L I S P I T O

A LISP PROCESSOR FOR THE IBM 1620

T. A. Brody
Instituto de Física, UNAM
Asesor Técnico, CNEN

Centro Nacional de Cálculo, Instituto Politécnico Nacional, Unidad Zacatenco
México, D.F.
1 9 6 5

Preface

The present attempt to create a symbol manipulation language processor for a relatively small computer was made because of the possibility it would offer to open up the field of symbol manipulation to the many groups who do not have a large computer at their disposal. This is the case, for instance, in Latin America and in several European countries. The resulting processor has proved useful both in the teaching of symbol manipulation techniques and for several applications of a not too extensive character. Of these, the simulation of a Turing machine in LISPITO has proved pedagogically useful.

The author would like to acknowledge the help he has received from Mr Harold V. McIntosh, who introduced him to the pleasures of LISP and to whom are due many of the ideas incorporated in LISPITO. He would also like to express his thanks to the Centro Nacional de Cálculo of Mexico, which put its IBM 1620 computer at his disposal and whose members helped in more than one way towards the completion of the processor.

Note on Machine Requirements

The processor described in this manual will run on an IBM 1620 machine equipped with the following additional features :

1622 card reader/punch

1623 storage unit to expand core memory to at least 40 000 positions

indirect addressing feature

automatic division feature

floating point arithmetic feature

special instructions 71 (move flag)
72 (transmit numerical strip)
73 (transmit numerical fill)

If more than 40 000 positions of core storage are available, the card deck may be modified to make use of the additional memory.

I. Introduction : The LISP Language

LISP, originally developed by McCarthy[1], may be characterised as a language for applying recursive functions to list structures.

A recursive function (for the theory see, e.g. Davis[2], Péter[3]) is one which is used within its own definition, though with different arguments. Thus n! may be defined recursively as follows :

$$n! = \begin{array}{l} 1, \text{ if } n = 0 \\ n(n-1)! \text{ otherwise} \end{array}$$

Thus $2! = 2 \times 1!$, which in turn $= 2 \times 1 \times 0! = 2 \times 1 \times 1 = 2$.

It will be seen that it is necessary that for some values of the argument the function be defined non-recursively ; otherwise the recursion will not end, and its value (if it exists) could not be obtained in a finite number of steps. The non-recursive part of the definition is known as the terminal condition.

The recursion may be more complicated ; the function may have several arguments, in its definition several terminal conditions and several recursive elements may occur ; the recursion may also be indirect, as when a function f calls another function g , which in turn calls h , in whose definition f is used.

For the machine realisation of a recursive function it is evidently necessary to save the values of its arguments and intermediate results so that, directly or indirectly, it may be called again. There are various methods of dealing with this problem, all making use of a push-down list or stack ; anything stored in such a list does not erase the previously stored information, but pushes it down, so that on calling the push-down list the last element stored is recovered and removed, so that the next call will produce the preceding element. In LISP, the arguments are placed on a push-down list, and if the function is called recursively the old values are pushed down, only to pop up again when the routine is carried out.

There are variants of this method ; in the one adopted in LISPITO, the current values of arguments are placed under the variable names, and the old values are placed on a push-down list from which they are restored at the end of the function execution.

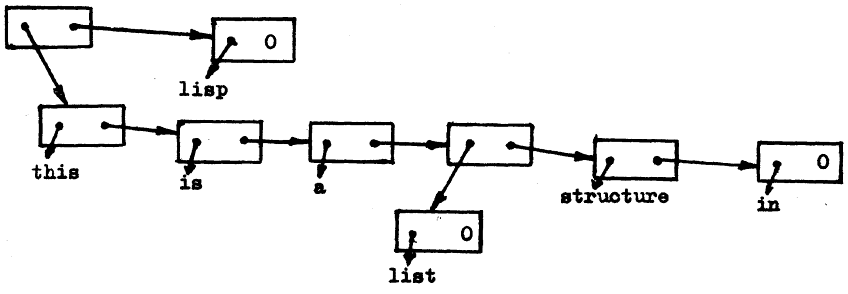
List structures, to which recursive functions are applied in LISP, provide a technique both for eliminating all storage assignment problems as far as the user is concerned and a flexible means of specifying the structural relations between the elements which form the field of interest of LISP. Since essentially these elements are not further analysed in LISP (except under special circumstances ; see e.g. the LISP 1.5 Manual[4]), they are known as "atoms". A linear list is an ordered set of atoms ; it may be empty. A non-linear list, or list structure, is a list whose elements are in turn lists or list structures.

