Given a flowchart with a single entrance and a single exit, it is easy to write down the recursive function that gives the transformation of the state vector from entrance to exit in terms of the corresponding functions for the computation blocks and the predicates of the branch points. In general, we proceed as follows.

In figure 6, let β be an n-way branch point, and let $f_1, \cdots, f_n$ be the computations leading to branch points $\beta_1, \beta_2, \cdots, \beta_n$. Let ϕ be the function that transforms ξ between β and the exit of the chart, and let $\phi_1, \cdots, \phi_n$ be the corresponding functions for $\beta_1, \cdots, \beta_n$. We then write

$$\phi[\xi] = [p_1[\xi] \rightarrow \phi_1[f_1[\xi]]; \cdots; p_n[\xi] \rightarrow \phi_n[f_n[\xi]]]$$

Acknowledgments

The inadequacy of the λ-notation for naming recursive functions was noticed by N. Rochester, and he discovered an alternative to the solution involving *label* which has been used here. The form of subroutine for *cons* which permits its composition with other functions was invented, in connection with another programming system, by C. Gerberick and H. L. Gelernter, of IBM Corporation. The LISP programming system was developed by a group including R. Brayton, D. Edwards, P. Fox, L. Hodes, D. Luckham, K. Maling, J. McCarthy, D. Park, S. Russell.

The group was supported by the M.I.T. Computation Center, and by the M.I.T. Research Laboratory of Electronics (which is supported in part by the U.S. Army

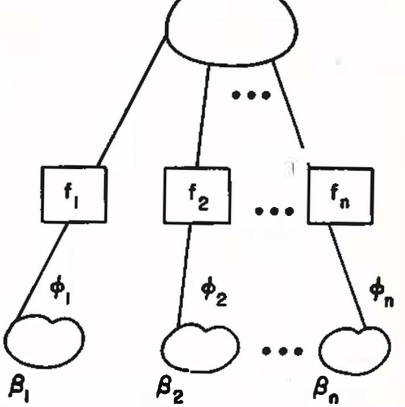


FIG. 6

REFERENCES

1. J. MCCARTHY, Programs with common sense, Paper presented at the Symposium on the Mechanization of Thought Processes, National Physical Laboratory, Teddington, England, Nov. 24–27, 1958. (Published in Proceedings of the Symposium by H. M. Stationery Office).
2. A. NEWELL AND J. C. SHAW, Programming the logic theory machine, Proc. Western Joint Computer Conference, Feb. 1957.
3. A. CHURCH, *The Calculi of Lambda-Conversion* (Princeton University Press, Princeton, N. J., 1941).
4. FORTRAN Programmer's Reference Manual, IBM Corporation, New York, Oct. 15, 1956.
5. A. J. PERLIS AND K. SAMELSON, International algebraic language, Preliminary Report, *Comm. Assoc. Comp. Mach.*, Dec. 1958.

Symbol Manipulation by Threaded Lists*

42533

A. J. PERLIS AND CHARLES THORNTON, *Carnegie Institute of Technology, Pittsburgh, Pa.*

Part I: The Threaded List Language

1. Introduction

In the field variously called artificial intelligence, heuristic programming, automata theory, etc., many of the most interesting problems do not lend themselves readily to solutions formulated in the automatic programming systems now in wide use. Several new approaches to more adequate and natural programming systems have been made in the past few years. Notable among these are the list structure languages of the IPL family developed by Newell-Simon-Shaw [1] and LISP by McCarthy [2]. They provide great flexibility for the construction of highly composed programs, and are able to represent and process systems of arbitrarily great complexity, subject

* The work was supported in part by the Office of Naval Research under contract number Nonr-760 (18), Nr 049 141 and by the U. S. Army Signal Corps under contract number Da 36-039 Sc 75081, File No. 0195-PH-58-91 (4461).

to machine limitations. Specifically, they possess the properties:

(i) Arbitrary nesting of subroutines, thus permitting definition of complex functions by programs composed of more "simple" functions.

(ii) Functions may be defined recursively.

(iii) A memory structure permitting the use of data whose space requirements may continuously vary during the execution of a program.

In addition, the LISP language permits its commands to be represented in a statement form.

This paper presents an addition to the list structure languages which is expected to add to the above advantages while simplifying machine processing of lists. This is done by the use of *threaded lists*.

A threaded list is a list structure in which the last element of each list specifies the location of the head of the list of which it is the terminal member. A formal definition is given in the following section. The advantage of this structure is that it permits the definition of various modes for sequencing through lists without requiring the use of the usual push-down lists for retaining sequencing information. This corresponds, in the representation of programs by list structures, to the representation of all computable functions iteratively, rather than recursively. That iteration is in fact adequate has been shown by R. M. Robinson [3] and Julia Robinson [4]. Explicit program-controlled sequencing permits—in many cases—simple coding and processing.

It is interesting to note that the iterative definitions can be made with little or no increase in complexity over more customary recursive definitions; in many cases they are actually simpler, as will be observed from the examples of threaded list programs in this paper.

The examples given of manipulation of threaded lists are from elementary algebra. The striking result is that algebraic operations which are distant from machine primitives can be built up quite easily using very few primitives. Aside from the computing simplicity which results from iterative operation, it is noted that the human programming task is lightened, since the processes which can be defined are similar to those often used by humans in the manipulations of symbols.

2. Definition of a Threaded List

Let I denote the set of positive integers, and let X denote the set of symbols which may occur as data. A threaded list is a set, S, of n quartets, $t_i = (A_i, f_i, L_i, R_i)$, $i = 1, 2, \cdots, n$, where the domains of A_i, f_i, L_i, and R_i are I, (0, 1, 2), I ∪ X, and I, respectively; and where the following algorithm establishes a 1-to-1 correspondence between the members of S and the integers $1, 2, \cdots, n$.

1. Let S′ be the subset of S whose members are defined by:

$$t_i \in S' \Leftrightarrow \exists t_j \in S \ni (R_j = A_i) \wedge (f_j = 2) \wedge (f_i = 1)$$

If S′ is empty, then S is not a threaded list.

