7. NUMBERS AND ARITHMETIC FUNCTIONS

There are four different types of numbers in Interlisp: small integers, large integers, bignums (arbitrary-size integers), and floating-point numbers. Small integers are in the range -65536 to 65535. Large integers and floating-point numbers are 32-bit quantities that are stored by "boxing" the number (see below). Bignums are "boxed" as a series of words.

Large integers and floating-point numbers can be any full word quantity. To distinguish among the various kinds of numbers, and other Interlisp pointers, these numbers are "boxed" When a large integer or floating-point number is created (by an arithmetic operation or by READ), Interlisp gets a new word from "number storage" and puts the number into that word. Interlisp then passes around the pointer to that word, i.e., the "boxed number", rather than the actual quantity itself. When a numeric function needs the actual numeric quantity, it performs the extra level of addressing to obtain the "value" of the number. This latter process is called "unboxing". Unboxing does not use any storage, but each boxing operation uses one new word of number storage. If a computation creates many large integers or floating-point numbers, i.e., does lots of boxes, it may cause a garbage collection of large integer space, or of floating-point number space.

The following functions can be used to distinguish the different types of numbers:

(**SMALLP** *X*) [Function]

Returns *X*, if *X* is a small integer; NIL otherwise. Does not generate an error if *X* is not a number.

(**FIXP** *X*) [Function]

Returns *X*, if *X* is an integer; NIL otherwise. Note that FIXP is true for small integers, large integers, and bignums. Does not generate an error if *X* is not a number.

(**FLOATP** *X*) [Function]

Returns *X* if *X* is a floating-point number; NIL otherwise. Does not give an error if *X* is not a number.

(**NUMBERP** *X*) [Function]

Returns *X*, if *X* is a number of any type; NIL otherwise. Does not generate an error if *X* is not a number.

Note: In previous releases, NUMBERP was true only if (FLOATP *X*) or (FIXP *X*) were true. With the additon of Common Lisp ratios and complex numbers, NUMBERP now returns T for *all* number types . Code relying on the "old" behavior should be modified.

Each small integer has a unique representation, so EQ may be used to check equality. EQ should not be used for large integers, bignums, or floating-point numbers, EQP, IEQP, or EQUAL must be used instead.

(**EQP** *X Y*) [Function]

Returns T, if *X* and *Y* are equal numbers; NIL otherwise. EQ may be used if *X* and *Y* are known to be small integers. EQP does not convert *X* and *Y* to integers, e.g., (EQP 2000 2000.3) => NIL, but it can be used to compare an integer and a floating-point number, e.g., (EQP 2000 2000.0) => T. EQP does not generate an error if *X* or *Y* are not numbers.

EQP can also be used to compare stack pointers (see Chapter 11) and compiled code objects (see Chapter 10).

The action taken on division by zero and floating-point overflow is determined with the following function:

(**OVERFLOW** *FLG*) [Function]

Sets a flag that determines the system response to arithmetic overflow (for floating-point arithmetic) and division by zero; returns the previous setting.

For integer arithmetic: If *FLG* = T, an error occurs on division by zero. If *FLG* = NIL or 0, integer division by zero returns zero. Integer overflow cannot occur, because small integers are converted to bignums (see the beginning of this chapter).

For floating-point arithmetic: If *FLG* = T, an error occurs on floating overflow or floating division by zero. If *FLG* = NIL or 0, the largest (or smallest) floating-point number is returned as the result of the overflowed computation or floating division by zero.

The default value for OVERFLOW is T, meaning an error is generated on division by zero or floating overflow.

Generic Arithmetic

The functions in this section are "generic" arithmetic functions. If any of the arguments are floating-point numbers (see the Floating-Point Arithmetic section below), they act exactly like floating-point functions, floating all arguments and returning a floating-point number as their value. Otherwise, they act like the integer functions (see the Integer Arithmetic section below). If given a non-numeric argument, they generate an error, Non-numeric arg. The results of division by zero and floating-point overflow is determined by the function OVERFLOW (see the section above).