The machine representation of alist is simple ; an atom is an address at which the name and other properties of the atom are stored, and when an atom occurs in a list, it is represented by this address to which that of the next list element is adjoined. The last element of the list carries a suitable indication. This type of representation was developed by Newell, Simon and Shaw[5] and is used in several list manipulating languages.

Outside the machine, LISP represents a list simply by enclosing the sequence of elements composing it in parentheses. Thus an empty list is ( ), and the list containing a and b is (a b) ; blank spaces are used in LISPITO to separate the names of atoms which follow each other, but they are not required if there are parentheses (LISP 1.5 uses a different technique[4]). A list structure will then appear as follows :

((this is a (list) structure in) lisp)

Its representation in the machine would be



Here arrows indicate the elements referred by the addresses at the starting point of the arrows, and 0 in the right-hand part of a rectangle representing a list element indicates that it is the last one in the list or sublist.

Clearly list structures lend themselves to recursive handling in a straightforward manner, provided that any newly constructed lists never overwrite earlier lists which may still be required. For this purpose the memory space available for list building is organised as a single linear list of unused elements, - the so-called vacuum. Whenever a new element is required, it is taken off this list and never made from an earlier list. However, the vacuum may come to an end before processing is ended ; since at this point there will generally be a large number of list structures and fragments of structures in memory which can no longer be referred to, these may be recombined to form a new vacuum. This is carried out by a special routine called the "garbage collector". Its calling is automatic and need not worry the user.

II. The Primitive Functions in LISPITO

LISP has five basic functions, the most important among the primitive or machine functions ; all other functions may be defined in terms of them by means of the recursion and function defining symbols However, it is very convenient to have some further primitives available, both for ease of programming and for running speed.

Of the five primitives that are basic, two permit the analysis of a list :

(car x)* evaluates x and if it is a list, the result is the first element of the list, whether this be an atom or a sublist. If x is not a list, (car x) is undefined.

(cdr x) evaluates the argument x and if it is a list, gives all but the first element of it ; the result is thus always a list. If x contains only one element, (cdr x) = ( ) ; if x is an empty list, the cdr of it is undefined.

Another function synthesises a list :

(cons x y) evaluates both x and y and then creates a list the first element of which is x, while the others are those of y ; x may be anything, but y must be a list, which may be empty. Evidently

(cons (car L) (cdr L)) = L

is an identity for any list L.

Two further functions are predicates, i.e. they take only the values t (for "true") or f (for "false") :

(atom x)evaluates x, and if it is an atom, the result is t ; if it is a list, the answer is f.

*Atom names will occasionally be underlined for readability, or written in capitals. Since the IBM 1620, like most machines, accepts only capitals letters, no semantic difference will be implied by these variations.

(eq x y) evaluates both x and y ; if they both refer to the same atom, the result is t ; if they refer to different atoms, or if one of them is a list, the answer is f ; if both are lists, the function is undefined.

From these basic functions it is possible to build up more complex ones, since all of them evaluate their arguments : if an argument is in turn a function - written, as has been seen, as a list which may be handled by LISP - it is, of course, evaluated by carrying out the function in question. However, more symbols are needed in order to express the logical relations of the conditions for a recursion : this is the main (though not the only) purpose of the symbols cond and if .

Of the three arguments of (if p e_1 e_2), the first is a predicate ; if its value is t, then the expression e_1 is evaluated and its value is that of the if-expression ; otherwise the expression e_2 is evaluated and taken as the result. In either case the other expression is not evaluated and may contain elements which under these conditions are undefined, e.g. the cdr of an empty list.

(cond (p_1 e_1) (p_2 e_2) ... (p_n e_n)) is used for more complicated cases ; it has any number of arguments, each of which is a pair of expressions, the first a predicate and the second any kind of expression ; the predicates are evaluated in turn until a true one is found, when the value of the cond becomes that of the associated expression e_i. No succeeding predicate or other expression is evaluated. If no predicate is true, cond is undefined ; it is therefore often convenient to have a last predicate which is always true. Such a predicate is written (and) in LISPITO, as will be seen below.