2. Let some $t_i \in S'$, say t_{i_1}, correspond to 1. The correspondence of the remaining members of S is specified inductively. Suppose t_j corresponds to m, $1 \leqq m \leqq n$. Then

$f_j = 0 \Rightarrow t_k$ for which $A_k = R_j$ corresponds to $m + 1$;
$f_j = 1 \Rightarrow L_j \notin I$ implies that S is not a threaded list; else t_k for which $A_k = R_j$ corresponds to $m + 1$;
$f_j = 2 \Rightarrow R_j = A_{i_1}$, algorithm terminates; else t_k for which $A_k = R_{R_j}$ corresponds to $m + 1$.

3. If exactly one member of S′, t_*, creates the 1-to-1 correspondence, then S is a threaded list and t_* is called the *head* of the list.

The four elements of each quadruplet of a threaded list carry information about the structure and content of the list. The quadruple $(A_z : f_z, L_z, R_z)$ is called a *word*, and is denoted by z. A_z is the address of z. f_z is the *prefix* part of z. If $f_z = 0$ or 2, then L_z is an elementary symbol, and z is called an *element*. If $f_z = 1$, there exists a word y such that $f_y = 2$ and $R_y = A_z$. In this case, z is the *first word* of a threaded list, and this threaded list is a sublist of the list of which z is a word. A_y is then called the *address* of this sublist.

R_z always contains the address of a word. If z is the first word of a sublist y, then R_z is the address of the next element or sublist of y (this may be a null word, i.e., the word of the form (w: 2, Λ, y), where Λ is the null symbol). If $f_z = 2$, then z is the *terminal word* of the list y of which it is a member, and $R_z = A_y$. Otherwise, R_z gives the address of another word in the list of which it is a member. This word is the null word if A_z is the last sublist of some list, and is not followed by a non-null terminal word.

An example of a threaded list will clarify the above relationship:

1: 1, 2, 0
2: 0, b, 3
3: 1, 6, 4
4: 1, 9, 5
5: 2, e, 1
6: 0, f, 7
7: 1, 11, 8
8: 2, Λ, 3
9: 0, h, 10
10: 2, i, 4
11: 2, j, 7

The sublists of 1 are 3 and 4; the sublist of 3 is 7. It is to be stressed that the numerical sequence of addresses in the above list is of no importance. Since R_z ties each word z to a successor, i.e., "threads" the list, the words are ordered, and it makes no difference what addresses the words occupy in memory. This is, of course, true of list structures in general. The essential innovation of threaded lists is the addition of threads from the end of each sublist of a list, to the next word on the list. This allows a simple, efficient method of sequencing to be carried out on a list, which will arrive at each element of every sublist, without the use of the usual pushdown lists.

3. Program Format

We assume a basic set of *primitive operations*, from which will be built definitions of all operations used to manipulate threaded lists. These definitions are made in the following format:

A *definition* is a string of primitive and/or defined operations, terminated on the left by the symbol ⟨, and on the right by the symbol ⟩, the interior of which is subject to the following structure conditions: The symbol | occurs 4n times, for some $n > 0$. For each $k \leq n$, the symbols and operations occurring between the (4k)-th and (4k + 4)-th |, are termed jointly the kth line. Each such line is divided by the punctuation mark | into four fields, some of which may be empty. The first field, called the kth identifier field, contains an identifying integer n_k. The second field is the kth condition field. The remaining two fields of the kth line are called the kth left and right action fields, respectively. They are denoted by "n_ka" and "n_kb". Each field contains at least one symbol, with the exception that the kth condition field is empty if the kth right action field is empty. Each non-empty field contains a string of operations, separated by commas, and terminated by one of the four symbols "n_k", "n_ka", "n_kb" or "exit", where n_k, n_ka, or n_kb respectively, are non-empty.

A definition is a *program* if the following procedure terminates.

1. The first listed line is examined first.
2. Examine the associated condition field. If its condition is satisfied, execute the operations listed in the associated left action field in order of occurrence. If not satisfied, execute the operations listed in the associated right action field.
3. The last symbol in an action field specifies the line or the field which is to be examined or executed next. Termination occurs when "exit" is encountered as the last symbol of an action field.

4. Primitives

We assume the arithmetic operations and representation of expressions as is customary in algebraic programming languages. Thus, (a−(b/c+d)*f) has its usual meaning if the variables have numerical values when they are encountered in the program.

The expressions A(z), f(z), L(z), R(z) occur only as arguments of operations or programs, or as the elements of relations. They represent the four fields of the word z, as defined above, and are defined if z is the name of a word.

The condition field of any line in a program contains a Boolean expression in relations between quantities of the program. The value of a relation is **true** or **false** according as it is satisfied or not. The value of a Boolean expression is **true** or **false** as determined by the given logical combination of its constituent relations. The empty expression is defined to be **true**. If the value of an expression is **true** then the associated left action field is executed next; otherwise the associated right action field is next executed.

Two relations are given as "atomic", i.e., undefined in terms of the language. They are = and <. = is defined on an arbitrary set of symbols. Either or both of these symbols may be indicated, that is, they may be the contents of some part of a word in a list. Thus, the relations A(z) = "+", f(z) = 1, R(w) = A(L(y)) are defined. The relation < is defined only when the symbols it relates are numbers. These numbers may also be indicated.

We agree to use a standard set of symbols of the sentential calculus i.e., ∧, ∨, ⊃, ≡, in the conditional field, to form combinations of relations. This is simply a shorthand method, for programs logically equivalent to those with logical connectives in the condition field can be written without connectives. For,

1 | ¬R | a, 2 | b, 3 | is equivalent to 1 | R | b, 3 | a, 2 | ;

and

1 | R ∧ S | a, 2 | b, 3 | is equivalent to

1 | R | 4 | b, 3 | 4 | S | a, 2 | 1b |.

Since these two logical operations form a functionally complete set, all logical operations of the sentential calculus may be expressed in terms of them. We are therefore justified in shortening programs by using any sentential connective in the condition fields. We have not, however, introduced any new primitives.

