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

LIST TECHNIQUES

PROGRAM NOTE #13

20 August 1963

ABSTRACT

For its own purposes, LISP employs a certain kind of list structure. Experience has shown that many other kinds of list configurations exist and may profitably be used in the appropriate circumstances. With the operator predicates which exist in the MBLISP processor, the LISP language may be used to govern the formation and utilization of alternative list structures, such as Threaded Lists. Several such schemes are described, together with their associated control functions.

LIST TECHNIQUES

Although many computer programmers were intuitively familiar with list techniques from the very earliest days of electronic computers, it appears that the subject first emerged as an organized discipline from the work of Newell, Simon and Shaw in connection with heuristic programs which would simulate human mental processes, insofar as they were known, in attacking the solutions of problems. Such programs would constantly generate unpredictable quantities of intermediate results, which needed to be cross-referenced, but not according to any particularly mathematically regular pattern. Such haphazard generation of data precluded fixed storage being allocated to retain it, while not only the complexity of the cross-references, but their continual revision and rearrangement would have required a continuing movement of the data, even were it possible to have reserved adequate spaces for it.

With the recognition that large quantities of haphazard data could be generated, and that list techniques---wherein certain cells were set aside specifically for the purpose of indicating the interrelations among themselves and the data to which they referred---provided an adequate means of handling this type of data arrangement, a number of specialized languages were developed to handle this type of program. They included the IPL series, particularly IPL-V, which was substantially a battery of macro instructions oriented toward list handling; FLPL (FORTRAN LIST PROCESSOR), LISP, and Threaded Lists.

The last mentioned languages concentrated on exploiting in each case just one particular list arrangement, as a means for accomplishing the most general program definitions and calculations. Such concentration has led to particularly simple and elegant programming languages in each case. Although the power of such languages is extremely impressive, particularly when considered in relation to their foundations, they also possess characteristic drawbacks. One is inevitably drawn to the conclusion that any such language must reserve for itself the ability to work directly with memory stores, as we now know them, in spite of its own predilections for list structure arrangement. It is a tribute to a language such as LISP that it may control the memory manipulation with little or no disturbance to its own operational procedures.

Before outlining the actual physical arrangement of the memory store, we shall describe a series of routine operations with lists which are of a fundamental nature, can be accomplished entirely within the LISP language, and which recur in almost every application of LISP. Generally, they are involved with searching a list, deleting or inserting information, or making simple modifications or alterations to their arrangements. In studying such functions it is helpful to think of a list simply as an ordered set.

(ELEM X L) is a predicate which determines whether the element X, assumed to be an atom, is a member of the list L.

```
(ELEM (LAMBDA (X L) (AND (NOT (NULL L))
(OR (EQ X (CAR L)) (ELEM X (CDR L)) ) )))
```

(SUCC X L) yields the element following X on the list L.

```
(SUCC (LAMBDA (X L) (IF (EQ X (CAR L))
(CADR L) (SUCC X (CDR L)) )))
```

This definition assumes that it is known that the element X actually belongs to the list L, for there is no precautionary test for (NULL L). Likewise it is assumed that X actually possesses a successor and that it is not the last element of the list. To forestall such a possibility it would also be necessary to add a test for (NULL (CDR L)).

(SUCC* X L) yields the element preceding X on the list L.

```
(SUCC* (LAMBDA (X L) (IF (EQ X (CADR L))
(CAR L) (SUCC* X (CDR L)) )))
```

Again, it is assumed that the list is neither empty nor that X is the first element.

(ASSOC X L) searches alternate elements of the list L for the presumed atom, X. If found the value of ASSOC is the succeeding element; otherwise the value is X. Such a list, L = (N1 D1 N2 D2 N3 D3 ...) is useful for storing the equivalents DI of the names NI; names alternate with definitions, and every other element is searched.

```
(ASSOC (LAMBDA (X L) (COND
((NULL L) X)
((EQ (CAR L) X) (CADR L))
((AND) (ASSOC X (CDDR L))) )))
```

In this definition it is assumed that the context of the search is known, so that no explicit check needs to be made that the list contains an even number of elements.