Certain other functions are frequently useful and are therefore provided as machine functions :

(quote x) = x. This function prevents the evaluation of the expression x, which may be anything. This is used when it is necessary to provide a constant, i.e. a quantity whose value is its name. The distinction between a quantity and its name, though not often made clearly in ordinary discourse, is fundamental in symbolic logic, where the device of quoting originates : thus the name of x is "x".

(null L) is a predicate which is true if L is an empty list, false if the list L contains elements. Note that (( )) is not null, since it contains the element ( ).

(list a b c ...) evaluates a, b, c, ..., and then forms a list of the results, in the order of the arguments. The number of arguments may be any, including none, in which case the result is ( ).

Finally, composites of car and cdr are frequently needed, often to considerable depth ; in view of the size of the IBM 1620, it has been possible to include only four of these composites :

```
(caar L) = (car (car L))
(cadr L) = (car (cdr L))
(cdar L) = (cdr (car L))
(cddr L) = (cdr (cdr L))
```

These are, of course, only defined if their arguments fulfil the appropiate conditions ; thus for cdar, the argument L must be a list whose first element is itself a non-empty list, etc.

EXAMPLES :

If the list L = ((a) b c), then

```
(car L) = (a)                        (cdr L) = (b c)
(caar L) = a                         (cadr L) = b
(cdar L) = ( )                       (cddr L) = (c)
(cons (caar L) (cddr L)) = (a c)
(cons (car L) (cdar L))= ((a))
(cons (car L) (cddr (cdr L))) = ((a))
(list (car L) (cddr (cdr L))) = ((a) ( ))
(atom (cdr L)) = f                   (atom (cadr L)) = t
(eq (caar L) (quote b)) = f
(eq (caar L) (quote a)) = T
(eq (caar L) (quote (a))) = f
(null (car L)) = F                   (null (cdar L)) = t
(if (null (cdar L)) (quote (empty list)) L) = (empty list)
(cond ((null (cddr L))   (caar L))
      ((atom (car L))    (cdr L))
      ((eq (quote a) (caar L))   L)
      ((and) (quote error)) )          = ((a) b c)
```

III. Logical Primitives in LISPITO

Logical functions differ from the predicates previously described in that not only their values but also their arguments take only the two possible values "true" or "false". Theyr are also called Boolean functions. In LISP, one of their principal uses is to combine predicates of various kinds into more powerful ones for use with if or cond.

The three logical primitives included in LISPITO are and, or and not. Of these, the last presents no difficulty. It has one argument, a predicate, whose value is found and converted into the opposite.

(and p_1 p_2 ... p_n) has any number of arguments, which it proceeds to evaluate in the order given, until it either encounters a predicate whose value is f or comes to the end of the arguments. In the first case the value of the function is f, and no further argument is evaluated ; in the second case, when no false argument has been found, the value of and is t. This last is always the case when and has no arguments at all : because of this, (and) is used to generate the value t for use in, say, cond, and it is not therefore necessary to reserve the symbols t and f for special use as logical values ; they may be freely used as function or variable names. In the output, however, the truthvalues appear as T and F.

Similarly, (or p_1 p_2 ... p_n) has any number of arguments and is true if an argument is found which evaluates to t ; if all arguments evaluate to f, or if there are none, the value of or is f. After a first true argument is found, no further ones are evaluated. (or) is used to generate the truthvalue f.

It is occasionally useful to be able to handle functions whose values may be either truthvalues or anything else. For this purpose, the logical primitives in LISPITO function as "semi-predicates", in the following sense : and takes the value of the first argument which differs from t ; or takes the value of the first argument which differs from f ; and not takes the value f if its argument is t and vice-versa, but takes the value of its argument if this is anything but t or f. If this feature is used in cond or if, it should be noted that these functions discriminate only between <u>t</u> - true for the purpose of the description given above -, on the one hand, and anything else (including <u>f</u>), taken as false, on the other.

EXAMPLES :

Let L = ((a) b c), M = (a b) ; then

```
     (and (not (null L))
          (eq (caar L) (car M)))       = t
     (and (or (null L)
              (null M)
              (not (atom (car L)))
              (atom (car M)))
          (eq (cadr L) (cadr M))
          (null (cddr M)) )            = t
     (and (atom (car L))
          (atom (caar M)))             = f, even though in this case
```

the second argument of <u>and</u> could not be carried out.

