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

OPERATORS FOR MBLISP

PROGRAM NOTE # 9

26 July 1963

ABSTRACT

The primitive operators contained within MBLISP are described, together with an example of their application. The operators fall into three categories: list processing operators, input-output operators, and data movement operators. Some of the operators are predicates; this permits programs to be constructed using them, controlled by the functions AND and OR.

ACKNOWLEDGEMENTS

The choice of an operator complement for MBLISP has depended to a great extend on Lowell Hawkinson's thorough analysis of LISP processors and the resultant insight into the most efficient form of the recursive processor. Considerable experimentation has gone into the choice of input-output operators, as well as data movement, but they owe their final form to the experience gained in the use of a preliminary set by T.A. Brody in his studies of LISP compilation. Many of the studies performed when we used much clumsier operators were carried out by R.A. Smith, Jr., and were indispensible in setting the style of the present operators.

Most of the development of MBLISP was completed during the period of my employment by RIAS, and it is a pleasure to thank the director, Mr. Welcome W. Bender, for his continued interest in and support of, this work. It is not often realized how much the success of the development of any program depends upon the skill, interest, and cooperation of the members of the computing center at which the work is done, and for this reason it is a particular pleasure to express my gratitude to the Digital Computations Section of MARTIN BALTIMORE. Were it not for their continued willingness to make their programs and facilities available, continued development of MBLISP would have been impossible.

The time required to verify the operators reported in this note was made available at the computer center of the University of Florida, for which time I am grateful to the committee and personnel associated with the operation of the Center, as well as their courtesy in making time available for Prof. Brody.

Harold V. McIntosh

Gainesville, 26 July 1963

OPERATORS

Due to the recursive nature of its operation, no information in the memory store which is accessible to a LISP program is ever desrtoyed, because it may have to be replaced in its original state at some later stage in the execution of the program. Gradually, information is abandoned, in the sense that no operation in the program can ever locate it again. From time to time the memory store may be examined, in order to recover the abandoned cells for further use. During this rennovation, or garbage collection as it is usually called, information will be destroyed, so that to be precise we should say that no information available to the program is ever altered in any manner.

Although different technical schemes may be imagined to realize such a manner of operation, the one generally chosen is to have a large resoirvoir of cells, called the vacuum, available for use by the program. The only primitive LISP function which writes information is CONS; all the others read information. Thus the only function which can modify the memory store is CONS. Rather than search the memory for a cell containing the desired linkages, if indeed one exists, CONS removes one cell from the vacuum, which, modulo garbage collection, has never before been used, and writes the necessary linkages into it. For example, the LISP function (APPEND U V) which adjoins the list V to the end of the list U in no way modifies either the list U or the list V. Rather, a new list is constructed whose head is an entirely new copy of the list U and whose tail points to the list V. Consequently if the bound variables U and V appear anywhere else in the same expression they still refer to the same lists, and not the one gotten by tacking V onto the end of U, which would have changed the meaning of the symbol U.

Needless to say, such a procedure is quite extravagant of memory cells and would ordinarily be felt to be justified only when one felt that he had to retain the identity of the list U intact for further use by the program. For some but hardly all programs this would be true.

Although automatic recursion is sometimes much more rapid from the point of view of writing a program, the necessary saving and unsaving of registers takes its toll on the operating speed, and often by paying attention to the organization of the memory store, such waste motion may be avoided.

In addition to the situations in which one voluntarily foregoes the recursive mode of operation, there are others which are inherently irreversible. Thus, if one were to introduce a LISP function which would read one record from the input apparatus to a certain portion of memory, repeated use of this function would result in repeated records being read in, most likely destroying the old records in the process.

For such reasons we distinguish operators from functions. The distinction is that a LISP function receives its arguments from a certain pushdown list, and delivers its values to this same pushdown list. By contrast, an operator may receive information from any register as well as modifying any register. Of course, in addition it may work with the pushdown list patronized by the functions, so that one must recognize a mixed case.

In planning a formal programming language, it is desirable to keep the number of primitive operators to a minimum, as well as the number of primitive functions. For list processing purposes, only three operators are needed. These allow one to locate a free cell in the vacuum, overwrite an address linkage, or overwrite a decrement linkage. More extensive operators are needed to perform input-output operations, movement of data, or arithmetic operations. However, the three list processing operators are:

(SAR E X) causes E to become (CAR X). Its value is T.