(ASSOC* X L) is used to invert the action of ASSOC; namely an alternating search of L is made starting with the <u>second</u> element; if X is found its predecessor is taken. Again it is assumed that the list is of even length.

```
(ASSOC* (LAMBDA (X L) (COND
((NULL L) X)
((EQ (CADR L) X) (CAR L))
((AND) (ASSOC* X (CDDR L))) )))
```

(EXPUNGE X L). All instances of the atom for which X stands are removed from the list L.

```
(EXPUNGE (LAMBDA (X L) (COND
((NULL L) L)
((EQ (CAR L) X) (EXPUNGE X (CDR L)))
((AND) (CONS (CAR L) (EXPUNGE X (CDR L)))) )))
```

(REMOVE X L). The first instance of the atom X is deleted from the list L.

```
(REMOVE (LAMBDA (X L) (COND
((NULL L) L)
((EQ (CAR L) X) (CDR L))
((AND) (CONS (CAR L) (REMOVE X (CDR L)))) )))
```

(SUBST X Y L). Y is presumed to be an atom. Each instance of Y on the list L is replaced by X.

```
(SUBST (LAMBDA (X Y L) (COND
((NULL L) L)
((EQ (CAR L) Y) (CONS X (SUBST X Y (CDR L))))
((AND) (CONS (CAR L) (SUBST X Y (CDR L)))) )))
```

(REPLACE D L). If an atom appears on the alternating dictionary D, it is to be replaced by its equivalent on the list L.

```
(REPLACE (LAMBDA (D L) (IF (NULL L)
L (CONS (ASSOC (CAR L) D) (REPLACE D (CDR L))
) )))
```

(POSSESSING P L). An extract of the list L is made, consisting of those elements possessing the property P.

```
(POSSESSING (LAMBDA (P L) (COND
((NULL L) L)
((P (CAR L)) (CONS (CAR L) (POSSESSING
      P (CDR L))))
((AND) (POSSESSING P (CDR L))) )))
```

(REVERSE L) is a list of the elements appearing on the list L, but in the opposite order. If the elements are themselves lists, their order is not affected. It is defined by the help of an auxiliary function.

```
(REVERSE (LAMBDA (L) (REVERSE* L (LIST))))

(REVERSE* (LAMBDA (L M) (IF (NULL L) M
(REVERSE* (CDR L) (CONS (CAR L) M)) )))
```

In the functions which follow, let us agree that U and V will mean the lists

```
U = (U1 U2 ... UN)
V = (V1 V2 ... VN).
```

Then

(APPEND U V) is the list resulting by attaching the list V to the end of the list U. If U and V are defined as above, then (APPEND U V) = (U1 U2 ... UN V1 V2 ... VN)

```
(APPEND (LAMBDA (U V) (IF
(NULL U) V (CONS (CAR U) (APPEND
(CDR U) V)) )))
```

It is interesting to contrast the use of CONS and LIST with APPEND; continuing to use the same example we would have

```
(CONS U V) =  ((U1 U2 ... UN) V1 V2 ... VN)

(LIST U V) =  ((U1 U2 ... UN) (V1 V2 ... VN))
```

(MERGE U V). Elements are taken alternately from the lists U and V, presumed to be of the same length, in order to form an alternating list. Thus (MERGE U V) = (U1 V1 U2 V2 ... UN VN).

```
(MERGE (LAMBDA (Y V) (IF (NULL U) U
(CONS (CAR U) (CONS (CAR V)
(MERGE (CDR U) (CDR V)) ) ) )))
```

(UNMERGE L) has as its argument a list L of even length, and as its value a list of two lists. The first of these contains the odd elements of L while the second contains the even elements. It thus inverts the action of MERGE.

```
(UNMERGE (LAMBDA (L) (IF (NULL L) (LIST L L)
((LAMBDA (X) (LIST (CONS (CAR L) (CAR X))
(CONS (CADR L) (CADR X)))) (UNMERGE (CDDR L))
) )))
```