IV. Arithmetic Functions in LISPITO

It is, in principle, possible to define all arithmetical operations by means of the basic LISP functions, for instance by using Peano's postulates on a list of symbols which function as the names of the successive integers (and would, of course, be the conventional numerical symbols). However, this is extremely slow and uses up much memory, so that because of its frequent usefulness integer arithmetic has been incorporated in LISPITO. Larger processors often allow floating point numbers as well, but the size limitations of the IBM 1620 have made this impossible.

For LISPITO, a number is defined as an integer of up to three digits, with an optional sign in front. The processor will treat a number as an atom whose value is its name. No other value may therefore be assigned to it, which means that a number cannot be the name of a function or of a variable ; it may, of course, be the value of a function or a variable. As a result, when a number occurs as a constant within a function definition, it is not necessary to quote it. This conforms to the usual practise in algebra. An integer of more than three digits is treated as an ordinary atom and no fixed value is therefore associated with it.

To handle numbers, five functions are available. Except for num, their arguments must be numbers ; the result of the arithmetic operations will also be a number which must be less than 1000 in magnitude, or else an error stop is caused. The functions are :

(add n m), which calculates the algebraic sum $n + m$

(sub n m), which calculates the algebraic difference $n - m$

(mult n m), which gives the product nm

(sl n m), a predicate which is true if $n < m$; the abbreviation is "strictly less"; and

(num x), a predicate which is true if x has a number as its value and false otherwise.

Numbers may occur, like other elements, within list structures, and the five functions just mentioned may be used in recursive function definitions.

For all other LISPITO functions, numbers behave like atoms. In particular the predicate <u>atom</u> of a number is true, and <u>eq</u> may be used to determine the equality of two numbers. +0 and -0 are equivalent, i.e. (eq +0 -0) = t.

EXAMPLES :

```
(add +5 17) = 23                 (add 5 -17) = -12
(sub 11 +9) = 2                  (sub -5 -7) = 2
(mult -9 3) = 27                 (mult 20 50) causes an error stop
(num -3) = t
(num -003) = t
(num -0003) = f
(num xyz) = f, if xyz = a, but = t if xyz = 36
(sl 3 6) = t
(sl -6 -3) = t
(sl 6 3) = f
(sl 77 77) = f
```

V. Defining and Using a Function

By means of the primitives introduced in the previous sections, it is possible to give a function description, in the sense that for instance $ax^2 + bx^5 + c$ is one ; however, it is not possible as yet to associate a name with such a description, nor to indicate precisely what are the variables and therefore to give them values. Hence such a description cannot be recursive. In the example given, x is conventionally considered the variable ; but no value can be found for the expression until a, b, c also have numerical values assigned them : they may therefore also be considered variables, at least in some connections. A precise and generally applicable indication of which are to be considered the variables in a particular case of function description (i.e. what is known as a form) is achieved in LISP by means of the λ-symbolism introduced by Church[6] :

$$(\text{lambda } (v_1\ v_2\ \dots\ v_n)\ \text{form})$$

Here "form" is the description of the operations to be carried out, and v_1, v_2, ..., v_n are the variables to which values must be assigned. This may be done directly following the λ-expression, by enclosing it together with the values of the variables in parenthesis. Thus the λ-expression becomes a function, and the values of the variables its arguments, according to the rules of LISP notation :

$$((\text{lambda } (v_1\ v_2\ \dots\ v_n)\ \text{form})\ a_1\ a_2\ \dots\ a_n)$$

Of course the arguments may, in turn, be expressions of an appropriate kind, which will be evaluated and assigned as values to the variables during the evaluation of the form.

The variables v_1, v_2, ..., v_n need not necessarily all occur within the form, and there may also be other variables in it. The variables in the list following the λ are said to be bound by it : that is to say, the values they have throughout the pair of parenthesis beginning just before the λ are fixed by the pairing with the arguments this λ carries out. Any other variables occurring within the form are free variables ;

before evaluating the form, their values must, of course, have been fixed ; this will have occurred at a higher level lambda, which bound them.