(**PLUS** *X X ... X*) [NoSpread Function]

X + *X* + ... + *X*.

(**MINUS** *X*) [Function]

- *X*

(**DIFFERENCE** *X Y*) [Function]

X - *Y*

(**TIMES** *X X ... X*) [NoSpread Function]

X * *X* * ... * *X*

(**QUOTIENT** *X Y*) [Function]

If *X* and *Y* are both integers, returns the integer division of *X* and *Y*. Otherwise, converts both *X* and *Y* to floating-point numbers, and does a floating-point division.

(**REMAINDER** *X Y*) [Function]

If *X* and *Y* are both integers, returns (IREMAINDER *X Y*), otherwise (FREMAINDER *X Y*).

(**GREATERP** *X Y*) [Function]

T, if *X* > *Y*, NIL otherwise.

(**LESSP** *X Y*) [Function]

T if *X* < *Y*, NIL otherwise.

(**GEQ** *X Y*) [Function]

T, if *X* >= *Y*, NIL otherwise.

(**LEQ** *X Y*) [Function]

T, if *X* <= *Y*, NIL otherwise.

(**ZEROP** *X*) [Function]

The same as (EQP *X* 0).

(**MINUSP** *X*) [Function]

T, if *X* is negative; NIL otherwise. Works for both integers and floating-point numbers.

(**MIN** *X X ... X*) [NoSpread Function]

Returns the minimum of *X* , *X* , . . . , *X* . (MIN) returns the value of MAX.INTEGER (see the Integer Arithmetic section below).

(**MAX** *X X ... X*) [NoSpread Function]

Returns the maximum of *X* , *X* , ..., *X* . (MAX) returns the value of MIN.INTEGER (see the Integer Arithmetic section below).

(**ABS** *X*) [Function]

X if *X* > 0, otherwise -*X*. ABS uses GREATERP and MINUS (not IGREATERP and IMINUS).

Integer Arithmetic

The input syntax for an integer is an optional sign (+ or -) followed by a sequence of decimal digits, and terminated by a delimiting character. Integers entered with this syntax are interpreted as decimal integers. Integers in other radices can be entered as follows:

123Q
#o123 If an integer is followed by the letter Q, or preceeded by a pound sign and the letter "o", the digits are interpreted as an octal (base 8) integer.

`#b10101` If an integer is preceeded by a pound sign and the letter "`b`", the digits are interpreted as a binary (base 2) integer.

`#x1A90` If an integer is preceeded by a pound sign and the letter "`x`", the digits are interpreted as a hexadecimal (base 16) integer.

`#5r1243` If an integer is preceeded by a pound sign, a positive decimal integer `BASE`, and the letter "`r`", the digits are interpreted as an integer in the base `BASE`. For example, `#8r123 = 123Q`, and `#16r12A3 = #x12A3`. When typing a number in a radix above ten, the uppercase letters A through Z can be used as the digits after 9 (but there is no digit above Z, so it is not possible to type all base-99 digits).

Medley keeps no record of how you typed a number, so `77Q` and `63` both correspond to the same integer, and are indistinguishable internally. The function `RADIX` (see Chapter 25), sets the radix used to print integers.

`PACK` and `MKATOM` create numbers when given a sequence of characters observing the above syntax, e.g. `(PACK '(1 2 Q)) => 10`. Integers are also created as a result of arithmetic operations.

The range of integers of various types is implementation-dependent. This information is accessible to you through the following variables:

MIN.SMALLP [Variable]
MAX.SMALLP [Variable]

The smallest/largest possible small integer.

MIN.FIXP [Variable]
MAX.FIXP [Variable]

The smallest/largest possible large integer.

MIN.INTEGER [Variable]
MAX.INTEGER [Variable]

The value of `MAX.INTEGER` and `MIN.INTEGER` are two special system datatypes. For some algorithms, it is useful to have an integer that is larger than any other integer. Therefore, the values of `MAX.INTEGER` and `MIN.INTEGER` are two special data types; the value of `MAX.INTEGER` is `GREATERP` than any other integer, and the value of `MIN.INTEGER` is `LESSP` than any other integer. Trying to do arithmetic using these special bignums, other than comparison, will cause an error.