(SDR E X) causes E to become (CDR X). Its value is T.

(BILE) is a function of no variables whose value is a fresh cell from the vacuum [PZE *,,NIL]. Its value is thus an empty set. Repeated usage of (BILE) produces a string of fresh cells.

From these primitive operators certain variants may be constructed which will be useful in particular contexts. For example, SAR and SDR have been designed as predicates, to facilitate their inclusion in programs. However, instead of returning the value T to the LISP processor, it would shorten the corresponding program if they would occasionally return other values. Although the variants might be written as LISP functions of SAR and SDR, in actual practice they would profitably be coded as machine language programs. For example,

(XAR E X) causes E to become (CAR X). Its value is the displaced value of (CAR X).

(XDR E X) causes E to become (CDR X). Its value is the displaced value of (CAR X).

Their respective LISP definitions are:

```
(XAR (LAMBDA (E X) ((LAMBDA (Z) ((LAMBDA (T) Z)
        (SAR E X))) (CAR X))))

(XDR (LAMBDA (E X) ((LAMBDA (Z) ((LAMBDA (T) Z)
        (SDR E X))) (CDR X))))
```

Yet another variant would yield the argument X as the value of SAR and SDR instead of the overwritten linkage. In the latter case, one would perhaps be using CAR of an expression as a temporary storage position, and is simultaneously updating the location, and retrieving its old value. By receiving the new value E instead of the displaced value, one would have a variant which would make it convenient to store the same expression in a series of locations by simply nesting the appropriate SAR or SDR variants with appropriate second arguments. Finally, in the last variant we have proposed, a series of quantities could be stored at the same or a chain of addresses, since the value of the operator would always be the last location at which the storage took place.

Besides operators which are useful in the various aspects of list processing, another important class may be used for data transmission purposes. The details of this latter class are somewhat machine dependent in that they will vary according to whether the memory store is decimal or binary, or whether it has fixed or variable word length and so on. They are in this sense only weakly machine dependent; the exigencies of the IBM 709 design make the following collection seem desirable:

($READ Z) causes words to be read from SYSPIT (A-2, decimal) as specified by the title Z.

($WRITE Z) causes words to be written on the printed output tape, SYSPOT, (A-3, decimal) as specified by the title Z.

($PUNCH Z) causes words to be written on the punch tape PCHTAP (B-4, decimal), as specified by the title Z.

Each of these operators makes use of the title of a buffer area. By a title we mean a description of an array [PTH ARRAY+N,,N] which occupies one cell, containing a flag indicating an array, the number of words comprising the array, and the final word plus one. The title itself occupies this last location unless other provision is made, allowing the garbage collector to recognize arrays and save them if necessary. All arrays are referred to through their titles.

The functions $READ, $WRITE, and $PUNCH really do little more than activate the relevant IOCD command. They are all predicates, whose value is T if they are functioning without incident. However, they are only semipredicates, and may return information to the LISP processor concerning possible error conditions, end of file or end of tape, and so on.

The choice of three functions such as these limits the number of tapes accessible to the LISP programmer, and no provision has been made for reading or writing binary tapes. All these variations may eventually be provided by internally compiled operators, so that we have described only the three essential for ordinary operation and made them inherent operators in the system.

The reason that these three operators have been given names commencing with the character $ is to indicate that they are not intended to be a portion of ordinary programs. The reason for this is that they make no provision of themselves for controlling various error conditions or other impediments to the free flow of information to and from the input-output mechanism. Concern with these latter conditions is not properly a part of a LISP program, and should be accomplished by standard data transmission operators which incorporate the proper tests and remedies to handle such problems. Should this latter treatment vary, or the style of input-output vary, the standard operators may be redefined without affecting any LISP programs in the slightest.

In order to allow a LISP program to actively manipulate the memory store, we further introduce operators which will allow bits to be read from or stored in desired locations. It is convenient to distinguish a numerical from a BCD mode, the latter treating the hollerith characters as the basic units, while the former treats octal numbers as the units. Again this choice is moderately machine dependent. Rather than carry an excessive number of arguments continually, each of the main functions is provided with a satellite which initializes it to work from a given array. It will then work through the array, from left to right, low address to high address until the limit to which it has been set has been exceeded. This event will produce a characteristic reaction by the function.

(PACK X) is a predicate, which stores its argument X in the location to which it has been set, and whose value is T if additional space remains in the array to pack an additional character. Otherwise its value is F.