(PAIR U V). A list of pairs is formed, composed of an element of U and a matching element of V, for all the elements of the two lists, which are presumably of the same length. In terms of our example, (PAIR U V) = ((U1 V1) (U2 V2) ... (UN VN))

```
(PAIR (LAMBDA (U V) (IF (NULL U) U
(CONS (LIST (CAR U) (CAR V)) (PAIR
(CDR U) (CDR V)) ) )))
```

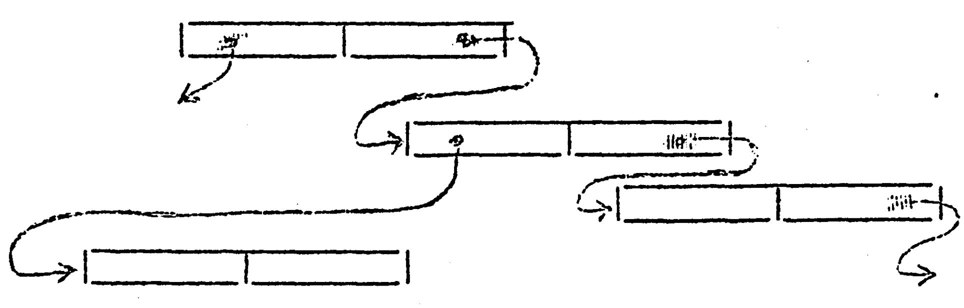
By continuing to enumerate further examples, one could prolong indefinitely the catalogue of possible operations with lists. However the functions cited show how readily one may manipulate lists with the aid of the LISP language. Logically, in fact, LISP is all which is logically necessary to perform every imaginable kind of operations with lists. From a practical point of view, however, the actions of LISP can then to be quite extravagant. To understand why this should be so, we have to consider the actual physical implementation of lists.

In the formal definition of LISP, a list is defined recursively as an entity which commences with a left parenthesis, terminates by a right parenthesis, and otherwise consists of a series of entities (separated by blanks) which are themselves either atomic symbols or lists. To give such a definition one has to have previously agreed that an atomic symbol is a string of characters devoid of parentheses or blanks. However, all such concepts as parentheses, characters, strings, blanks and so on have to have their representation in terms of some memory configuration in the memory store of the computer. In fact, in this realm a list seems something entirely different.

We recall that the memory store is composed of units called words, each of which contains a certain number of binary digits, or bits. The words of the memory are numbered serially (00000 to 77777 octal, in the IBM 709) in the sense that when one of these numbers is used as a part of an instruction and decoded by the proper organ of the central processor, the corresponding word can be retreived from the memory store.

In many machines of commercial design, a word is large enough to hold two of these serial numbers, or addresses. Even when it is not, it is generally possible to treat two consecutive words as a unit. We shall call the necessary combination of words which holds two addresses a BILE, or binary list element. It would in principle also be possible to work directly with MULEs, or multiple list elements. However, whenever one is dealing with a dynamic list structure, there is often such a great demand for new list elements, that eventually the memory store must be examined to see whether there are any abandoned words, no longer usable by the program, which may be returned to active use. The vacuum, or store of available words, must periodically be replenished. The difficulty arises that if one wishes to use large blocks of consecutive words, and if the size of the blocks vary, there will gradually be a degeneration of the vacuum, in that many small blocks will be available, but few large ones. To retain the maximum flexibility, it seems far preferable to construct MULEs from BILEs, even with the sacrifice of additional space in the memory to link the BILEs into MULEs.

In describing the memory store of a computer, it is convenient to introduce certain diagrammatic cinventions. In the figure below



we see the representation of a certain list configuration. The rectangles represent BILEs, which in the IBM 709 are just words. They are divided into two portions to indicate that they hold two addresses. In fact the left half corresponds to the decrement, bits 3-18, while the right half corresponds to the address portion of the word, bits 21-35. Actually there are 6 additional bits which are sometimes used as flags, which are not represented.