All of the functions described below work on integers. Unless specified otherwise, if given a floating-point number, they first convert the number to an integer by truncating the fractional bits, e.g., `(IPLUS 2.3 3.8) = 5`; if given a non-numeric argument, they generate an error, `Non-numeric arg`.

(**IPLUS** *X X ... X*) [NoSpread Function]

Returns the sum *X* + *X* + ... + *X* `(IPLUS) = 0`.

(**IMINUS** *X*) [Function]

-*X*

(**IDIFFERENCE** *X Y*) [Function]

`X - Y`

(**ADD1** *X*) [Function]

`X + 1`

(**SUB1** *X*) [Function]

`X - 1`

(**ITIMES** *X X ... X*) [NoSpread Function]

Returns the product `X * X * ... * X`. `(ITIMES) = 1`.

(**IQUOTIENT** *X Y*) [Function]

X / *Y* truncated. Examples:

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

If *Y* is zero, the result is determined by the function `OVERFLOW`.

(**IREMAINDER** *X Y*) [Function]

Returns the remainder when *X* is divided by *Y*. Example:

```
(IREMAINDER 5 2) => 1
```

(**IMOD** *X N*) [Function]

Computes the integer modulus of *X* mod *N*; this differs from `IREMAINDER` in that the result is always a non-negative integer in the range [0,*N*).

(**IGREATERP** *X Y*) [Function]

`T`, if *X* > *Y*; `NIL` otherwise.

(**ILESSP** *X Y*) [Function]

`T`, if *X* < *Y*; `NIL` otherwise.

(**IGEQ** *X Y*) [Function]

`T`, if *X* >= *Y*; `NIL` otherwise.

(**ILEQ** *X Y*) [Function]

`T`, if *X* <= *Y*; `NIL` otherwise.

(**IMIN** *X X ... X*) [NoSpread Function]

Returns the minimum of *X*, *X*, ..., *X*. `(IMIN)` returns the largest possible large integer, the value of `MAX.INTEGER`.

(**IMAX** *X X ... X*) [NoSpread Function]

Returns the maximum of *X* , *X* , ..., *X* . (IMAX) returns the smallest possible large integer, the value of MIN.INTEGER.

(**IEQP** *X Y*) [Function]

Returns T if *X* and *Y* are equal integers; NIL otherwise. Note that EQ may be used if *X* and *Y* are known to be small integers. IEQP converts *X* and *Y* to integers, e.g., (IEQP 2000 2000.3) => T.

(**FIX** *N*) [Function]

If *N* is an integer, returns *N*. Otherwise, converts *N* to an integer by truncating fractional bits For example, (FIX 2.3) => 2, (FIX -1.7) => -1.

Since FIX is also a programmer's assistant command (see Chapter 13), typing FIX directly to a Medley executive will not cause the function FIX to be called.

(**FIXR** *N*) [Function]

If *N* is an integer, returns *N*. Otherwise, converts *N* to an integer by rounding. FIXR will round towards the even number if *N* is exactly half way between two integers. For example, (FIXR 2.3) => 2, (FIXR -1.7) => -2, (FIXR 3.5) => 4).

(**GCD** *N N*) [Function]

Returns the greatest common divisor of *N* and *N* , (GCD 72 64)=8.

Logical Arithmetic Functions

(**LOGAND** *X X ... X*) [NoSpread Function]

Returns the logical AND of all its arguments, as an integer. Example:

```
(LOGAND 7 5 6)  =>  4
```

(**LOGOR** *X X ... X*) [NoSpread Function]

Returns the logical OR of all its arguments, as an integer. Example:

```
(LOGOR 1 3 9)  =>  11
```

(**LOGXOR** *X X ... X*) [NoSpread Function]

Returns the logical exclusive OR of its arguments, as an integer. Example:

```
(LOGXOR 11 5)  =>  14
(LOGXOR 11 5 9) = (LOGXOR 14 9)  =>  7
```

(**LSH** *X N*) [Function]

(Arithmetic) "Left Shift." Returns *X* shifted left *N* places, with the sign bit unaffected. *X* can be positive or negative. If *N* is negative, *X* is shifted right -*N* places.