(PACSET Z) is an operator predicate whose value is T and which prepares the operator PACK to store successive characters, left packed, into the array whose title is Z.

(STORE X) is a predicate, entirely analogous to PACK, save that its argument is a single atomic symbol representing an octal digit, and that only 3 bits at a time are stored, rather than 6.

(STOSET Z) is the analogue of PACSET, initializing STORE to the array whose title is Z. It takes the value T.

(DISINT) is a function of no variables, whose value is either the next character in the array to which it is set, or else () if none remain. Repeated usage from an exhausted array will continue to yield ()'s. Characters are removed from the array 6 bits at a time, non-destructively. The array is considered to be exhausted when either its upper limit is reached, or the illegal hollerith character 77 (which is used to fill out words) is encountered.

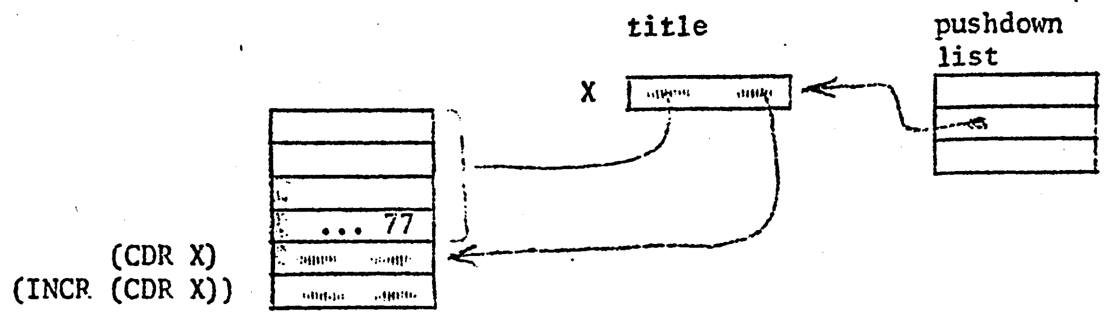
(DISSET Z) is a predicate, value T, which initializes DISINT to the array whose title is Z.

(DISSOC) yields the contents of the array ro which it is set, 3 bits at a time, in the form of octal digits. Its value is () when it reaches the upper limit of its array. The readout is non-destructive.

(DSCSET Z) is a predicate, whose value is T, which initializes DISSOC to the array whose title is Z.

Operators are typical of iterative programs, which are rather the antithesis of LISP. To follow completely the iterative format pertaining to operators would logically lead to the introduction of the LISP "program feature" wherein sequences of statements would be written in the style of FORTRAN or ALGOL. Such a step is unnecessary, and may be avoided by the use of operator predicates, allowing such LISP functions as IF, COND, AND, and OR to exercise control over the program.

Another characteristic of iterative programs is that the variable names have a permanent meaning and cummulate the effect of operations performed upon them, whereas in a recursive program variable names are transiently bound and while bound never vary. By properly arranging the internal structure of atomic symbols, and using the operators which we have defined, it is possible to arrive at a system which may be used both redursively and iteratively. The diagram below shows the canonical arrangement of an atomic symbol:



The internal constitution of MBLISP is such that whenever an atom appears in a LISP program, it actually appears as an address in the memory store. That address is the address of a title, which refers to an array which contains the print name of the atom, its function definition and its value. By the print name we mean the string of hollerith characters by which it is represented in printed output and input. These are allocated six at a time to whatever number of words, not exceeding six, required to hold them. Any vacant space at the right of the last word is filled with 77's, rather than blanks (60's), for convenience in collating, as well as to allow blanks to appear in internally stored messages. (CAR X) is a numeral, which is the size of this array. (CDR X) is the cell following, which is used to contain information, if any, concerning the use of that atomic symbol as a function name. If it is a primitive function or a machine language function,the decrement of this cell contains the address of the subroutine corresponding to it. If it is a defined function, the decrement contains the definition, so that for example, (CADR (QUOTE APPEND)) is the definition of APPEND. The address of this cell is the address of the appropriate subroutine to cause the arguments of the function to be evaluated; in the case of a defined function, this is the subroutine EXPARG of the processor.