This mechanism does not yet provide for giving a functional description a name to be used within its own definition, recursively. For this purpose LISPITO has a more general function, which defines functions non-recursively by associating their descriptions as values with their names :

```
(define
     (f₁ (lambda (v₁) form₁))
     (f₂ (lambda (v₁ v₂) form₂))
     (f₃ (lambda (v₃ v₂) form₃))
             ...                  )
```

Thus define has any number of arguments, each of which is a two-element list in which the first is a function name - which must be an atom, of course, - and the second the definition of the function. Within the corresponding forms any function names may now occur, provided that at the time of execution they have already been defined. Moreover, the names of the variables, being only dummy variables to be substituted for at execution time, need not all be distinct in different λ-expressions, though they must, of course, be so within one and the same λ. However, care must be taken that free variables to be used at some level do not coincide with variable names used at an intermediate level, since the λ at this level will then bind them to new values, which will be the ones used at the lower level rather than the intended ones, which have now been stored away on the push-down list.

To illustrate this, consider a function which determines whether a certain atom occurs in a list :

```
(define
     (member (lambda (x L) (cond ((null L) (or))
                                 ((eq (car L) x) (and))
                                 ((and) (member x (cdr L))) )))
     )
```

Observing that the recursion condition binds the variable x to the same value as it previously had, one can eliminate this variable by using an auxiliary function which is called by member but has only one variable. The definition then runs :

```
(define
     (member (lambda (x L) (member+ L)))
     (member+ (lambda (L) (cond ((null L) (or))
                                 ((eq (car L) x) (and))
                                 ((and) (member+ (cdr L))) )))
          )
```

Another way of solving the same problem is by using two interlaced functions :

```
(define
     (member (lambda (x L) (if (null L) (or) (member++))) )
     (member++ (lambda ( ) (if (eq (car L) x) (and)
                                (member x (cdr L))) ))
          )
```

Here both variables in <u>member++</u> are free.

In many cases it is convenient to be able to use a variable number of arguments with a function. Since the pairing-off mechanism will not work here, the solution is to place the name of a single variable after the but without enclosing it in parenthesis. The arguments, whatever their number, are then evaluated and formed into a list which will be the value of the variable at execution time. Thus a function to determine simply the number of arguments given it might be written

```
(number (lambda m (cond ((null m) (quote none))
                        ((null (cdr m)) (quote one))
                        ((null (cddr m)) (quote two))
                        ((and) (quote more))) ))
```

and given to <u>define</u> as an argument.

It is occasionally necessary to suppress the evaluation of the arguments, i.e. to let the names of the arguments be their values. This is achieved by using lambda+ in place of lambda, in either of the two types of uses mentioned above. This is equivalent to quoting all the arguments. (lambda+ ( ) e) is of course the same function as (lambda ( ) e).

It should be noticed that in LISP, contrary to algebraic practice, a function - either the name or the lambda expression - and its arguments are enclosed in parentheses to form a list. Thus (f x) rather than f(x) is used.

The function <u>define</u> permits also the redefinition of functions ; for instance, if it is wished to use + and - for the arithmetic functions, one can add

```
                    (+ add)
                    (- sub)
```

to the functions defined. In the same manner the values of variables ~~can~~ be set initially ; if a variable of the same name occurs later, this later value will take precedence, while the initially set value is placed on the push-down list and restored from it in due course.

A function defined as described may now be applied to values of its arguments by means of two functions belonging to the same class of generality as <u>define</u> ; the statements take the form

(<u>apply</u> f a_1 a_2 ... a_n)

(<u>eval</u> f a_1 a_2 ... a_n)

The difference between them is that <u>apply</u> takes the names a_1, a_2, ..., a_n as its arguments, i.e. it quotes them, and then gives them to the function f to carry out, while <u>eval</u> does not quote them but evaluates them. Hence

(eval f (quote a_1) (quote a_2) ... (quote a_n))

is equivalent to the apply statement above. Of course any functions named in <u>eval</u> or <u>apply</u> and any functions used directly or indirectly by them must have occurred previously in <u>define</u> statements.