Another shorthand notation used is $R^n(z)$, which indicates the n-fold composition of R with itself. $L^n(z)$ is similarly defined.

The predicate *quote*(z), which occurs in a condition field, transfers control to the corresponding left action field if z¢ contains the symbol " (and perhaps others).

The primitive operation a ← b is defined for a word, a, or the f-, R- or L-field of a word, and b any symbol, written or indicated. This operation places b in the position a. If a is the name of a word, then b is placed in a, right-justified, where a is considered as a unit, and not field delimited. The operation a ← b occurs only in the action fields of a program.

There is assumed to exist a list—though not defined in the language—called the free storage list. A word in this list is said to be free.

The primitive operation list(y, c) forms a list of the following form:

y: 1, b, c

b: 2, Λ y

for b free. If c = R(y), the notation is abbreviated to list (y). There are three operations for inserting free words into lists. Although these operations can be described in terms of the operations already defined, they are given here as primitive operations, since their definition is determined by the mechanics of list representation, rather than the mathematical structure of threaded lists. The first such is isrte(z), where z is a word. The result

of applying isrte to z is dependent on the form of z:

Form of z	*isrte(z)*
z: 0, b, c	z: 0, b, d d: 0, Λ, c
z: 1, b, c	z: 1, d, c
where z is the name of a list:	d: 0, Λ, b
otherwise	alarm
z: 2, b, c	c: 1, a, d
c: 1, a, e	d: 0, Λ, e

where d was free.

The second such operation is isrtw(z), defined as follows:

Form of z	*isrtw(z)*
z: 0, b, c	z: 0, b, d d: 0, Λ, c
z: 1, b, c	z: 1, d, c d: 0, Λ, b
z: 2, b, c	c: 1, a, d
c: 1, a, e	d: 0, Λ, e

where d was free.

Finally, isrtl(z) is defined:

Form of z	*isrtl(z)*
z: 0, b, c	z: 0, b, d d: 0, Λ, c
z: 1, b, c	z: 1, b, d d: 0, Λ, c
z: 2, b, c	z: 0, b, d
c: 1, a, e	d: 2, Λ, c

where d was free.

The significance of these operations will become clearer after the three sequencing operations have been introduced.

5. List Representations

A list may be specified explicitly in programs. In so representing a linear representation is desirable, which does not explicitly indicate internal addresses. The symbols ",", "(", and ")" are reserved to denote list delimiters in this representation. "(" and its matching ")" delimit a compound list, while "," is used to separate the list entries. Thus P ≡ (x, (y, b, c,), d, e) is the list

```
P: 1, 2, 0
2: 0, (, 3
3: 0, x, 4
4: 0, (, 5
5: 0, y, 6
6: 0, b, 7
7: 0, c, 8
8: 0, ), 9
9: 0, d, 10
10: 0, e, 11
11: 2, ),  P.
```

Such a "string" list representation is essentially free of internal addresses and so is useful for input/output purposes. Threaded lists are, however, simpler for processing. Consequently, two programs are defined which transform from one representation to the other. *Thread*(*X*,*y*) takes the "string" list y and creates X as its threaded counterpart. *String*(*y*,*X*) takes the threaded list X and creates y as its "string" counterpart.

thread (X, y) ≡

```
⟨ 1 || seql (y, exit), list (X), seqe (X, exit), 2 ||
  2 | L(y*) = "(" | isrte (X¢), list (X*), 2 | 3 |
  3 | L(y¢) = ")" | 2 | isrtw (X¢), L(X*) ← L(y¢), 2 |
  4 | quote (y¢) | 2 | f(X¢) ← 1, 2 | ⟩
```

string (y, X) ≡

```
⟨1 || seq w (X, exit), list (y), seql (y, exit), 2 ||
  2 | f (X*) = 1 | isrtl (y¢), L(y*) ← "(", 2 | 3 |
  3 | f (X¢) = 0 | isrtl (y¢), L (y*) ← L(X¢), 2 | 4 |
  4 | L (X¢) = Λ | L(y*) ← ")", 2 | isrtl (y¢), L(y*) ←
        L(X¢), 4a | ⟩
```

6. Sequencing

In the manipulation of threaded lists, it is necessary to have modes of sequencing through the lists in such a manner that all of the elements, or all the words, or all of

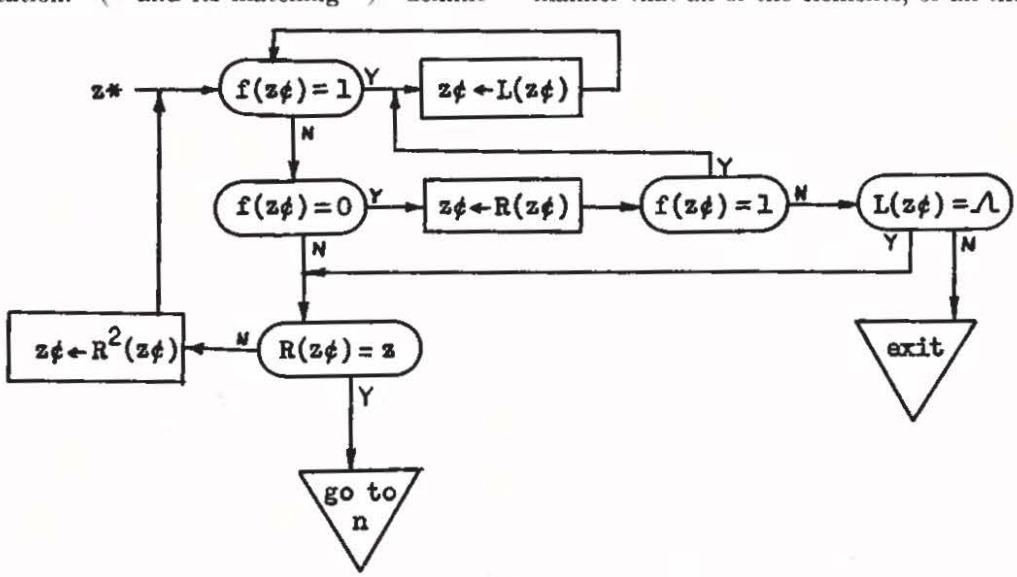


FIG. 1. *The value of z* is the value of z¢ upon exit.*

the sublists, or a given list, will be encountered in some order without repetition. For this purpose, three related primitives are introduced: *seqe*(x,n), *seqw*(x,n), and *seql*(x,n). The generic term for these is *seq*(x,n). These are defined, in each case, when x is a threaded list and n specifies a location field or non-empty action field of the program. A list, called the Sequencing List, is a permanent part of memory. When the instruction *seqe*(x,n), for instance, is encountered in a program, an entry is made in the Sequencing Table (which may be thought of as being a threaded list), at the bottom of the list. The table has the following form:

1	w	w¢	n	e
2	x	x¢	m	e
3	y	y¢	p	1
4	a	a¢	ka	e
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.