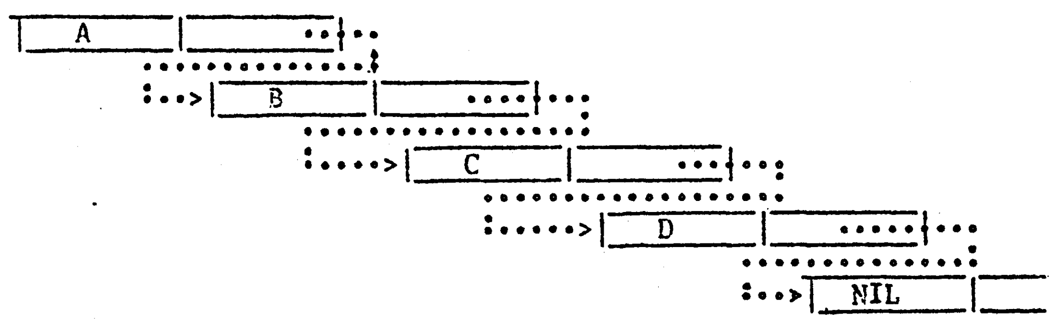
An arrow running from either side of one word and pointing to another, represents the fact that at the position indicated by the tail of the arrow is stored the address of the word lying at the tip of the arrow. These linkages serve to determine the list structure.

In terms of these diagrams, we can relate certain list structures to the "lists" upon which LISP operates. Since we do not wish to enquire how atomic symbols are represented, nor how a proper printed representation of a list is eventually produced, we shall agree that a BILE of the form:

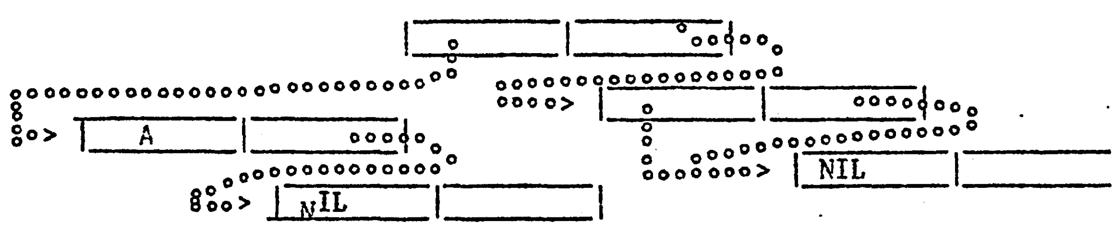
constitutes an adequate left hand linkage to the atom ZZZZZ.

In this sense, an empty list, (), has the representation

The special atom, NIL, is used to terminate a list. On the other hand we would represent the list (A B C D) by the diagram:



As another example, the list ((A) ()) would be diagrammed as:



There is a certain peculiarity in this drawing, in that the empty set which is the second element of the list is represented physically by the same NIL which terminates the entire list. Common subexpressions may be represented by identical list structure, although as the second NIL-bearing BILE shows, this is not necessarily universally the case. It is nevertheless one of the advantages which LISP possesses over say the threaded list type of structure, that common subexpressions may be so represented.

It is readily percieved that the LISP function CONS is readily adapted to this type of list representation. It requires only that a fresh BILE be extracted from the vacuum, the first argument be written as its left linkage, and the second argument as its right linkage.

It is actually a matter of taste whether the empty list be assigned a unique memory address, with its decrement pointing to the atom NIL. McCarthy's LISP so represents (), thereby slightly simplifying a number of operations. For instance, the test EQUAL does not have to include as a special case the test whether both arguments are empty sets. Moreover, since lists in many LISP programs tend to be fairly short, individual empty lists terminating each list consume a sizeable percentage of the active memory store.

As soon as lists are to be used by other processors than the LISP processor, the considerations change, and it may be necessary not to have an empty list uniquely represented. In particular, it is desirable to have the assurance that every value of the function (LIST) is distinct.

