7. NUMBERS AND ARITHMETIC FUNCTIONS

Numerical atoms, or simply numbers, do not have value cells, function definition cells, property lists, or explicit print names. There are four different types of numbers in Interlisp: small integers, large integers, bignums (arbitrary-size integers), and floating point numbers. Small integers are those integers that can be directly stored within a pointer value (implementation-dependent). Large integers and floating point numbers are full-word 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. In order to distinguish between those full word quantities that represent large integers or floating point numbers, and other Interlisp pointers, these numbers are "boxed": When a large integer or floating point number is created (via an arithmetic operation or by **READ**), Interlisp gets a new word from "number storage" and puts the large integer or floating point number into that word. Interlisp then passes around the pointer to that word, i.e., the "boxed number", rather than the actual quantity itself. Then 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". Note that unboxing does not use any storage, but that each boxing operation uses one new word of number storage. Thus, 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.

Note: Different implementations of Interlisp may use different boxing strategies. Thus, while lots of arithmetic operations *may* lead to garbage collections, this is not necessarily always the case.

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 (**FIXP** or **FLOATP**); **NIL** otherwise. Does *not* generate an error if *X* is not a number.

Note that if **(NUMBERP *X*)** is true, then either **(FIXP *X*)** or **(FLOATP *X*)** is true.

Each small integer has a unique representation, so **EQ** may be used to check equality. Note that **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 **EQ**, or equal numbers; **NIL** otherwise. Note that **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.

Note: **EQP** can also be used to compare stack pointers (page 11.4) and compiled code objects (page 10.10).

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 (page 7.1).

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 to cause an error on division by zero or floating overflow.

7.1 Generic Arithmetic

The functions in this section are "generic" arithmetic functions. If any of the arguments are floating point numbers (page 7.11), they act exactly like floating point functions, and float all arguments, and return a floating point number as their value. Otherwise, they act like the integer functions (page 7.4). If given a non-numeric argument, they generate an error, **NON-NUMERIC ARG**.

(PLUS X_1 X_2 ... X_N**)** [NoSpread Function]

$X_1 + X_2 + ... + X_N$.

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

$-X$

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

$X - Y$

(TIMES X_1 X_2 ... X_N**)** [NoSpread Function]

$X_1 * X_2 * ... * X_N$

(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.

The results of division by zero and floating point overflow is determined by the function **OVERFLOW** (page 7.2).

(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]

(EQP *X* **0)**.

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

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

(MIN X_1 X_2 ... X_N**)** [NoSpread Function]

Returns the minimum of X_1, X_2, ..., X_N. **(MIN)** returns the value of **MAX.INTEGER** (page 7.5).

(MAX X_1 X_2 ... X_N**)** [NoSpread Function]

Returns the maximum of X_1, X_2, ..., X_N. **(MAX)** returns the value of **MIN.INTEGER** (page 7.5).

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

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

7.2 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 proceeded by a vertical bar and the letter "o", the digits are interpreted an octal (base 8) integer.

|b10101 If an integer is proceeded by a vertical bar and the letter "b", the digits are interpreted as a binary (base 2) integer.

|x1A90 If an integer is proceeded by a vertical bar and the letter "x", the digits are interpreted as a hexadecimal (base 16) integer. The upper-case letters **A** though **F** are used as the digits after **9**.

|5r1243 If an integer is proceeded by a vertical bar, 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 inputting a number in a radix above ten, the upper-case 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).

Note that **77Q** and **63** both correspond to the same integers, and in fact are indistinguishable internally since no record is kept of the syntax used to create an integer. The function **RADIX** (page 25.13), sets the radix used to print integers.

Integers are created by **PACK** and **MKATOM** 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 accessable to the user 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 smallest/largest possible integers. 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 bignums; 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_1 X_2 ... X_N) [NoSpread Function]

Returns the sum $X_1 + X_2 + ... + X_N$. **(IPLUS) = 0**.