The second column contains the names of lists; the fourth column their exit addresses—that is, the field in the program from which an instruction will be taken upon completion of sequencing; the last column contains a symbol indicating the type of sequencing that is to be done on that particular list. Each entry in the Sequencing List has the effect of initializing a sequencing mode for a list. At program compile time, when the table is created, each L¢ is set equal to its corresponding L, which is given a numerical value (the assigned name of the list).

During the running of the program, use is made of another primitive operation, *def*(a,b). This instruction has the effect of allowing a *seq*(a,n) to be executed at run time. Although a and n have already been entered in the table at assembly time, *def*(a,b) changes the entry a to the current value of b (and b of course may be an indicated value, such as L(j)). It is thus possible to have any number of sequencing modes operative simultaneously on any list. This is simply done by giving the list several different names by means of the *def*(a,b) instruction. After program compilation the machine executing the program recognizes each list that is being sequenced, only by its entry in column one of the Sequencing Table.

Once the sequencing modes for each list have been initialized by entry in the Sequencing Table, the actual sequencing through lists can be carried out at run time by the instruction z*. This primitive operation is defined when z is a threaded list. It produces one of three results, depending on which sequencing mode was defined for the list z.

The symbol z¢ means the current position of the sequencer in list z. Before the first instruction z*, this is equal to z. If z was initialized by *seqe*(z,n), z* causes the figure 1 to be executed. This has the effect of sequencing through z and all of its sublists, encountering a different element on each pulse, z*. When the list z has been exhausted, the program transfers to instruction n.

If z was initialized by *seql*(z,n), z* causes figure 2 to be executed. This has the effect of sequencing through the main sublists of z, regarding every element in z itself as a

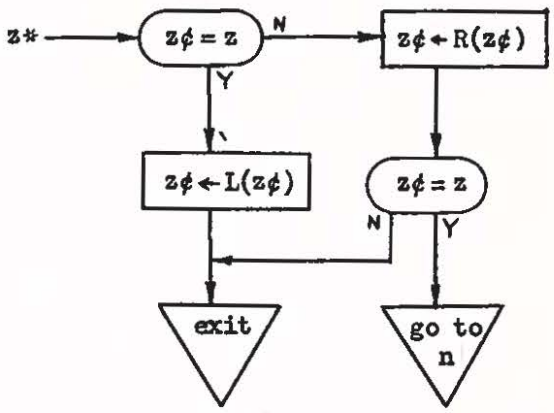


FIG. 2. *The value of z* is the value of z¢ upon exit.*

sublist. Each pulse z* causes a new list to be encountered. When z has been exhausted, the program transfers to n.

If z was initialized by *seqw*(x,n), z* causes figure 3 to be executed. This is identical with *seqe*(x,n), except that every *word* will be encountered exactly once. In

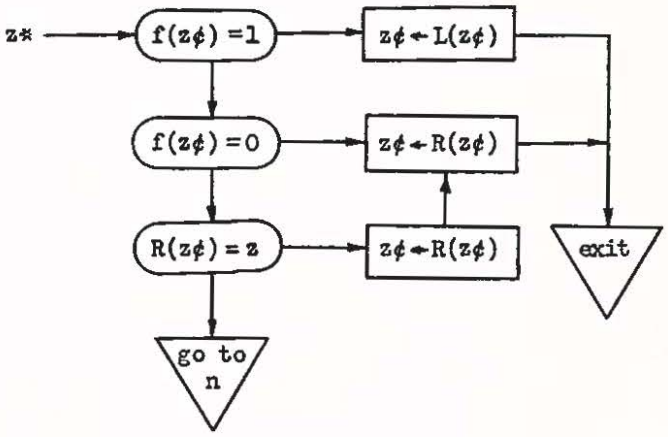


FIG. 3. *The value of z* is the value of z¢ upon exit.*

each sequencing mode, the occurrence of z* causes a sequencing pulse, even if z* is written as part of another expression. In a compound expression, the z*'s are pulsed in the order of execution of the subexpressions in which they occur.

The function of the operations *isrte*(z), *isrtl*(z), and *isrtw*(z) can be viewed as the insertion of a word following the word z. In terms of the sequencing operations, the null word inserted would be the next word encountered after z, when using the corresponding type of sequencing.

Several other sequencing variants appear to be of some use. Thus a conditional variant on each of the three basic sequencing types may be useful. In this mode, the value 0 or 1 stored in some specified location determines whether, upon reaching a list word for which f = 2, sequencing

continues or repeats on the list of which this word is the terminal one. It is also simple to specify a program in the available primitives which, when invoked, will move from the current list position "back and up" to the head of the list which is k levels above the current one.

Part II. Examples

1. Introduction

We have now defined a list language in which all parts of a list structure can be manipulated independently of their content. By using the different sequencing modes, any sublist may be regarded as a single word, or its internal structure can be considered. Efficient sequencing is possible since the threaded property of these lists indicates, at the end of each sublist, how to reach the next word or sublist in the list. Thus, in contrast to the more usual list structure representation, a list of complex structure requires essentially no more storage space during sequencing for housekeeping information than does a simple list. All of the structural information is immediately accessible at the point in the list structure where it is applicable.