(**RSH** *X N*) [Function]

(Arithmetic) "Right Shift." Returns *X* shifted right *N* places, with the sign bit unaffected, and copies of the sign bit shifted into the leftmost bit. *X* can be positive or negative. If *N* is negative, *X* is shifted left -*N* places.

Warning: Be careful if using RSH to simulate division; RSHing a negative number isn't the same as dividing by a power of two.

(**LLSH** *X N*) [Function]
(**LRSH** *X N*) [Function]

"Logical Left Shift" and "Logical Right Shift". The difference between a logical and arithmetic right shift lies in the treatment of the sign bit. Logical shifting treats it just like any other bit; arithmetic shifting will not change it, and will "propagate" rightward when actually shifting rightwards. Note that shifting (arithmetic) a negative number "all the way" to the right yields -1, not 0.

Note: LLSH and LRSH always operate mod-2^{32} arithmetic. Passing a bignum to either of these will cause an error. LRSH of negative numbers will shift 0s into the high bits.

(**INTEGERLENGTH** *X*) [Function]

Returns the number of bits needed to represent *X*. This is equivalent to: 1+floor[log2[abs[*X*]]]. (INTEGERLENGTH 0) = 0.

(**POWEROFTWOP** *X*) [Function]

Returns non-NIL if *X* (coerced to an integer) is a power of two.

(**EVENP** *X Y*) [NoSpread Function]

If *Y* is not given, equivalent to (ZEROP (IMOD *X* 2)); otherwise equivalent to (ZEROP (IMOD X Y)).

(**ODDP** *N MODULUS*) [NoSpread Function]

Equivalent to (NOT (EVENP N MODULUS)). *MODULUS* defaults to 2.

(**LOGNOT** *N*) [Macro]

Logical negation of the bits in *N*. Equivalent to (LOGXOR *N* -1).

(**BITTEST** *N MASK*) [Macro]

Returns T if any of the bits in *MASK* are on in the number *N*. Equivalent to (NOT (ZEROP (LOGAND *N MASK*))).

(**BITCLEAR** *N MASK*) [Macro]

Turns off bits from *MASK* in *N*. Equivalent to (LOGAND *N* (LOGNOT *MASK*)).

(**BITSET** *N MASK*) [Macro]

Turns on the bits from *MASK* in *N*. Equivalent to (LOGOR *N MASK*).

(**MASK.1'S** *POSITION SIZE*) [Macro]

Returns a bit-mask with *SIZE* one-bits starting with the bit at *POSITION*. Equivalent to (LLSH (SUB1 (EXPT 2 *SIZE*)) *POSITION*).

(**MASK.0'S** *POSITION SIZE*) [Macro]

Returns a bit-mask with all one bits, except for *SIZE* bits starting at *POSITION*. Equivalent to (LOGNOT (MASK.1'S *POSITION SIZE*)).

(**LOADBYTE** *N POS SIZE*) [Function]

Extracts *SIZE* bits from *N*, starting at position *POS*. Equivalent to (LOGAND (RSH *N POS*) (MASK.1'S 0 *SIZE*)).

(**DEPOSITBYTE** *N POS SIZE VAL*) [Function]

Insert *SIZE* bits of *VAL* at position *POS* into *N*, returning the result. Equivalent to

```
(LOGOR (BITCLEAR N (MASK.1'S POS SIZE))
  (LSH (LOGAND VAL (MASK.1'S 0 SIZE))
    POS))
```

(**ROT** *X N FIELDSIZE*) [Function]

"Rotate bits in field". It performs a bitwise left-rotation of the integer *X*, by *N* places, within a field of *FIELDSIZE* bits wide. Bits being shifted out of the position selected by (EXPT 2 (SUB1 *FIELDSIZE*)) will flow into the "units" position.