EXAMPLES :

```
(define
     (equal (lambda (x y) (cond ((atom x) (eq x y))
                                ((atom y) (or))
                                ((null x) (null y))
                                ((null y) (or))
                                ((equal (car x) (car y))
                                        (equal (cdr x) (cdr y)))
                                ((and) (or))) ))
                  )
(apply equal (a b) (a b))
```

```
(apply equal (a (b c)) ((a b) c))

(apply equal (a (b c)) (a (b ( ))) )
```

This function determines whether two list structures are the same ; the three cases given it here to work on will produce, respectively, the answers t, f, f.

```
(define
     (reverse (lambda (L) (rev+ L (list))) )
     (rev+ (lambda (L m) (if (null L) m
                              (rev+ (cdr L) (cons (car L) m))) ))


(apply reverse (a b c))
```

answer will be (c b a).

VI. Debugging in LISPITO

By far the most frequent error in a LISP programme is an unbalanced parenthesis. The first step in debugging a new set of functions should therefore be a careful check of all parentheses. The input routine will detect an unpaired parenthesis, but will not, of course, detect an improperly placed one.

As a second step, particularly if some of the functions are complex in logical structure, it is recommended to try each function out individually. First those which require no auxiliary functions in their definitions should be tested, and if they involve conds or ifs, all possible paths through the functions should receive a try-out. Building up from tested functions step by step, one finally arrives at the tests for the complete set. Of course, some functions may have to be tested jointly.

A function involving free variables may be tested separately by setting these to appropriate values with a define statement, as described in the previous section. Such a statement will override preceding define statements ; it is therefore possible to alternate a series of define statements, setting free variables, with apply statements testing the function under varying conditions.

In a complicated recursion it is often helpful to be informed of intermediate values. This may be achieved with a special primitive function :

(print x) has its value the value of x ; but before proceeding with the evaluation of the function in which this primitive occurs, the value of x is printed by the output routine. (print x) thus has the same effect as x. A useful trick in debugging is to place variables or functions whose value might be of interest at intermediate points on separate cards, placed between one card in which "(print" is punched and another with only ")". These two cards can then be removed at will.

Care must be taken in the use of print, since too free a use of it will give rise to very large quantities of output whose utility is doubtful. Usually not more than one print per function should be used.

The last - and least desirable - debugging aids are the error stops. Of these, the following three may arise due to an incorrectly structured function :

UA@ signals an undefined atom. It is due either to a missing function definition (a misplaced parenthesis can cause an atom to look like a function to the processor) or to a variable which has not been assigned a value.

PL@ indicates that the push-down list is full. This is usually caused by a recursion which is infinite. However, it is possible to write a perfectly valid function with an excessive recursion depth ; such a function should be rewritten so as to use less push-down depth.

NV@ means "no vacuum" : the garbage collector has been unable to find sufficient free storage to allow the programme to carry on. Again this is usually a sign of an infinite recursion whose terminal condition cannot be reached, but may be due to a correct function which requires too many elements off the vacuum. The remedy is to rewrite the function.

A fourth error stop, usually caused by incorrect data, may also be due to a wrongly written function. This is the overflow stop in the arithmetic functions, signalled by OV@.

A frequent error is attempting to take the <u>car</u> of an atom or the cdr of an empty list. These will cause the machine to halt with the check-stop light on.

Other error stops are discussed in the following section.

For debugging purposes it is in general useful to punch the functions with few symbols per card, indented so as to make a readable listing. Once they work, they may be condensed into a minimum of card space by placing them within a single <u>define</u> statement and giving this as argument to a function

```
(condense (lambda(x) x))
```

which will eliminate all superfluous blank columns and moreover number the cards in sequence.

VII. Implementation in the 1620

LISPITO is an interpreter of the LISP language, i.e. the translation into machine language is carried out just prior to execution. A compiler would first produce a complete translation and then proceed to execution ; this has considerable advantage in that a set of functions, once compiled, runs much faster, but on the other hand the complete generality of LISP can only be achieved by an interpreter.

The 1620 is a variable word-length machine ; however, for the purpose of creating list structures it is necessary to introduce a standard cell size, which is that of the list element. Since addresses in the 1620 are five digits long, a ten-digit cell size was chosen to hold two addresses All pointers to list elements therefore have the same units digit, which was chosen to be 9. The left-hand pointer in each cell gives the next element on the list (i.e. the cdr), the right-hand one the contents of the list element (i.e. the car) ; this arrangement is the mirror image of that described in section I. In the last cell of a list the units digit of the left-hand pointer is 0. The region from 20000 to 39999 is available for the construction of lists, i.e. forms the vacuum.

