QUANTUM THEORY PROJECT
FOR RESEARCH IN ATOMIC, MOLECULAR, AND SOLID STATE
CHEMISTRY AND PHYSICS
UNIVERSITY OF FLORIDA, GAINESVILLE, FLORIDA

INTEGER ARITHMETIC FUNCTIONS IN MBLISP

PROGRAM NOTE # 6

15 July 1963

ABSTRACT

In addition to the primitive functions contained in MBLISP which perform operations on integers in the range 0-2**15, a number of LISP functions are described by which simple arithmetic operations belonging to a number of systems, such as complex numbers, matrices and vectors, may be performed.

ACKNOWLEDGEMENTS

The coding for the arithmetic functions was mostly done while I was a guest, first of the Yale Computer Center, and then the Computer Center of the Applied Physics Laboratory of Johns Hopkins University. During a two month period since then, they have been in continual use by the students of my LISP programming seminar, and thus appear to be substantially debugged.

Since LISP was conceived as a list processing language, one has to follow drastic techniques, reaching back to the Paeno postulates, to employ the natural numbers. In spite of this limitation, one can accomplish quite useful calculations from such first principles. Yet one can quickly reach a state of considerable aggrevation with primarily numerical operations, and some machine language arithmetical capability is essential. The functions described in this report do not yet allow the full flexibility possible with the computer's vocabulary of floating point operations, but alleviate the problem considerably.

The style employed here was chosen deliberately, for it allows us to have arithmetic operations with a minimum of disturbance to the LISP machinery. The reason is that the data linkages used by the list structure are employed in an immediate manner, so indicated with the aid of a flag bit, as data rather than linkages. Experience with such functions has the further advantage that were it possible to think of machine design, one would have some idea of the utility of non-arithmetic flag bits, or of an arithmetic unit which performed its arithmetic operations on only part of a word, leaving the remainder for control purposes.

Most of the development of MBLISP was completed while I worked at RIAS; it is assuredly a pleasure to acknowledge the interest and wholehearted cooperation of the Digital Computations Section of MARTIN Baltimore, as well as Mr. Welcome W. Bender, the Director of RIAS. Continued development has taken place with the support of the Computer Center of the University of Florida, to the direction, management and operating staff of which we are indebted for their never failing interest and attention.

Harold V. McIntosh

Gainesville, 15 July 1963

ARITHMETIC FUNCTIONS

Since most of the computers on which one would wish to implement LISP lack a non-arithmetical flag bit, the introduction of arithmetical data words on the same par with list linkages causes no small amount of inconvenience. The functions described here use the 15 bit decrement of the IBM 709 as a binary number, with a prefix 2 (PTW) to indicate that the number is not a data linkage. Nevertheless they are described in a thoroughly hardware-independent fashion so that when flag bits become very common, no change in existing LISP programs will be required to increase the number of figures handled as a unit.

The arithmetic functions themselves each have two values rather than one. In this way none of the information concerning the operation is lost, although it can be discarded by the appropriate auxiliary functions. For example, since the product of two 15 bit numbers is a 30 bit number, the two values of the product yield respectively the high and low order bits of the product. The functions produce two values rather than a list of their two values in order not to place the burden of constructing a list upon purely arithmetic functions, as well as to yield a slight improvement in their operating efficiency.

The auxiliary functions to be used to select one of the two values are:

(1STVAL (G X Y)) which selects the first value of the two-valued function G.

(2NDVAL (G X Y)) which selects the second value of the two-valued function G.

Under the action of 1STVAL or 2NDVAL the value not selected is lost. If one wishes to use both values of such a function he should write:

```
((LAMBDA (U V) (...)) (G X Y))
```

whereupon U will become the first value of G, V the second.

Being operators rather than functions, 1STVAL and 2NDVAL may be written anywhere in a program, even as functions of no variables. Good coding practice, however, demands that they enclose their two-valued function as an argument so that the time of their operation will be precisely known.

The purely arithmetical functions have been introduced within the system with the character $ appearing as a prefix to their name. The purpose of this is to render these names unusual, so that they will clearly be understood to be system functions, not ordinarily used by a programmer. They would be embedded in more sophisticated arithmetic functions having more universally accepted names. Observation of this convention will permit definitive versions of functions using arithmetic operations to be written, while still allowing a certain freedom for experimentation in the actual constitution of the primitive arithmetic functions.

The arithmetical functions are:

($PLUS X Y) The first value is a predicate assuring us that overflow has not occurred in the sum; its value is F for overflow, T for no overflow. The second value is the actual numerical sum, modulo overflow. Both X and Y are assumed to be positive.

($MINUS X Y) The first value is a predicate telling whether the difference X-Y is positive, for which it is T; if negative F. The second value is the absolute value of the difference. Both X and Y are assumed to be positive.