The notions of position and size can be combined to make up a "byte specifier", which is constructed by the macro BYTE [note reversal of arguments as compared with the above functions]:

(**BYTE** *SIZE POSITION*) [Macro]

Constructs and returns a "byte specifier" containing *SIZE* and *POSITION*.

(**BYTESIZE** *BYTESPEC*) [Macro]

Returns the SIZE componant of the "byte specifier" *BYTESPEC*.

(**BYTEPOSITION** *BYTESPEC*) [Macro]

Returns the POSITION componant of the "byte specifier" *BYTESPEC*.

(**LDB** *BYTESPEC VAL*) [Macro]

Equivalent to

(LOADBYTE *VAL* (BYTEPOSITION *BYTESPEC*) (BYTESIZE *BYTESPEC*))

(**DPB** *N BYTESPEC VAL*) [Macro]

Equivalent to

(DEPOSITBYTE *VAL* (BYTEPOSITION *BYTESPEC*) (BYTESIZE *BYTESPEC*) *N*)

Floating-Point Arithmetic

A floating-point number is input as a signed integer, followed by a decimal point, and another sequence of digits called the fraction, followed by an exponent (represented by E followed by a signed integer) and terminated by a delimiter.

Both signs are optional, and either the fraction following the decimal point, or the integer preceding the decimal point may be omitted. One or the other of the decimal point or exponent may also be omitted, but at least one of them must be present to distinguish a floating-point number from an integer. For example, the following will be recognized as floating-point numbers:

```
5.      5.00    5.01    .3
5E2     5.1E2   5E-3    -5.2E+6
```

Floating-point numbers are printed using the format control specified by the function FLTFMT (see Chapter 25). FLTFMT is initialized to T, or free format. For example, the above floating-point numbers would be printed free format as:

```
5.0     5.0     5.01    .3
500.0   510.0   .005    -5.2E6
```

Floating-point numbers are created by the reader when a "." or an E appears in a number, e.g., 1000 is an integer, 1000. a floating-point number, as are 1E3 and 1.E3. Note that 1000D, 1000F, and 1E3D are perfectly legal literal atoms. Floating-point numbers are also created by PACK and MKATOM, and as a result of arithmetic operations.

PRINTNUM (see Chapter 25) permits greater control over the printed appearance of floating-point numbers, allowing such things as left-justification, suppression of trailing decimals, etc.

The floating-point number range is stored in the following variables:

MIN.FLOAT [Variable]

The smallest possible floating-point number.

MAX.FLOAT [Variable]

The largest possible floating-point number.

All of the functions described below work on floating-point numbers. Unless specified otherwise, if given an integer, they first convert the number to a floating-point number, e.g., (FPLUS 1 2.3) <=> (FPLUS 1.0 2.3) => 3.3; if given a non-numeric argument, they generate an error, Non-numeric arg.

(**FPLUS** *X X ... X*) [NoSpread Function]

X + *X* + ... + *X*

(**FMINUS** *X*) [Function]

- *X*

(**FDIFFERENCE** *X Y*) [Function]

X - *Y*

(**FTIMES** *X X ... X*) [NoSpread Function]

X * *X* * ... * *X*

(**FQUOTIENT** *X Y*) [Function]

X / *Y*.

The results of division by zero and floating-point overflow is determined by the function OVERFLOW.

(**FREMAINDER** *X Y*) [Function]

Returns the remainder when *X* is divided by *Y*. Equivalent to:

```
(FDIFFERENCE X (FTIMES Y (FIX (FQUOTIENT X Y))))
```

Example:

```
(FREMAINDER 7.5 2.3) => 0.6
```

(**FGREATERP** *X Y*) [Function]

T, if *X* > *Y*, NIL otherwise.

(**FLESSP** *X Y*) [Function]

T, if *X* < *Y*, NIL otherwise.