Having established a program format for defining new list processing operations, we can proceed to give examples of symbol manipulation that can be carried out using threaded lists. For definiteness, rather than as any indication of its limit of usefulness, we restrict our attention to certain manipulation problems involving algebraic structures, more particularly to polynomials and rational functions.

2. Useful Definitions

First, we will define a few useful operations from the primitives that are available.

exch(x, y) exchanges the contents of x and y, where each may take on one of the forms f(z), L(z), R(z), or z, where z is a word:

exch(x, y) ≡ ⟨1 ‖ T←x, x←y, y←T, exit ‖⟩

copy(x,y) creates a threaded list y which is, except for internal address differences, identical to the list x:

```
copy(x, y) ≡
  ⟨1 | f(x) ≠ 1 | L(y) ← L(x), exit |
    list(y),seqw(x, exit), seqw(y, exit), 2 |
  2 | f(x*) = 1 | isrtw(y¢), list(y*), 2 | 3 |
  3 | f(x¢) = 0 | isrtw(y¢), 3b | L(y*) ← L(x¢), 2 |⟩
```

appndr(x,y) copies the list or word x, adding it as the last word of list y:

```
appndr(x, y)≡
  ⟨1 ‖ seql(y, exit), 2 ‖
  2 | f(y*) = 2 | 3 | 2 |
  3 | L(y¢) = Λ | 4 | isrtl(y¢), y*, 4 |
  4 | f(x) = 1 | isrtl(y¢), copy(x, y¢), exit |
    L(y¢) ← L(x¢), exit | ⟩
```

NOTE: In the preceding and in the sequel, "a" is used to refer to the symbol a itself.

count(x,y,z) counts the number of occurrences of the symbol "x" in the list y and in the sublists of y, and puts this number in location z:

```
count(x, y, z) ≡ ⟨1 ‖ seqe(y, exit), z ← 0, 2 ‖
                  2 | L(y*) = "x" | z ← z + 1, 2 | 2 | ⟩
```

Predicates assign a value **true** or **false** to some specified variable when evaluated. Several such useful predicates follow.

equal(E,x,y) assigns a (logical) value, **true** or **false**, to E depending on whether the list x is the same as the list y—except for internal addressing differences:

```
equal(x, y) ≡
  ⟨1 ‖ seqw(x, exit), seqw(y, exit), E ← true, 2 ‖
  2 | f(x*) = f(y*) | 3 | E ← false, exit |
  3 | L(x¢) = L(y¢) | 2 | 4 |
  4 | f(x¢) = 1 | 2 | 2b | ⟩
```

among(E, x, y) determines if the element x is in the list y:

```
among(E, x, y) ≡
  ⟨1 ‖ seqe(y, exit), E ← false, 2 ‖
  2 | L(x) = L(y¢) | E ← true, exit | 2 | ⟩
```

sublist(E, x, y) determines if the list x is a sublist of the list y:

```
sublist(E, x, y) ≡ ⟨1 ‖ seql(y, exit), E ← false, 2 ‖
                    2 ‖ def(z, y*), equal(x, z), 3 ‖
                    3 | L = true | E ← L, exit | 2 | ⟩
```

3. Manipulation of Algebraic Structures

Algebraic structures can be represented in threaded list form in a natural way. The expression a A b, where A is one of the operations +, −, ×, ÷, is represented by a list of the form:

```
z: 1, 1, 0                 z: 1, 1, 0
1: 0, A, 2      or         1: 0, A, 2
2: 1, a, 3                 2: 0, a, 3
3: 1, b, 4                 3: 2, b, z
4: 2, A, z
```

In the first form 2 and 3 are lists, i.e., expressions, and in the latter both are elements. Combinations of the two types are represented in an obvious way. By the use of parentheses, which of course indicate sublist structure, we can define addition and multiplication for an arbi-

trary number of variables, by the recursive definitions

$$A(a_1, a_2) = A(a_1, a_2)$$

$$A(A(a_1, a_2, \cdots, a_n), a_{n+1}) = A(a_1, a_2, \cdots, a_{n+1})$$

It is convenient to introduce a standard form for the representation of polynomials, which we term the *vertebrate form*. It corresponds to the usual representation of polynomials by terms in powers of the variable. The general list of this form is given in figure 4, with a schematic diagram of the list structure. Notational liberties have been taken for the purpose of readability. Downward pointing arrows represent the address of the next following word; upward pointing arrows represent the address of the preceding word which had the address of the present word in its L-field. This representation of polynomials is not efficient in terms of storage space, but it is efficient in terms of processing.