An atom in LISPITO is formed of two parts : its value - a pointer to the function definition if it is the name of a function, or the argument value if it is a variable - and its print name, i.e. the string of characters which is read or printed out. There are 100 five-digit cells for the atom values, from 07901 to 08400 ; the print names are stored in variable-length fields, from 08401 to 09999.

The first part of the print-name table, up to 08887, contains the names of the machine functions. It should be noted that no value cells correspond to these names, so that, like numbers, <u>they cannot be used as variable names nor can new function definitions be associated with them</u>. They may, however, be renamed. Thus (car (lambda (L) e)), where e is an expression, will not work, though (car+ (lambda (L) e)) and (newcar car) are both valid definitions.

The last region which is needed in the system is the push-down list.

LISPITO uses a single push-down list on which both operations still to be carried out and previous values of arguments are placed ; for the latter it therefore functions as a restoration list, from which the previous values are restored to the value cells of the variable names whenever the execution of a λ-defined function is completed.

The push-down list occupies the region 10000 to 19999 and can hold 1000 elements. Processing a function is carried out as a double interpretation. On going down the push-down list the function names encountered are analysed sufficiently far to determine how their arguments are to be evaluated, and at the same time a digit indicating the kind of processing to be done on returning is placed into the units position. This process is carried forward until either an atom value is brought onto the push-down list from a value cell - i.e. the current value of a variable - or a quote is executed, in which case the argument is used the value. The second interpretation then begins : the units digit in the preceding element indicates to the processor the operation to be carried on the value, and the result overwrites the function ; the processor then goes back one element and repeats this. The two phases may alternate any number of times, of course, for instance in the successive evaluation of the arguments of a function which has more than one, also in carrying out and, or, or cond.

The input routine works on a different principle. It must establish the list structure and also build up the atom table. For the second purpose a linear search of the atom table is made, which is a little slow but saves programme space. If the atom has not yet been entered, an entry is made at the end of the table. Whenever a left parenthesis is encountered, a new sublist must be opened, but the place at which the list on the previous level continues must be kept available ; this may recur to any depth. LISPITO carries along this return address at the end of each list, in the last created element. This structure is essentially that of a "threaded list" (Perlis[7]). Once the input is complete, these return address are simply ignored by the processor, except for their units digit which is made 0 at the end of a list.

The output routine uses the push-down list. Whenevr the car of a list element points to a list, the cdr is placed on the push-down list and a left parenthesis placed in the output region ; otherwise the print name of the atom is found by a double linear search of the atom table and the next list element found. At the end of a list a right parenthesis is written and the preceding element on the push-down list picked up for continuation. Only one output region is used, since in the 1620 input-output is not simultaneous and so nothing is gained from buffering.

The garbage collector is composed of two passes through the vacuum. The first follows down through the list structure which hangs from the atoms with a value and from the elements waiting in the push-down list. The mechanism is similar to that employed by the output routine and uses the push-down list. A flag is set on all list elements which are reached. The second pass then goes linearly through the vacuum, removes the flags from the marked elements and combines the unmarked ones into a single linear list which constitutes the new vacuum. A garbage collection takes approximately six seconds.

Errors in the input will cause the machine to stop when they are discovered. Certain of these will also print out an error message :

AT@	- the atom table is full : if names are long, there may be few
EA@	- excessively long atom : no more than 30 characters are allowed
EP@	- error in the parenthesis count
NV@	- the garbage collector can find no more vacuum
OV@	- overflow in the arithmetic routines
PL@	- the push-down list is exhausted
UA@	- undefined atom : a function has no definition or an atom no value.

The machine will halt with the check-stop light on if an illicit argument has been given to a primitive function, such as an atom to car or an empty list to cdr.

The remedy in the first three cases is obvious ; the others are discussed in the preceding section.

VIII. Operating Instructions

Writing the programme. A LISPITO programme consists of three kinds of statements : define statements which define functions or set variables, evaluation statements beginning with apply or eval, and commentary statements of the form (comment ...). The first two types are discussed in section V ; the last is simply reproduced in the output without any processing.

The three kinds of statements may occur in any number and order, with the sole limitation that any function name called, directly or indirectly, by an evaluation statement must have been defined already. All machine functions are, of course, considered as defined.