(**FEQP** *X Y*) [Function]

Returns T if *X* and *Y* are equal floating-point numbers; NIL otherwise. FEQP converts *X* and *Y* to floating-point numbers.

(**FMIN** *X X ... X*) [NoSpread Function]

Returns the minimum of *X* , *X* , . . . , *X* . (FMIN) returns the largest possible floating-point number, the value of MAX.FLOAT.

(**FMAX** *X X ... X*) [NoSpread Function]

Returns the maximum of *X* , *X* , . . . , *X* . (FMAX) returns the smallest possible floating-point number, the value of MIN.FLOAT.

(**FLOAT** *X*) [Function]

Converts *X* to a floating-point number. Example:

```
(FLOAT 0) => 0.0
```

Transcendental Arithmetic Functions

(**EXPT** *A N*) [Function]

Returns A^N. If *A* is an integer and *N* is a positive integer, returns an integer, e.g, (EXPT 3 4) => 81, otherwise returns a floating-point number. If *A* is negative and *N* fractional, generates the error, Illegal exponentiation. If *N* is floating and either too large or too small, generates the error, Value out of range expt.

(**SQRT** *N*) [Function]

Returns the square root of *N* as a floating-point number. *N* may be fixed or floating-point. Generates an error if *N* is negative.

(**LOG** *X*) [Function]

Returns the natural logarithm of *X* as a floating-point number. *X* can be integer or floating-point.

(**ANTILOG** *X*) [Function]

Returns the floating-point number whose logarithm is *X*. *X* can be integer or floating-point. Example:

```
(ANTILOG 1) = e => 2.71828...
```

(**SIN** *X RADIANSFLG*) [Function]

Returns the sine of *X* as a floating-point number. *X* is in degrees unless *RADIANSFLG* = T.

(**COS** *X RADIANSFLG*) [Function]

Similar to SIN.

(**TAN** *X RADIANSFLG*) [Function]

Similar to SIN.

(**ARCSIN** *X RADIANSFLG*) [Function]

The value of ARCSIN is a floating-point number, and is in degrees unless *RADIANSFLG* = T. In other words, if (ARCSIN *X RADIANSFLG*) = Z then (SIN Z *RADIANSFLG*) = *X*. The range of the value of ARCSIN is -90 to +90 for degrees, $-\pi/2$ to $\pi/2$ for radians. *X* must be a number between -1 and 1.

(**ARCCOS** *X RADIANSFLG*) [Function]

Similar to ARCSIN. Range is 0 to 180, 0 to π.

(**ARCTAN** *X RADIANSFLG*) [Function]

Similar to ARCSIN. Range is 0 to 180, 0 to π.

(**ARCTAN2** *Y X RADIANSFLG*) [Function]

Computes (ARCTAN (FQUOTIENT *Y X*) *RADIANSFLG*), and returns a corresponding value in the range -180 to 180 (or $-\pi$ to π), i.e. the result is in the proper quadrant as determined by the signs of *X* and *Y*.

Generating Random Numbers

(**RAND** *LOWER UPPER*) [Function]

Returns a pseudo-random number between *LOWER* and *UPPER* inclusive, i.e., RAND can be used to generate a sequence of random numbers. If both limits are integers, the value of RAND is an integer, otherwise it is a floating-point number. The algorithm is completely deterministic, i.e., given the same initial state, RAND produces the same sequence of values. The internal state of RAND is initialized using the function RANDSET.

(**RANDSET** *X*) [Function]

Returns the internal state of RAND. If *X* = NIL, just returns the current state. If *X* = T, RAND is initialized using the clocks, and RANDSET returns the new state. Otherwise, *X* is interpreted as a previous internal state, i.e., a value of RANDSET, and is used to reset RAND. For example,

```
←(SETQ OLDSTATE (RANDSET))
 ...
←(for X from 1 to 10 do (PRIN1 (RAND 1 10)))
 2847592748NIL
←(RANDSET OLDSTATE)
 ...
←(for X from 1 to 10 do (PRIN1 (RAND 1 10)))
 2847592748NIL
```

[This page intentionally left blank]