(IMINUS *X*) [Function]

-X

(IDIFFERENCE *X Y*) [Function]

X - Y

(ADD1 *X*) [Function]

X + 1

(SUB1 *X*) [Function]

X - 1

(ITIMES X_1 X_2 ... X_N) [NoSpread Function]

Returns the product $X_1 * X_2 * ... * X_N$. **(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** (page 7.2).

(IREMAINDER *X Y*) [Function]

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

(IREMAINDER 3 2) => 1

(IMOD *X N*) [Function]

Computes the integer modulus; 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_1 X_2 ... X_N**)** [NoSpread Function]

Returns the minimum of $X_1, X_2, ..., X_N$. **(IMIN)** returns the largest possible large integer, the value of **MAX.INTEGER**.

(IMAX X_1 X_2 ... X_N**)** [NoSpread Function]

Returns the maximum of $X_1, X_2, ..., X_N$. **(IMAX)** returns the smallest possible large integer, the value of **MIN.INTEGER**.

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

Returns **T** if *X* and *Y* are **EQ** or 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**. Causes **NON-NUMERIC ARG** error if either *X* or *Y* are not numbers.

(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**.

Note: Since **FIX** is also a programmer's assistant command (page 13.12), typing **FIX** directly to Interlisp 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. For example, **(FIXR 2.3) => 2**, **(FIXR -1.7) => -2**, **(FIXR 3.5) => 4)**.

(GCD *N1 N2***)** [Function]

Returns the greatest common divisor of *N1* and *N2*, e.g., **(GCD 72 64) = 8**.

7.3 Logical Arithmetic Functions

(LOGAND X_1 X_2 ... X_N) [NoSpread Function]

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

(LOGAND 7 5 6) => 4

(LOGOR X_1 X_2 ... X_N) [NoSpread Function]

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

(LOGOR 1 3 9) => 11

(LOGXOR X_1 X_2 ... X_N) [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; **RSH**ing a negative number is not generally equivalent to 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** are currently implemented using mod-2↑32 arithmetic. Passing a bignum to either of these will cause an error. **LRSH** of negative numbers will shift in 0s in the high bits.

(INTEGERLENGTH *X*) [Function]

Returns the number of bits needed to represent *X* (coerced to an integer). 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 compare with 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***)**

7.4 Floating Point Arithmetic

A floating point number is input as a signed integer, followed by a decimal point, followed by 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** (page 25.13). **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 read program 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 (page 25.15) permits greater controls on 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_1 X_2 ... X_N) [NoSpread Function]

$X_1 + X_2 + \ldots + X_N$

(FMINUS X) [Function]

$-X$

(FDIFFERENCE X Y) [Function]

$X - Y$

(FTIMES X_1 X_2 ... X_N) [NoSpread Function]

$X_1 * X_2 * \ldots * X_N$

(FQUOTIENT X Y) [Function]

X / Y.

The results of division by zero and floating point overflow is determined by the function **OVERFLOW** (page 7.2).

(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 N and M are equal floating point numbers; **NIL** otherwise. **FEQP** converts N and M to floating point numbers.Causes **NON-NUMERIC ARG** error if either N or M are not numbers.

(FMIN X_1 X_2 ... X_N) [NoSpread Function]

Returns the minimum of $X_1, X_2, \ldots, X_N$. **(FMIN)** returns the largest possible floating point number, the value of **MAX.FLOAT**.

(FMAX X_1 X_2 ... X_N) [NoSpread Function]

Returns the maximum of X_1, X_2, ..., X_N. **(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

7.5 Other Arithmetic Functions

(EXPT *A N*) [Function]

Returns $A \uparrow N$. 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, an error is generated, **ILLEGAL EXPONENTIATION**. If *N* is floating and either too large or too small, an error is generated, **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]

X is a number between -1 and 1 (or an error is generated). 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.

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

Similar to **ARCSIN**. Range is 0 to 180, 0 to "PI".

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

Similar to **ARCSIN**. Range is 0 to 180, 0 to "PI".

(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 "PI"), i.e. the result is in the proper quadrant as determined by the signs of *X* and *Y*.

(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** described below.

(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
```