Once we have a model for the internal operation of a computer, we may begin to find fault with the LISP mode of operation. Recalling the definition of the function (APPEND U V): (LAMBDA (U V) (IF (NULL U) V (CONS (CAR U) (APPEND (CDR U) V))))), we see that an entire new copy of the list U is created, to which V is attached, simply for the sake of the fact that somewhere else in the program U may be required intact. For the recursive mode op operation this is an entirely justified and proper assumption. Nevertheless, we may find ourselves contemplating a list which we are sure that will be used nowhere else in its original form, and wondering whether the complete new copy of U is entirely necessary. So long as we are to use computers as presently constituted, this will remain a valid question. We moreover suspect that this constitution is bound to persist.

APPEND yields only one example, but the principle is equally valid whenever we are forced to reproduce the entire head of a list for the sake of making some change at some distance along the list.

It is only necessary to adjoin two operators to the LISP language as primitive "functions", to manipulate lists in the most general fashion. They are most conveniently introduced as operator predicates, so that their operation may be controlled by the LISP functions AND and OR. These operators are:

(SAR E X) which causes (CAR X) to become E, and whose value is T.

(SDR E X) which causes (CDR X) to become E, and whose value is T.

Together with the function of no variables (LIST) which will produce as its value a new cell, freshly detached from the vacuum, CAR of which is the (unprintable) atom NIL, these two operators allow us to generate a BILE, and set either of its two linkages to any values we desire. In addition, the linkage of any already existing BILE may be altered.

Although they are logically sufficient for all list manipulations, there are certain of their composites which are very convenient in certain circumstances. Also convenient are certain variants which take other values than T.

Among these are:

(XAR E X) whose value is the old (CAR X)
(XDR E X) whose value is the old (CDR X)

(QAR E X) whose value is X
(QDR E X) whose value is X

(RAR E X) whose value is E
(RDR E X) whose value is E.

In terms of these functions we can define CONS:

(CONS (LAMBDA (X Y) (QAR X (QDR Y (LIST))))).

Other functions are:

(DESTROY (LAMBDA (L) (SDR (CDDR L) (QAR (CADR L) L))))

DESTROY obliterates the first element of a list in such a fashion that any pointers to L automatically now point to (CDR L). Jowever, if there were any pointers to (CDR L), these still point to (CDR L) although these two instances of (CDR L) ane no longer represented by the same physical list structure.

(DESTROY* (LAMBDA (L) (SDR (CDDR L) (CDR L))))

DESTROY* obliterates the second element in the list L, without disturbing the remainder of the list in any fashion. It is an operator predicate.

(INSERT E L) is an operator which yields a new list containing E at the head of L. It differs from CONS in the respect that pointers to L now all point to the new list.

(INSERT (LAMBDA (E L) (SAR E (QDR (QAR (CAR L) (QDR (CDR L) (LIST))) L))))

(INSERT* E L) is an operator predicate which inserts E into the list L following (CAR L). In non-LISP terms, we may think of it as inserting the item E into a list following the designated cell. Unlike the operator INSERT, we assume that all the pointers to L wish to continue to point to the <u>same</u> item of information, rather than to the <u>first item</u> on the list, whatever it may be.

(INSERT* (LAMBDA (E L) (QDR (QAR E (QDR (CDR L) (LIST))) L)))

One can readily envision extensions of these operators, which make conditional insertions into a list at selected points. For instance, let us suppose that we wish to build up a list whose elements occur in increasing order---say according to the predicate SL (STRICTLY LESS). We do this by comparing the new element with each element of the list in turn until its proper place in the list is found. An operator accomplishing this result is FILE:

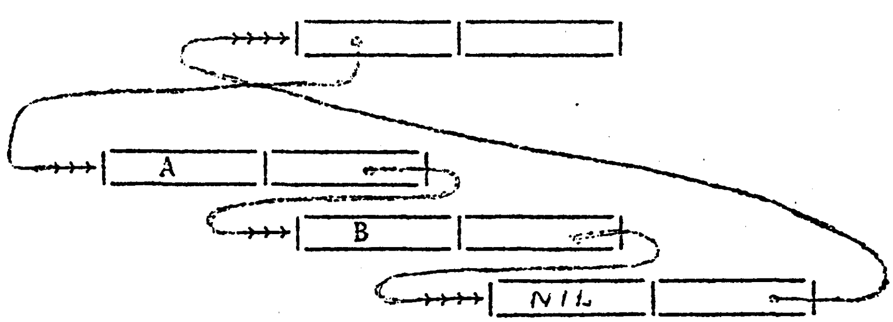
```
(FILE (LAMBDA (E L) (OR
      (AND (OR (NULL L) (SL E (CAR L))) (INSERT E L))
      (FILE E (CDR L)) )))
```

A closely related operator predicate, FILEONCE, will generate an ordered list without repetitions:

```
(FILEONCE (LAMBDA (E L) (OR
        (AND (OR (NULL L) (SL E (CAR L))) (INSERT E L))
        (EQ E (CAR L))
        (FILEONCE E (CDR L)) )))
```

Given a convenient assemblage of operators to be used in working with lists, the next topic to which one turns his attention is the establishment of certain list patterns which are of basic serviceability, and with the peculiarities of whose usage he wishes to become familiar. As we have seen, one of the most fundamental of these, and the one favored by LISP, is the binary tree. However, characteristically the usage of a binary tree requires an auxiliary push down list, if one is to remember the right half of the tree while he is working with the left half. The problem requiring this solution can be phrased in the following terms: One wishes to pass through a binary tree in such a fashion that after seeing each expression, he then sees all the subexpressions in sequence. We may think of each node in the tree as representing representing a subexpression, formed by all the nodes to which it is connected. The minimal elements, in the context of LISP, correspond to the atomic symbols. A pushdown list (which may actually be an array) has the property that new items are adjoined to its head, and moreover whenever an item is removed, it is removed from the head. Thus, the first item adjoined will be the last to be removed, while the last adjoined will be the first removed.

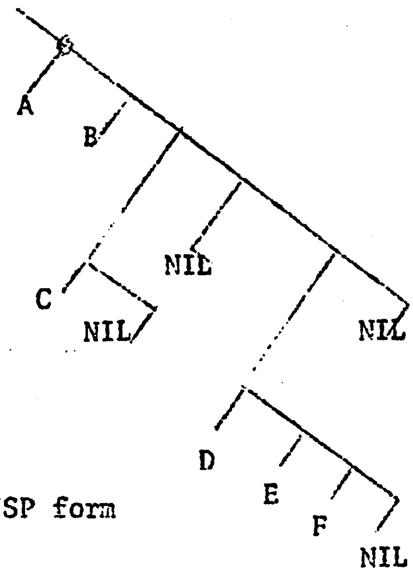
If we recall that for the purposes of LISP each tree terminates either with a proper atom, or else the unprintable atom NIL, and that furthermore only the NIL terminating a list corresponds to a point at which we would wish to conclude a subexpression and return to the main expression, we see that it is possible to incorporate the continuation address which we would have relegated to the pushdown list directly into the binary tree itself. This is the basic scheme of the Threaded List system of Perlis; each subexpression terminates with a connector to the head of the expression. In terms of rectangle-diagrams, the layout is the following:



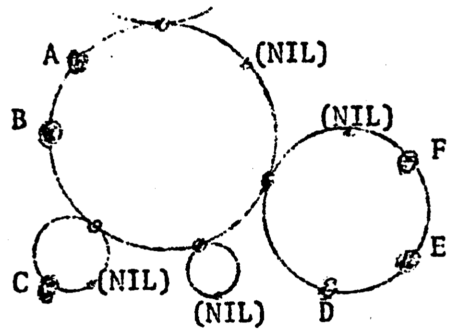
which represents the expression (A B). As we see, each subexpression is linked to the cell representing it in the expression of which it is a part. This is the form of the linkage rather than a mere connector to the beginning of the subexpression, because it allows us to return to the higher level, while the other arrangement would only allow us to circulate continually around the same sublevel.