($TIMES X Y) The first value is a numeral, the first 15 bits of the product, or more generally the highest order bits of the product. The second value, also numerical, yields the last 15, or the low order bits of the product. If X and Y are interpreted as integers, 2NDVAL would normally be used to select their product, while if they were regarded as fractions, 1STVAL would be used. Since the representation of the numeral is binary, only octal or binary fractions may be used conveniently, which somewhat limits the usefulness of the latter represnetation.

($DIVIDE X Y) The first value is the quotient; the second value the remainder. The latter can be used in calculating a number modulo a base.

Certain words of caution apply to the use of these functions. First, they assume that their arguments are positive integers, so that one must already write a composite LISP function to deal with signed integers. This restriction is partially an artifact of the details of construction of the MBLISP processor, which reserves the sign bit of a memory cell as a flag for the garbage collector. The ability to deal directly with signed integers is sufficiently valuable that the processor will probably be reorganized accordingly.

Secondly, although we speak of the arguments of the arithmetical functions and certain of their values as being numerical, these functions make no effort to make any identification of their arguments. If the arguments are numerical, the values will result properly, but the addition, for instance, of two non-numerical arguments will result in a non-numerical second value. The difference of non-numerical arguments presents a particular hazard to the unwary programmer, for MBLISP detects atoms by a test of numerical magnitude, NIL, which terminates lists, being the lowest atom. Thus the ALIST may appear to be prematurely terminated by such a difference.

Some static test is necessary to distinguish numbers from addresses at least in that part of the memory store which contains the list structure. For instance, the garbage collector must be restrained from confusing the two cases; by either attempting to save spurious list structure, or falsely interpreting an end of list and neglecting to save the remainder. Likewise, since lists and numbers may be passed back and forth between programs which have no dynamic ability to make the distinction, it must be made statically.

For example, PRINT must be forewarned whether to make an octal or decimal conversion before attempting to deliver its argument to the output tape.

For these reasons, a predicate is provided to sense the prefix which denotes a number.

(NUM X) is true if its argument is a numeral, false otherwise.

This flag is presumed set by the conversion routine which first creates the numeral; all functions having a numerical value (or values) will automatically preserve the flag. The functions CAR, CDR, CONS will automatically copy the numerical flag when it is attatched to their arguments.

Two functions allow us to make comparisons of numerals:

(EQ X Y) takes the value T if its arguments are both numerals and both equal. It will take the value F if either argument is a numeral and the other not. In addition it has its usual significance for atoms and lists.

(SL X Y) is true if its first argument, X, is strictly less than its second argument, Y; otherwise it is false. It ignores the numeral flag, so that it may also be used to compare an absolute core address (which is the way atoms and lists both are actually represented in the memory store) to a numeral.

Four functions may be used to convert an atomic symbol to the number which it represents, or back again:

(DEC N) The value of this function is the binary equivalent of the atomic symbol N which is supposed to be a string of not more than 5 decimal digits.

(OCT N) makes the corresponding octal conversion of the atomic symbol N, which is a string of not more than 5 octal digits. In both cases the capacity of the conversion is $32768_{10} = 77777_8$

(UNDEC X) converts the number X into the atomic symbol representing the decimal equivalent of X. No checking is performed to ensure that X is in fact a numeral, so that if it is a list connector which appears as the argument X, the result will be the decimal address of the referenced expression in the memory store.

(UNOCT X) makes the analogous conversion of X into its octal equivalent as an atomic symbol.

For the present, no conversion of floating point numbers or of exponential notation is attempted. Thus, only integers can be dealt with directly.

There is some question whether the conversion of a numerical atomic symbol into its binary equivalent should be explicitly commanded or not. Often, one adopts the convention that any string of numerals is to be converted upon initial readin, and any string containing other markings, such as decimal points or E's should be dealt with accordingly. Such a decision presents the difficulty that one cannot use numerals to designate variables, at least in the present constitution of the porcessor, for instance. Admittedly such practices are rare, but they do occur. Also, the automatic conversion releives the programmer of the corresponding responsibility. We nevertheless prefer to order the explicit conversion as necessary.

Continual conversion is wasteful and time consuming, so that in those contexts where it is known that all arguments of a series of functions are numerical, it is preferable to make an initial numerical conversion, and a final conversion to allow the printing of the functionvalues. This precaution, of converting the arguments must also be followed in tracing, or in printing intermediate results.

While numerals may easily enough be located in output by the aid of the predicate NUM, one must establish conventions concerining input expressions. Rather than disintegrate each atom, to see whether it is a digit-string, it is more convenient to let context determine numerical arguments. In that case, the argument is automatically converted, whether or not its constituent atoms are numerical, and it is the responsibility of the programmer to follow the proper format for such expressions.