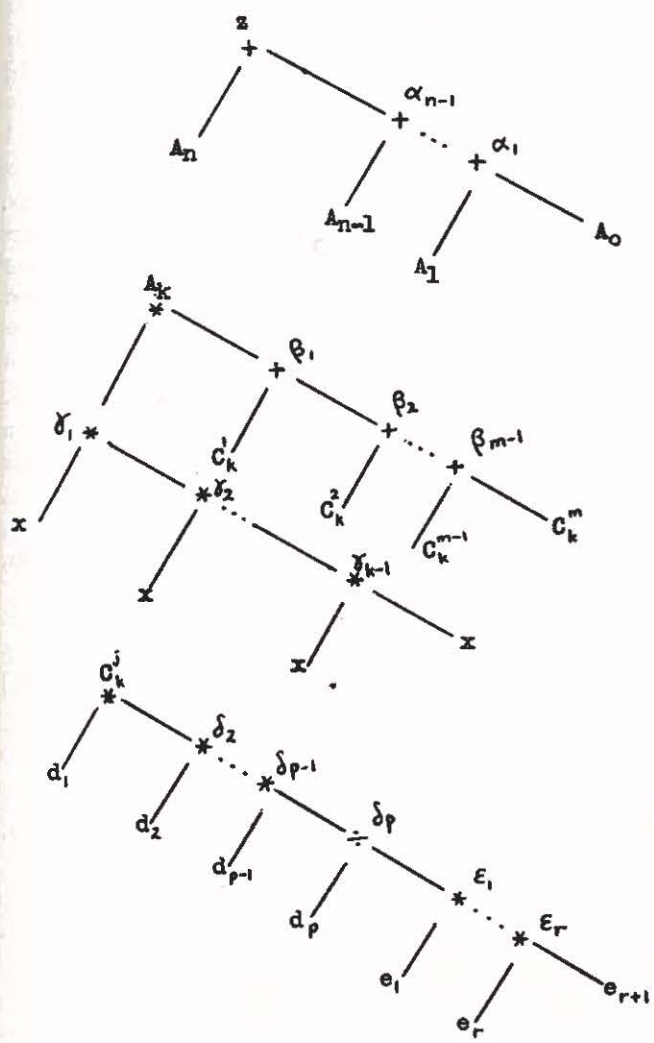
4. Forms

An intuitive notion in symbol manipulation is that of form. A programmatic interpretation of this concept is given in the following. It is not intended to be a completely formal description, but rather to show how form may be defined and used in symbol manipulation. A *format*, P, is an empty threaded list. The string list representation of an empty list is useful for describing formats. A specification of the objects which may occupy the empty places and an assignment of them, one to each empty place in a format, defines a *form*. In the sequel the place occupiers are:

1. specific symbols
2. identifiers of certain sets of symbols
3. identifiers of arbitrary forms.

Hereafter, the symbol A is to identify the universal set of symbols and B is to identify the universal set of all forms.

It is often important to correlate among the occupants of a format. The notation C(k) means that each occupant so designated refers to the same—but otherwise arbitrary—member of the set C. In particular A(2) refers to any symbol, while A(3) refers to an arbitrary symbol which is not necessarily the same as that denoted by A(2).



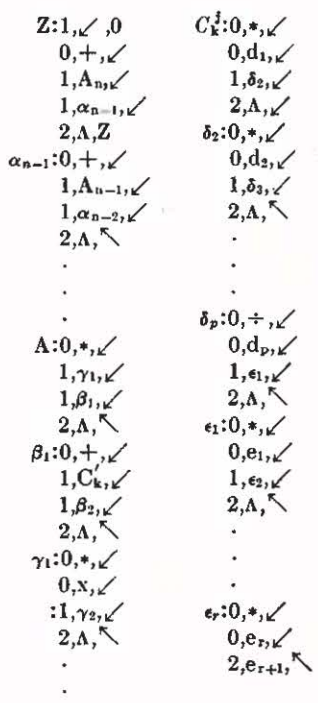


FIG. 4. *The value of z* is the value of z¢ upon exit.*

Forms are used to construct the relation, denoted by L$\mathfrak{F}$P: a list, L, is in the relation *instance of*, $\mathfrak{F}$, to a form, P, by which is meant that the occupant of each place in L, corresponding to a specified place in P, corresponds to the occupant of the specified place in P in the sense:

If x is the occupant of the space in P, then

(i) If x is a list head, e.g., a "(", then the L place occupant, y, satisfies f(y) = 1.

(ii) If x is a specific symbol, y is the same symbol.

(iii) If x is a set designator, then y is an element of x.

(iv) If x is A or B, then y is any symbol or list, respectively.

(v) If x is C(k), then the occupants of all places in L corresponding to the places in P occupied by C(k) must be the same in the sense of the predicate *equal(E,s,t)*.

It is often the case that the property of set membership or symbol equality is inconvenient to apply since more general predicates seem called for. Consequently, a natural and simple generalization is to permit place occupants in P of the form \$(Pk) where \$ is any identifying symbol and Pk identifies a predicate. In the predicate code, \$ stands for its place correspondent in L, and the logical value of the predicate is determined on that basis. Consequently,

(vi) If x is C(Pk), Pk(y) must provide a value of **true**.

Then one of (i) through (vi) must be satisfied for each place in P in order that L$\mathfrak{F}$P.

In the course of determining the logical value of the predicate, $\mathfrak{F}$, specified in a condition field it is convenient to generate a list of pairs of correspondents. It is natural then to define in the associated left action field operations involving the place occupiers of the form P. In the execution of these operations the L-place occupiers replace their P-place correspondents wherever they occur in the action field. In the programs that follow x$\mathfrak{F}$(···) is a short notation for form (E,w,x,thread((···))) and x←$\mathfrak{T}$(···) is a short notation for transform (x,thread((···))). The program *form(E,w,y,x)* which follows does not permit place occupiers utilizing predicates. The list x is a threaded list description of the form P, the list y is the list being checked, the list w is the list of pairs of correspondents generated, and E is the logical value of the predicate:

form(E, w, y, x) ≡

⟨1 ‖ seqw(x, exit), list(w), seql(w, exit), def(p, y$\not\epsilon$), E ← *true*, 2 ‖

2 | f(x∗) ≠ 1 | 3 | 4 |

3 | (L(x$\not\epsilon$) = A(k)) ∧ (f(p) ≠ 1) ∧ (f(x$\not\epsilon$) = f(p)) | prev(E, x$\not\epsilon$, p), 5 | 10 |

5 ‖ def(p, R(p)), 2 ‖

4 | f(y$\not\epsilon$) = 1 | def(p, L(p)), 2 | E ← *false*, exit |

10 | (f(x$\not\epsilon$) = f(y$\not\epsilon$)) ∧ (L(x$\not\epsilon$) = A(k)) ∧ (f(y$\not\epsilon$) ≠ 1) ∨ (L(x$\not\epsilon$) = L(y$\not\epsilon$)) | prev(E, x$\not\epsilon$, p), 7 | 8 |

7 | f(p) = 2 | def(p, R(p)), 5 | 5 |

8 | L(x$\not\epsilon$) = B(k) | 3a | E ← *false*, exit | ⟩

prev(H, r, s) ≡

⟨1 ‖ def(z, w), seql(z, 5), seql(w, 5), H ← *true*, 2 ‖

2 | equal(z∗, r) | 3 | z∗, 2 |

3 | equal(z∗, s) | exit | H ← *false* |

5 ‖ isrtl(w), list(w∗), copy(w$\not\epsilon$,r), isrtl(w$\not\epsilon$), list(w∗), copy(w$\not\epsilon$, s), exit ‖

The program *transform(x, y)* creates the list x from the form y by replacing each place occupier of y by its correspondent in w generated from the program above.

transform(x, y) ≡

⟨1 ‖ seqw(y, exit), seqw(x, exit), 2 ‖

2 | f(y∗) ≠ 1 | 3 | list(x$\not\epsilon$), 4 |

3 | L(y$\not\epsilon$) = A(k) | copy(x$\not\epsilon$, assoc(L(y$\not\epsilon$), w)), 4 | L(x$\not\epsilon$) ← L(y$\not\epsilon$), 4 |

4 ‖ isrte(x$\not\epsilon$), x∗, 2 ‖

As shorthand for *transform(x,y)*, we write y ← $\mathfrak{T}$x. The program *assoc(t, z)* assumes z to be a list of pairs and that t is among the pairs. The list sharing a pair with t is the output of this program(2). Its coding is left as an exercise to the reader.