We may readily perceive the feature which is one of the greatest drawbacks of threaded lists---it is impossible to allow common subexpressions to be represented by the same physical list structure, because the return linkage can point to only one cell. Weizenbaum's Knotted Lists represent a compromise, by hanging a pushdown list at the bottom of each subexpression.

If our model for LISP's lists is a binary tree, then the model for a threaded list is a family of tangent circles. To illustrate this proposition, the two diagrams below show the two representations of the expression (A B (C) () (D E F)):



LISP form



Threaded List form

Composites of CAR and CDR may be used to isolate selected items from a threaded list just as they are in LISP. However, they would probably be used in a slightly different manner, in that one would probably have sequenced variables designating locations in the threaded list, and the operator XEC would be used with CAR or CDR as its argument to move them. In fact, in threaded list theory, there are three basic sequences for list variables. Assuming that L is a pointer to a list, we have:

```
(SEQA (LAMBDA (L) (SEQA* (CDR L))))

(SEQA* (LAMBDA (L) (COND
        ((NULL L) (SEQA* (CDDR L)))
        ((ATOM (CAR L)) (CDR L))
        ((AND) (SEQA* (CAR L))) )))

(SEQW (LAMBDA (L) (SEQW* (CDR L))))

(SEQW* (LAMBDA (L) (IF (NULL L) (SEQW* (CDDR L)) L)))

(SEQL (LAMBDA (L) (CDR L)))
```

Of these, the function SEQA, or the atom sequence, yields all the atoms of an expression from left to right as they would appear in the written expression, disregarding parentheses. Actually, the value of SEQA is a list, CAR of which is the desired atom.

SEQW, on the other hand when applied repeatedly will yield every subexpression in the order written from left to right, but each subexpression will be followed by its own subexpressions in turn. Again we must take (CAR (SEQW L)).

SEQL, which also must be composed with CAR, is designed simply to run through the subexpressions of one level only.

For example, if

L = (((A B) C (D E F)) G (H I))

Then SEQA would (composed with CAR) yield the sequence

A B C D E F G H I

while SEQW would yield

(((A B) C (D E F)) G (H I)), ((A B) C (D E F)),
(A B), A, B, C, (D E F), D, E, F, G, (H I), H, I.

and SEQL would produce

((A B) C (D E F)), G, (H I).

In addition to questions of reading information from threaded lists, one also has to deal with the problem of constructing and modifying threaded lists. Although SAR and SDR theoretically suffice for this purpose, they cannot be used directly without further thought. Another consideration is the fact that a threaded list is a very carefully adjusted structure, and the intemperate insertion of linkages will destroy the thread. It is therefore desireable to use, insofar as possible, primitive operations in the construction of threaded lists which always leave a threaded list a threaded list after their operation. Three such functions seem to suffice:

```
(LISTHEAD (LAMBDA (L) (SDR (QDR L) (LIST)) L)))

(ISPTL (LAMBDA (E L) (SDR (QAR E (QDR (CDR L) (LIST))) L)))

(ISRTW (LAMBDA (E L) (SAR (QAR E (QDR (CAR L) (LIST))) L)))
```

The first of these, (LISTHEAD L) causes (CAR L) to become an empty list. This operator requires special treatment because of the particular structure of a threaded empty list, that the linkage following the NIL must return to the main expression. We assume that (LIST) has been so designed, that there will always appear a NIL as CAR of the new cell withdrawn from the vacuum.

The second, (ISPTL E L) causes the expression E to be inserted into the threaded list L in such a way that it will be the next expression delivered by the sequence mode, SEQL. That is to say in LISP, it will become (CADR L), automatically displacing the remainder of the list one place. (CAR L) will remain unchanged.

The third operator, ISRTW causes the expression E to be inserted into the threaded list L in such a way that it will be the next expression delivered by the sequence mode, SEQW. That is to say in LISP, it will become (CAAR L), automatically displacing the remainder of (CAR L) by one place. (CDR L) will remain unaffected. It is assumed for this purpose that (CAR L) is a list (possibly empty) and not an atom.