The format in question is the following: All atoms are presumed to represent numbers save the minus sign, -. A positive number is a single atom, as 0, 1, 3, 967, etc. A negative number is written as a list of two atoms, the first a minus sign, as (- 1), (- 911), etc. Other quantities, such as vectors, complex numbers and the like, are expressinns having these former quantities as elements. For example, we write the complex number x + iy as (x y), instances of which would be 1 = (1 0), i = (0 1), 3 - 2i = (3 (- 2)), and so on.

The following three functions are useful for performing conversions:

```
(NUMBETHERE (LAMBDA (L) (COND
        ((EQ L (QUOTE -)) L)
        ((ATOM L) (DEC L))
        ((NULL L) L)
        ((AND) (CONS (NUMBETHERE (CAR L))
                     (NUMBETHERE (CDR L)))) )))

(NUMBEGONE (LAMBDA (L) (COND
        ((NUM L) (UNDEC L))
        ((ATOM L) L)
        ((NULL L) L)
        ((AND) (CONS (NUMBEGONE (CAR L))
                     (NUMBEGONE (CDR L)))) )))
        (PRINT* (LAMBDA (L) ((LAMBDA (X) L)(PRINT (NUMBEGONE L)))))
```

This last function avoids a repetitous conversion at the price of an additional LAMBDA; for short expressions one might also write an alternatitive definition,

```
        (PRINT* (LAMBDA (L) (NUMBETHERE (PRINT (NUMBEGONE L)))))
```

The configuration, ...(NUMBEGONE (... (NUMBETHERE L)))... must occur at some level of any LISP function handling numerical quantities.

It is not difficult to systematically produce a series of compound functions designed to perform arithmetic operations upon specialized sorts of numbers. Since the addition of positive integers, for instance, is different from that for signed integers, vectors, complex numbers, and other quantities, it is convenient to give the particular operations systematic names which denote the type of quantity to which they are to be applied. The unadorned symbols, + - * / ** and so on, are preferably reserved to have appropriate meanings for a definite problem area in which a series of functions are applied.

Toward this end, the following system of prefixes may be used to designate different classes of operations:

I	operations on positive integers
G	operations on signed integers
R	operations on rational numbers
K	operations on complex numbers
V	operations on vectors
M	operations on matrices
Q	operations on quaternions
S	scalar operations on vectors and matrices

<u>I operations</u> The arithmetic functions, $PLUS, $MINUS, $TIMES, $DIVIDE are designed to treat their arguments as positive integers (or zero). However, they possess two values, to allow the possibility of writing multiple precision functions using a list of integers in the range 0-77777_8, or to allow the detection of overflow and the like. If one assumes that he is dealing with small positive integers always, this additional information can be neglected, and it is only necessary to compose each of the functions with 2NDVAL in order to discard it. Thus we define:

```
        (I+ (LAMBDA (X Y) (2NDVAL ($PLUS X Y))))

        (I- (LAMBDA (X Y) ((LAMBDA (X Y) (IF X Y
                (LIST (QUOTE -) Y))) (%MINUS X Y))))

        (I* (LAMBDA (X Y) (2NDVAL (%TIMES X Y))))

        (I/ (LAMBDA (X Y) (1STVAL ($DIVIDE X Y))))

        (REM (LAMBDA (X Y) (2NDVAL (%DIVIDE X Y))))
```

All these functions yield positive values save I-, which will yield a negative integer, represented as (- n), if its second argument is larger than the first, so that x-y is actually negative. I/ yields the integral part of the quotient x/y; REM the remainder.

G operations To represent signed integers, we adopt the convention that an isolated numeral is to represent a positive number, while the list (- n), in which n is a numeral, represants the negative number -n. Consequently the predicate NUM may be used as the test for positiveness of a number. All the signed operations procede by an enumeration of the four possible sign combinations of the two arguments.

```
(NEG (LAMBDA (X) (IF (NUM X) (LIST (QUOTE -) X)
        (CADR X))))

(G+ (LAMBDA (X Y) (IF (NUM X) (IF (NUM Y) (I+ X Y)
        (I- X (CADR Y))) (IF (NUM Y) (I- Y (CADR X))
        (LIST (QUOTE -) (I+ (CADR X) (CADR Y)))) )))

(G- (LAMBDA (X Y) (G+ X (NEG Y))))

(G* (LAMBDA (X Y) (IF (NUM X) (IF (NUM Y) (I* X Y)
        (LIST (QUOTE -) (I* X (CADR Y))) )
        (IF (NUM Y) (LIST (QUOTE -) (I* (CADR X) Y))
        (I* (CADR X) (CADR Y)) ) )))

(G/ (LAMBDA (X Y) (IF (NUM X) (IF (NUM Y) (I/ X Y)
        (LIST (QUOTE -) (I/ X (CADR Y))) )
        (IF (NUM Y) (LIST (QUOTE -) (I/ (CADR X) Y)) )
        (I/ (CADR X) (CADR Y)) ) )))

(GREM (LAMBDA (X Y) (IF (NUM X) (IF (NUM Y) (REM X Y)
        (REM X (CADR Y)) ) (IF (NUM Y) (LIST (QUOTE -)
        (REM (CADR X) Y)) (LIST (QUOTE -) (REM (CADR X)
        (CADR Y))) ) )))
```