5. Examples

The program *eval(n)* is used as a subprogram in *div(p,q,r,s,x)*. It is defined for any list, n, of algebraic structure, where all L-field entries are either addresses, arithmetic operations, or numbers. Its output is in L(n), a number which results from evaluating the expression, performing the indicated arithmetic operations.

eval(n) ≡

⟨1 | L(n) ≠ ("+" ∨ "−" ∨ "÷" ∨ "∗") | exit | seqw(n, 1a), 2 |

2 | f(n∗) = 1 | 3 | 2 |

3 | n$\not\epsilon\mathfrak{F}$("−", A(1), A(2)) | L(n$\not\epsilon$) ← A(1) − A(2), 2 | 4

4 | n$\not\epsilon\mathfrak{F}$("+", A(1), A(2)) | L(n$\not\epsilon$) ← A(1) + A(2), 2 | 5 |

5 | n$\not\epsilon\mathfrak{F}$("∗", A(1), A(2)) | L(n$\not\epsilon$) ← A(1) ∗A(2), 2 | 6 |

6 | n$\not\in\mathfrak{F}$("÷", A(1), A(2)) | L(n$\not\in$) ← A(1) ÷ A(2), 2 | 2 | ⟩

With the apparatus of forms at our disposal, we can write a program that, given a list of the form:

z: 1, 1, 0
1: 0, "z", 2
2: 1, a, 3
3: 1, b, 4
4: 2, A, z

where a and b are lists of algebraic structure, will reduce z to a list in which a and b are polynomials in vertebrate form. No further restrictions are placed on a and b. This program is called *simplify*(*z*, *x*), where z is a polynomial in x.

The program *simp*(*R*, *x*) takes any algebraic expression, R, in x, and transforms it into a rational expression, i.e., a quotient of two polynomial expressions in vertebrate form.

simp(R, x) = ⟨1 ‖ thread(S, ("=", R, 1), simplify(S, x), thread(R, ("÷", RL(S), R^2L(S)), exit ‖ ⟩

We can, by means of a relatively simple program, execute the synthetic division of one polynomial by another. The program *div*(*p*, *q*, *r*, *s*, *x*) stores in r the vertebrate form of the polynomial that is the quotient of p and q, and in s the remainder polynomial. p and q are assumed to be in vertebrate form in the variable x.

Given a rational function R, the program *reduce*(*R*, *x*) factors out all common (in the symbolic sense) polynomial factors in numerator and denominator, and leaves the result in R. The input list is the quotient of two polynomials in vertebrate form. This is essentially the Euclidean algorithm for(symbolic) polynomials.