The next cell, (INCR (CDR X)) in the diagram, holds the "value" of the atom in its decrement. INCR is equivalent to (2NDVAL ($PLUS X (DEC (QUOTE 1)))), and is used to add 1 to its argument, adsumed numerical. We may regard the value simply as a special storage space associated with the atom

Since this arrangement is subject to slight change as growing experience warrants a shifting of priorities for different storage locations, it is better to introduce certain special functions to manipulate this storage, which may be later redefined without affecting the functions using them.

```
(SET (LAMBDA (X Y) (SAR X (INCR (CDR Y)))))

(VAL (LAMBDA* (X) (CAR (INCR (CDR X)))))

(SEQ (LAMBDA* (X) (XAR ((CADR X) (CAR (INCR (CDR X))))
       (INCR (CDR X)) )))

(XEC (LAMBDA* (X Y) (SAR (X (CAR (INCR (CDR Y))))
       (INCR (CDR Y)) )))

(SHELVE (LAMBDA (X Y) (SAR (CONS X (CAR (INCR (CDR Y))))
       (INCR (CDR Y)) )))

(UNSHELVE (LAMBDA* (X) (CAR (XAR (CDAR (INCR (CDR X)))
       ((INCR (CDR X)) ) )))

(QUEUE (LAMBDA (X Y) (SAR (APPEND (CAR (INCR (CDR Y)))
       (LIST X)) (INCR (CDR Y)) )))
```

The significance of these functions is the following: With every atom there is supposed to be associated an abstract quantity called its value. In RLIST LISP this would be the same as the value to which it was bound if it were a bound variable; however it may be simply regarded as an abstract property. The functions enumerated above manipulate this value, and as occasion demands deliver it to the LISP processor's pushdown list.

(SET X Y) causes the value of Y to become X. Ordinarily Y would be a quoted atom, since SET is defined in terms of LAMBDA. Any previous value is lost, and the value is retained until altered by another operator.

(VAL X) yields the value of X. Since it is defined in terms of LAMBDA*, its argument need (indeed must) not be quoted.

(SEQ X) regards S as a function of one variable, which it applies to the value of X to obtain a new value, which displaces the old value; the latter becoming the value of SEQ reported to the processor. SEQ is an adaptation of the concept of a sequence mode or a sequenced variable used by Perlis in connection with THREADED LISTS. (VAL X) corresponds to his notation x¢, while (SEQ X) corresponds to x*. The motivation of (SEQ X) is to have a way that X can automatically be replaced by the next value of a predetermined sequence each time that it is consulted. (VAL X) offers us an opportunity to consult X arbitrarily often without going on to the next value, however.
As with VAL, the argument of SEQ should not be quoted.

(XEC X Y) causes the value of Y to be replaced by (X Y). It is a predicate, always taking the value T. It is used when one has a variable which is to be sequenced in several alternative ways. It is no longer possible to give the sequence functions all the same name as the sequence variable, so that XEC may be invoked to apply the chosen one. Beyond doubt, one could define appropriate functions to apply a particular sequence function to a variable, so that XEC stands as a sort of general case.

(SHELVE X Y) replaces the value of Y by (CONS X (VAL Y)), so that it acts like the shelving operation in Yngve's COMIT. It is a predicate, having a value of T to assure us that the shelving has taken place. The "shelf" is simply a list hung under the atom Y, which must be set to () or some other list initially. The shelving operation consists in simply inserting the given expression X on the front of this list.
Since it is defined by LAMBDA, so that the expression X may be evaluated before being shelved, it is necessary to quote the argument Y.

(UNSHELVE X) deletes the first item from the shelf X, taking that item as its value. X should not be quoted.

(QUEUE X Y) works in the same fashion as SHELVE save that X is placed at the end of the "shelf", or list hanging under Y, rather than on the front.

Programs written in terms of these functions remain entirely ignorant of the structure of the atoms, and thus will be uninfluenced by changes in this latter structure.

As an example of an application of a sequenced variable, consider the following definition of a sequence function:

```
(WRIBU (LAMBDA (X) (IF (EQ X ($WRIBU1)) ($WRIBU2)
        ($WRIBU1) )))
```

($WRIBU1) and ($WRIBU2) are two functions whose values are the titles of two arrays each holding one record for the output tape, SYSPOT. We wish to construct the output records alternately in one array and then in the other, so that we may use one while the other is being written on tape. WRIBU is then a variable whose values under (SEQ WRIBU) alternate between these two arrays.