K operations Complex numbers are conveniently represented as a list consisting of their real part followed by their imaginary part. Both real and imaginary parts may be presumed to be signed integers; thus one would represent 3-5i as (3 (- 5)). One must observe the precautionthat complex numbers with imaginary part zero are only isomorphic to the signed integers and not identical to them, as may be seen by contrasting 1 with (1 0).

```
(KCONJ (LAMBDA (Z) (LIST (CAR Z) (NEG (CADR Z)))))

(K+ (LAMBDA (W Z) (LIST (G+ (CAR W) (CAR Z))
        (G+ (CADR W) (CADR Z)))))

(K- (LAMBDA (W Z) (LIST (G- (CAR W) (CAR Z))
        (G- (CADR W) (CADR Z)))))

(K* (LAMBDA (W Z) (LIST (G- (G* (CAR W) (CAR Z))
        (G* (CADR W) (CADR Z))) (G+ (G* (CAR W) (CADR Z))
        (G* (CADR W) (CAR Z))) )))

(S/ (LAMBDA (A Z) (IF (NULL Z) Z (CONS (G/ (CAR Z) A)
        (S/ A (CDR Z))) )))

(K/ (LAMBDA (W Z) (S/ (CAR (K* Z (KCONJ Z)))
        (K* W (KCONJ Z)) )))
```

V operations Vectors, being n-tuples of numbers, are readily represented as lists, upon which appropriate operations may be defined. In fact, one such operation, S/, was included in the definitions for complex operations in a sufficiently general form to be useful for dividing each element of a vector by a certain scalar.

```
(V+ (LAMBDA (X Y) (IF (NULL X) Y (CONS
        (G+ (CAR X) (CAR Y)) (V+ (CDR X) (CDR Y))) )))

(V- (LAMBDA (X Y) (IF (NULL X) Y (CONS
        (G- (CAR X) (CAR Y)) (V- (CDR X) (CDR Y))) )))

(S* (LAMBDA (A X) (IF (NULL X) X (CONS
        (G* A (CAR X)) (S* A (CDR X))) )))

(IP (LAMBDA (X Y) (IF (NULL X) (DEC (QUOTE 0))
        (G+ (G* (CAR X) (CAR Y)) (IP (CDR X) (CDR Y))) )))
```

For testing purposes, it is convenient to have a way of applying an arithmetic function to atomic arguments, without having to continually write DEC and UNDEC in the program. The style of the control function used for such purposes may be varied according to the number of arguments expected by the function to be applied.

```
(OPER (LAMBDA (O X Y) (NUMBEGONE (O (NUMBETHERE X)
        (NUMBETHERE Y)) )))
```

A general purpose function of this nature, which will apply a function irrespective of the number of its arguments, can be written using the functions EVAL and ALIST, which give direct access to the corresponding portions of the interpreter:

```
(OPTEST (LAMBDA L (NUMBEGONE (EVAL (CONS (CAR L)
        (APPQ (NUMBETHERE (CDR L)))) (CDDR (ALIST))) )))
```

where we have

```
(APPQ (LAMBDA (L) (IF (NULL L) L (CONS (LIST (QUOTE QUOTE)
        (CAR L)) (APPQ (CDR L))) )))
```

To illustrate the use of OPTEST, we might write

```
(OPTEST V+ (1 (- 3) 2) (0 2 (- 1)))
```

which will result in the vector sum (1 (- 1) 1) of the last two arguments.

Many specialized representations of numbers may readily be devised in LISP; for instance wh-n one is dealing with the complex nth roots of unity, it is more natural to represent exp(2pi i k/n) as the pair (K N) with the rule that

```
(K1 N1).(K2 N2) = ((K1.N2+K2.N1) N1.N2),
```

the left hand sum being reduced modulo N1.N2.

The following function will perform such a calculation:

```
(KE* (LAMBDA (X Y) (IF (EQ (CADR X) (CADR Y))
        (LIST (REM (I+ (CAR X) (CAR Y)) (CADR X))
        (CADR X))) (LIST (REM (I+ (I* (CAR X) (CADR Y))
        (I* (CADR X) (CAR Y))) (I* (CADR X) (CADR Y)))
        (I* (CADR X) (CADR Y))) )))
```

It is likewise possible to devise functions which will handle integers represented as a list of exponents of their prime factors, or other specialized representations.

4/27/63
7/15/63