LISPITO uses input lines of 72 characters, either in cols. 1 to 72 of cards or in a typewriter line. There is no particular format to be observed, since the parentheses will indicate everything that is needed to the processor. Hence a programme may be punched with as many or as few symbols per card as desired ; indenting successive lines in accord with the logical structure - as has been done in many of the examples given in earlier sections - helps towards understanding a programme ; blank cards may be interspersed to make a readable listing.

The choice of atom names is restricted by the following considerations :

1. All characters except the record mark, the two parenthesis, and the blank space may be used. The first may not be used at all, the other three serve to separate atoms and indicate the list structure.

2. An atom name may not exceed 30 characters in length.

3. The names of the machine functions must not be used as variable or defined function names.

4. A symbol of three or less digits beginning with a numeral (or of four or less digits beginning with an algebraic sign and a numeral) will be treated as a number (see section IV). Thus +71, 419, -6AA are all numbers (the last will be interpreted as -611), while .71, 5, 1234, -1234 are non-numerical atoms.

Loading the processor. To load the processor, set the console switches as follows : parity STOP, input/output STOP, overflow PROGRAM, programme switches as desired for the input and output (see below). Then place the card deck into the reader hopper, press RESET on the console and the LOAD button on the card reader. The processor does all necessary memory clearing. If the input is from cards, these should follow the processor deck immediately, and they will be read as soon as the processor is loaded ; there is no intermediate machine halt. Otherwise operation follows the IBM manual.

Input. The input is controlled by console switch # 1 : turn it off for card input, on for input from the typewriter. Anything punched or written beyond column 72 will be ignored, and column 1 of the next card or line is considered to follow immediately upon column 72. If input is from the typewriter, the MAR display will show the address into which the last character has gone ; more than 00999 in MAR means that column 72 has been passed. If input from the typewriter goes to 01015 or beyond, the processor must be reloaded. Any other error made in typing may be corrected by turning console switch # 3 on, pressing RELEASE and START (or the RS key on the typewriter), returning the carriage and turning switch # 3 off. This will cancel the line just written. For card input console switch # 3 should be off.

Console switch # 1 may be turned on or off at any time and will change the input mode starting with the next line or card.

Input will be immediately converted into list structure, but processing will not start until a special symbol FINENT), is encountered, which must always end the input ; the right parenthesis following it is unpaired. At this point the processor checks whether all other parentheses are properly paired, emits an error message if not, and otherwise starts processing, first setting up the vacuum. FINENT) is usefully placed on a separate card.

Output. The output routine is entered automatically by the processor, but the output medium is under control of programme switch # 2 : if it is off, the output is punched on cards, if it is on, it is written on the typewriter. In both cases only 72 columns are occupied ; the card output is successively numbered in the last five columns, as a help in keeping the cards in order. The format is the same as for input, except that a single blank space is used between atoms or between a right and following left parenthesis. For this reason the function condense, described in section VI, will reduce the number of cards to a mimimum. Atoms are not divided between successive cards.

Garbage collector. The processor enters and leaves the garbage collector automatically as required. It is, however, sometimes useful to know how many garbage collections have taken place. If this is desired, turn console switch # 4 on : for each garbage collection the processor will then write the single symbol ● on the typewriter. Like the error messages, this is independent of the setting of switch 2.

References

[1] J. McCarthy, "Recursive Functions of Symbolic Expressions and Their Computation by Machine", Comm. ACM, 3 (1960), 184.

[2] Martin Davis, Computability and Unsolvability, McGraw-Hill, New York 1958.

[3] Rózsa Péter, Rekursive Funktionen, 2. Aufl., Akadémiai Kiadó, Budapest 1957.

[4] LISP 1.5, Programmer's Manual, Massachusetts Institute of Technology Computation Center and Research Laboratory of Electronics, Cambridge, Mass., 1962.

[5] See e.g. Information Processing Language-V Manual, ed. Allen Newell, Prentice-Hall, Englewood Cliffs, N.J. 1961, and references given there.

[6] Alonzo Church, The Calculi of Lambda-Conversion, Annals of Mathematics Studies No. 6, Princeton University Press, Princeton, N.J., second printing 1951.

[7] A.J.Perlis and C. Thornton, "Symbol Manipulation by Threaded Lists", Comm. ACM, 3 (1960), 195.