Operator predicates, in conjunction with LISP's innate ability to define functions, yield a very powerful technique for constructing programs. By using the Boolean functions AND and OR, we can require respectively that as many as possible of a series of operations be performed, or that as many as necessary be performed. There is no provision for an analogue of a transfer order, but the possibility of a function definition allows a portion of the program to be regarded as a unit and repeated as often as desired, so that the repetition characteristic of an iterative program can still be achieved.

In order to illustrate the construction of a program, as well as to exhibit a typical example of the usage to which the operators which we have defined may be put, let us consider the function COMPACTIFY. Its purpose is to produce a card deck serving as the source program for a LISP function, from which all superfluous blanks have been removed. When subjected to this treatment, the program for a working function may be reduced to 1/2 or 1/3 the number of cards which would be found convenient during the testing stages, when separate statements are placed on individual cards and ample room is left on each card for legibility and modification. When a card deck has to be read on-line frequently, this saving in volume can result in an appreciable saving in time.

```
(COMPACTIFY (LAMBDA L (AND
        (SET ($PCHBU1) (QUOTE PCHBU))
        (PACSET (VAL PCHBU))
        (NOT (///))
        ($PUNCH (SEQ PCHBU))
        (PACSET (VAL PCHBU))
        (COMPACTIFY* L)
        (NOT (///))
        ($PUNCH (SEQ PCHBU))
        (PACSET (VAL PCHBU)) )))
```

COMPACTIFY is a function of an arbitrary number of arguments, which will compactify each of them in turn. However, it is only the control function which initializes a number of variables, prepares a blank card (with the help of the function ///) to precede and follow the output, and calls on the satellite COMPACTIFY* for the actual compression.

```
(COMPACTIFY* (LAMBDA (L) (OR
        (NULL L)
        (AND
                (NOT (///))
                ($PUNCH (SEQ PCHBU))
                (PACSET (VAL PCHBU))
                (COMPEXPR (CAR L))
                (NOT (///))
                ($PUNCH (SEQ PCHBU))
                (PACSET (VAL PCHBU))
                (COMPACTIFY* (CDR L)) ) )))
```

COMPACTIFY* in its turn simply ensures that a blank card intersperses each case, as well as ensuring that the terminal line is completed with blanks and punched out. Since the termination requires three lines of program it could perhaps profitably be defined as a separate function.

```
(COMPEXPR (LAMBDA (E) (OR
        (AND (ATOM E) (PUNCHATOM E))
        (AND (PUNCHATOM (LPAREN))
             (COMPLIST E)
             (PUNCHATOM (RPAREN)) ) )))
```

COMPEXPR distinguishes whether the expression to be compactified is an atom or a list. In the former case, the atom is sent to PCHTAP through the intercession of PUNCHATOM, whilE in the latter, the list is surrounded by parentheses and broken into its constituent parts by COMPLIST.

```
(COMPLIST (LAMBDA (L) (OR
        (NULL L)
        (AND
                (ATOM (CAR L))
                (PUNCHATOM (BLANK))
                (COMPLIST* (CDR L)) )
        (AND
                (COMPEXPR (CAR L))
                (COMPLIST (CDR L)) ) )))

(COMPLIST* (LAMBDA (L) (OR
        (NULL L)
        (AND
                (ATOM (CAR L))
                (PUNCHATOM (BLANK))
                (PUNCHATOM (CAR L))
                (COMPLIST* (CDR L)) )
        (AND
                (COMPEXPR (CAR L))
                (COMPLIST (CDR L)) ) )))
```

These two functions are used alternatively, depending upon whether two consecutive blanks appear in the expression or not. They both terminate upon an empty list, and otherwise see that every subexpression of their argument is compactified.

```
(PUNCHATOM (LAMBDA (X) (AND
        (DISSET X)
        (PUNCHATOM* (DISINT)) )))

(PUNCHATOM* (LAMBDA (X) (OR
        (NULL X)
        (AND
                (OR
                        (PACK X)
                        (AND
                                ($PUNCH (SEQ PCHBU))
                                (PACSET (VAL PCHBU)) ) )
                (PUNCHATOM* (DISINT)) ) )))
```

These last two functions dissect the print name of an atom character by character, and simultaneously place it in the output buffer.

The function PCHBU is a sequenced variable defined analogously to the example WRIBU.

The reason that PUNCHATOM is used consistently, even to store the punctuating blanks and parentheses, instead of PACK, is that it sends a filled line immediately to the output tape, whereas PACK might overflow the buffer area without notice of this fact being taken.