or-more-args*

lessp compares its arguments from left to right. If any argument is not less than the next, **lessp** returns **nil**. But if the arguments are monotonically strictly increasing, the result is **t**.

Examples:

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

<= &rest *two-or-more-args*
≤ &rest *two-or-more-args*

≤ 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**.

= *x y*
Returns t if *x* is not numerically equal to *y*, and nil otherwise.

max &rest *one-or-more-args*
max returns the largest of its arguments, which must all be real.
Example:

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

max requires at least one argument.

min &rest *one-or-more-args*
min returns the smallest of its arguments, which must all be real.
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 flonums).

plus &rest *args*
+ &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.

difference *arg* &rest *args*
Returns its first argument minus all of the rest of its arguments.

minus *x*
Returns the negative of *x*.
Examples:

```
(minus 1) => -1
(minus -3.0) => 3.0
```

- *arg* &rest *args*
-$ *arg* &rest *args*
With only one argument, - is the same as minus; it returns the negative of its argument. With more than one argument, - is the same as difference; it returns its first argument minus all of the rest of its arguments.

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

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

abs of a complex number is

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

times &rest *args*
***** &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.

quotient *arg* &rest *args*

Returns the first argument divided by all of the rest of its arguments.

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

The name of this function is written // rather than / because / is the quoting character in Lisp syntax and must be doubled. With more than one argument, // is the same as quotient; 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*).

quotient and // 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 of truncate (see below). This will eventually be changed, and then the value will be a rationalnum if necessary so that the it is mathematically correct. All code that relies on quotient or // to return an integer value rather than a rationalnum should be converted to use truncate (or floor or ceiling, which may simplify the code further). In the mean time, use the function %div 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
```

remainder *x y*
**** *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*).

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

%div *dividend divisor*

Divides, returning a mathematically correct quotient (a rationalnum if necessary) when the arguments are integers. If either argument is a noninteger, the result is the same as that of using //.

```
(%div 1 2) => 1\2
```

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.

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 largest integer less 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.

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.

add1 *x*
1+ *x*
1+$ *x*
(add1 x) is the same as (plus x 1).

sub1 *x*
1- *x*
1-$ *x*
(sub1 x) is the same as (difference x 1). Note that the short name may be confusing: (1- x) does *not* mean 1-x; rather, it means x-1.

gcd *x y* &rest *args*
**** *x y* &rest *args*
Returns the greatest common divisor of all its arguments. The arguments must be integers (fixnums or bignums).

expt *x y*
^ *x y*
^$ *x y*
Returns *x* raised to the *y*'th power. The result is rational (and possibly an integer) if both arguments are integers, and floating-point if either *x* or *y* or both is floating-point. If the exponent is an integer a repeated-squaring algorithm is used; otherwise the result is (exp (* *y* (log *x*))).

Currently complex exponents are not allowed, and complex bases are allowed only with integer exponents.

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*. It is currently implemented only for nonnegative real *x*.

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*.

sys:negative-sqrt (sys:arithmetic-error error) *Condition*

This is signaled when **sqrt** or **isqrt**'s argument is negative. The **:number** operation on the condition instance will return the argument.

***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.

7.4 Complex Number Functions

See also the predicates **realp** and **complexp** (page 10).

complex *x* &optional *y*

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

If *y* is zero, a peculiar complexnum is created whose numeric value is actually real. If *y* is omitted, a number is used whose value is zero and whose type is the same as that of *x*. Note that **realp** of this peculiar complexnum will be **nil** even though the mathematical value is indeed real.

realpart *x*

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

imagpart *x*

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

conjugate *x*

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

phase *x*

Returns the phase angle of the complex number *x* in its polar form. This is the angle from the positive x-axis to the ray from the origin through *x*. The value is always in the interval **[0, 2π)**.

cis *angle*

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

signum *x*

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

If *x* is real, the value is either **1** or **-1**.

7.5 Transcendental Functions

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

Currently complex arguments are not allowed. This will be changed some day.

exp *x*

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

log *x*

Returns the natural logarithm of *x*.

sys:non-positive-log (sys:arithmetic-error error) *Condition*

This is signaled when the argument to **log** is a negative number or zero. The **:number** operation on the condition instance returns that number.

sin *x*

Returns the sine of *x*, where *x* is expressed in radians.

sind *x*

Returns the sine of *x*, where *x* is expressed in degrees.

cos *x*

Returns the cosine of *x*, where *x* is expressed in radians.

cosd *x*

Returns the cosine of *x*, where *x* is expressed in degrees.

atan *y x*

Returns the angle, in radians, whose tangent is *y/x*. **atan** always returns a non-negative number between zero and 2π.

atan2 *y x*

Returns the angle, in radians, whose tangent is *y/x*. **atan2** always returns a number between $-\pi$ and π.

7.6 Numeric Type Conversions

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

fix *x*

Converts *x* from a flonum (or small-flonum) or rationalnum 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.

float *x*

Converts any kind of number to a flonum.

small-float *x*

Converts any kind of number to a small flonum.

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 rationalnum, an error is signaled.

demoninator *x*

Returns the demoninator of the rational number *x*. If *x* is an integer, the value is **1**. If *x* is not an integer or rationalnum, an error is signaled.

rational *x*

Converts *x* to a rational number. If *x* is an integer or a rationalnum, 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 rationalnum, 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.

7.7 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 24-bit fixnums. Negative numbers are operated on in their 2's-complement representation.

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

Returns the bit-wise logical *inclusive or* of its arguments. At least one argument is required.

Example:

```
(logior 4002 67) => 4067
```

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

Returns the bit-wise logical *exclusive or* of its arguments. At least one argument is required.

Example:

```
(logxor 2531 7777) => 5246
```

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

Returns the bit-wise logical *and* of its arguments. At least one argument is required.

Examples:

```
(logand 3456 707) => 406
(logand 3456 -100) => 3400
```

lognot *number*

Returns the logical complement of *number*. This is the same as logxor'ing *number* with -1.

Example:

```
(lognot 3456) => -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 logxor are usually preferred over the equivalent forms of boole, to avoid putting magic numbers in the program.

bit-test *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)))
```

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) => 10      ;(octal)
(lsh 14 -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 flonum, 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 24-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) => 20000000
(rot -1 7) => -1
(rot 15 24.) => 15
```

haulong *x*

This returns the number of significant bits in |*x*|. *x* may be a fixnum or a bignum. Its sign is ignored. The result is the least integer strictly greater than the base-2 logarithm of |*x*|.

Examples:

```
(haulong 0) => 0
(haulong 3) => 2
(haulong -7) => 3
```

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))))
```

7.8 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. Byte specifiers are 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, counting from the right in bits. A position of zero means that the byte is at the right end of the number. For example, the byte-specifier 0010 (i.e. 10 octal) refers to the lowest eight bits of a word, and the byte-specifier 1010 refers to the next eight bits. These byte-specifiers will be stylized below as *ppss*. The maximum value of the *ss* digits is 27 (octal), since a byte must fit in a fixnum although bytes can be loaded from and deposited into bignums. (Bytes are always positive numbers.) The format of byte-specifiers is taken from the pdp-10 byte instructions.

ldb *ppss num*

ppss specifies a byte of *num* to be extracted. The *ss* bits of the byte starting at bit *pp* are the lowest *ss* bits in the returned value, and the rest of the bits in the returned value are zero. The name of the function, **ldb**, means "load byte". *num* may be a fixnum or a bignum. The returned value is always a fixnum.

Example:

```
(ldb 0306 4567) => 56
```

load-byte *num 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 *ppss y*

ldb-test is a predicate which returns t if any of the bits designated by the byte specifier *ppss* are 1's in *y*. 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 ppss y) ==> (not (zerop (ldb ppss y)))
```

mask-field *ppss num*

This is similar to ldb; however, the specified byte of *num* is returned as a number in position *pp* of the returned word, instead of position 0 as with ldb. *num* must be a fixnum.

Example:

```
(mask-field 0306 4567) => 560
```

dpb *byte ppss num*

Returns a number which is the same as *num* except in the bits specified by *ppss*. The low *ss* bits of *byte* are placed in those bits. *byte* is interpreted as being right-justified, as if it were the result of ldb. *num* may be a fixnum or a bignum. The name means "deposit byte".

Example:

```
(dpb 23 0306 4567) => 4237
```

deposit-byte *num 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 ppss num*

This is like dpb, except that *byte* is not taken to be left-justified; the *ppss* bits of *byte* are used for the *ppss* bits of the result, with the rest of the bits taken from *num*. *num* must be a fixnum.

Example:

```
(deposit-field 230 0306 4567) => 4237
```

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 *ppss fixnum*

%logldb is like ldb except that it only loads out of fixnums and allows a byte size of 30 (octal), i.e. all 24. bits of the fixnum including the sign bit.

%logdpb *byte ppss fixnum*

%logdpb is like dpb except that it only deposits into fixnums. Using this to change the sign-bit will leave 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.

7.9 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. Between calls, its state is saved in a data object called a *random-array*. Usually there is only one random-array; however, if you want to create a reproducible series of pseudo-random numbers, and be able to reset the state to control when the series starts over, then you need some of the other functions here.

random &optional *arg random-array*

(random) returns a random fixnum, positive or negative. If *arg* is present, a fixnum between 0 and *arg* minus 1 inclusive is returned. If *random-array* is present, the given array is used instead of the default one (see below). Otherwise, the default random-array is used (and is created if it doesn't already exist). The algorithm is executed inside a without-interrupts (see page 540) so two processes can use the same random-array without colliding.

si:random-in-range *low high*

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

A random-array 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 $x \uparrow length + x \uparrow offset + 1$ 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-array is created, and it can be changed. To have several different controllable resettable sources of random numbers, you can create your own random-arrays. If you don't care about reproducibility of sequences, just use random without the *random-array* argument.

si:random-create-array *length offset seed* &optional (*area* **nil**)
Creates, initializes, and returns a random-array. *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 array before returning it.

si:random-initialize *array* &optional *new-seed*
array must be a random-array, 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).

7.10 24-Bit Numbers

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. 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.

7.11 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 two's-complement notation.

%multiply-fractions *num1 num2*
Returns bits 24 through 46 (the most significant half) of the product of *num1* and *num2*. If you call this and %24-bit-times on the same arguments *num1* and *num2*, regarding them as integers, you can combine the results into a double-precision product. If *num1* and *num2* are regarded as two's-complement fractions, $-1 \leq num < 1$, %multiply-fractions returns 1/2 of their correct product as a fraction. (The name of this function isn't too great.)

%divide-double *dividend[24:46] dividend[0:23] 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.

%remainder-double *dividend[24:46] dividend[0:23] 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 *high24 low24*

high24 and *low24*, which must be fixnums, are concatenated to produce a 48-bit unsigned positive integer. A flonum 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**.