The programs follow:

```
div(P, Q, R, S, x) =
⟨1 ‖ copy(P,S), list(R), copy(R⁴L(Q),B), count
     (x,RL(Q),L(n)), 4 ‖
 4 ‖ copy(R⁴L(P1),A), count(x,RL(S),L(m)),
     L(e) ← L(m) − L(n), thread, (X,(1)),
     copy(Q, B2), 3 ‖
 2 | e = 0 | copy(X,Y), copy(A,A1), copy(B,B1),
     thread(P1,("−", S, ("*", B2, ("÷", A, B)))),
     copy(P1,S), simp(S,x), thread(R1,("+", R,
     ("*", Y, ("÷", A1, B1)))), copy(R1, R), 4 |
     copy(X, X1), thread(X, ("*", "x", X1)), 2 |
 3 | e = −1 | simp(R, x), exit | 2 | ⟩
simplify(Z, X) =
 ⟨1 ‖ seqe(Z, 10), list(a), list(b), 2 | L(a) ← L(a) +
     1, 6 |
 2 | Z$\not\in\mathfrak{F}$("*", ("+", C, D, ), B) | Z$\not\in$ ← 𝔗 ("+", ("*",
     C, B), ("*", D, B)), z*, 1b | 3 |
 3 | Z$\not\in\mathfrak{F}$("+", ("÷", C, D), B) | Z$\not\in$ ← 𝔗 ("÷",
     ("+", C, ("*", B, D)), D), Z*, 1b | 4 |
 4 | Z$\not\in\mathfrak{F}$("*", ("÷", C, D), B) | Z$\not\in$ ← 𝔗 ("÷", ("*",
     C, B), D), Z*, 1b | 5 |
 5 | Z$\not\in\mathfrak{F}$("÷", ("÷", C, D,), B) | Z$\not\in$ ← 𝔗 ("÷",
     ("*", B, D), C), Z*, 1b | 6 |
 6 | Z$\not\in\mathfrak{F}$("*", A, ("+", B, C)) | Z$\not\in$ ← 𝔗("+", ("*",
     B, A), ("*", B, C)), Z*, 21b | 7 |
 7 | Z$\not\in\mathfrak{F}$("+", A, ("÷", B, C)) | Z$\not\in$ ← 𝔗("÷", ("+",
     B, ("*", A, C)), C), Z*, 21b | 8 |
 8 | Z$\not\in\mathfrak{F}$("*", A, ("÷", B, C)) | Z$\not\in$ ← 𝔗("÷" "*",
     B, A), C), Z*, 21b | 9 |
 9 | Z$\not\in\mathfrak{F}$("÷", A, ("÷", B, C)) | Z$\not\in$ ← 𝔗("÷", ("*",
     A, C), B), Z*, 21b | Z*, 2 |
10 | L(a) = 0 | 11 | 1a |
11 | L(b) = 0 | 12 | Z ← A(z) , seqe(Z, 23), 15 |
12 | Z$\not\in\mathfrak{F}$("=", ("÷", A, B), ("÷", C, D)) | Z$\not\in$ ←
     𝔗("=", ("*", A, D), ("*", C, B)), seqe(Z, 10),
     L(a) ← 0, 2 | 13 |
13 | Z$\not\in\mathfrak{F}$("=", ("÷", A, B), D) | Z$\not\in$ ← 𝔗("=", ("*",
     D, B), A), seqe(Z, 10), L(a) ← 0, 2 | 14 |
14 | Z$\not\in\mathfrak{F}$("=", A, ("÷", B, C)) | Z$\not\in$ ← 𝔗("=", B,
     ("*", A, C)), seqe(Z, 10), L(a) ← 0, 2 | Z ← A(Z),
     seqe(Z, 23), 15 |
15 | Z$\not\in\mathfrak{F}$("+", ("+", A, B), C) | Z$\not\in\mathfrak{T}$("+," A,
     ("+", C, B)), Z*, 15 | seqe(Z, 18), seqe(Z, 21),
     list (d), d ← Z, 17 |
17 | Z$\not\in\mathfrak{F}$("*",A,B),("*",C,D)) | Z$\not\in$ ← 𝔗("*",A, ("*",
     B, ("*", C, D))), Z*, 17 | list (c), 16 |
18 | Z$\not\in\mathfrak{F}$("*",C(P1), ("*",A,B)) | Z$\not\in$ ← 𝔗("*",A, ("*",
     C, B)), Z*, 18 | Z*, 18 |
16 | C = 0 | Z ← d, seqe(Z, 15b), 22 | C = 0, Z ← d,
     seqe(Z, 21), 19 |
19 | Z$\not\in\mathfrak{F}$("*", A, B(P2)) | 16 | 18 |
22 | LRL²R²(Z*) = X | exch(LR²(Z)), LRL(d)),
     15 | 22 |
23 ‖ Z ← A(Z), list (C), seql (Z, 25), 23b | count(X, Z*,
     a), count(X, LR²(Z), b), 24 |
24 | a < b | exch(Z, LR²(Z)), C← C + 1, 23b | 23b |
```

```
25 | C = 0 | seqe (Z, exit), 27 | 24 |
26 | Z¢§("+", ("*," A(P3), B), ("+", ("*", D(P4),
   E), F)) | Z¢ ← §("+", ("*", A, ("+", B, E)), F),
   Z*, 26 | Z*, 26 |
21 || L(a) ← L(a) + 1, 2 ||
P1 | C = "X" | T, 18 | F, 18b |
P2 | B ≠ "*" | T, 19 | F, 19b |
P3 || list(n), count(X, A, n) 26 ||
P4 || list(m), count(X, D, m) P41 ||
P41 | m = n | T, 26 | F, 26b | )
reduce(R, x) ≡
  (1 || copy(RL(R), R2), copy(R²L(R), R1), 3 ||
   3 || div(R2, R1, A, B, x), copy(R1, R2), copy(B,
   R1), 2 ||
   2 | L(R1) = 0 | div(RL(R), R1, RL(R), A, x), div
   (R²L(R), R1, R²L(R), A, x), exit| 3 | )
diff(R, x, S) ≡
  (1 || copy(R, T), seqe(T, 5), thread(U, ("+", "x",
   "Δx")), 2 ||
   5 || copy(R, U), thread(S, ("÷", U, T), "Δx"),
   simp(S, "x"), seqe(S, 4), 3 |
   2 L(T*) = x | copy(U, T¢), 2 | 2 |
   3 | L(S*) = "Δx" | L(S¢) ← 0, 3 | 3 |
   4 | simp(S, x), exit || )
poly(X, Y, Z, y, E) ≡
  (1 || list(Z2), L²(Z2) ← 0, copy(RL(Y), R), E ←
   false, def(LRLR²LRL(X), A), count(A, RLRL(x),
   n), count(A, R²L(Y), m), count(A, R²L(x), q),
   count(A, RL(Y), r), e ← L(m) + L(q), d ← L(r)
   ÷ L(n), thread(V, (1)), thread(W, (1)), c ← e, 2 ||
   2 | e = f ∧ int(e) | 3 | E ← false, f ← 0, exit |
   3 | e = 0 | e ← c − f − 1, 4 | copy(Z, Z1), thread
   (Z1, ("*", y, Z1)), copy(RL(x), P1), copy(W, W1),
   thread(W1, ("*", P1, W1)), 3 |
   4 | f = 0 | f ← c − e, thread(T, ("*", U, V)), simp
   (T, A), div(R, T, C, A, A), copy(Z1, Z5), copy
   (Z, Z4), thread(Z2, ("+", (Z2), "*", Z3, C))),
   thread(R, ("−", R, ("*", (C, ("*", U, Z4))))),
   simp(R, A), 5 | copy(U, U1), copy(R²L(X), Q1),
   thread(U, ("*", Q1, U1), 3 |
   5 | L²(R) = 0 | simp(Z, y), exit | 3 | )
```

int(*e*) means e is an integer. This is not considered a primitive, but rather a part of the integer arithmetic we have had available from the beginning.

The program *diff*(*R*,*x*,*S*) differentiates the rational function in list R with respect to the symbol x occurring in R, and stores the result in list S, in the form of a rational function. The program uses direct evaluation by the delta method. This gives some indication of the facility attainable in handling algebraic expressions as threaded lists.

Poly(*E*,*X*,*Y*,*Z*,*y*) takes two rational functions X and Y and determines whether or not X is a polynomial in Y, *true* or false in E, and if *true* stores in Z the vertebrate polynomial form in the designator y such that if Y is substituted in Z for all occurrences of y, X is obtained in a (possibly) transformed representation.

6. Conclusion

Two features of processing of symbolic structures expressed as threaded lists stand out. Considering the complexity possible in algebraic expressions, it is pleasant to be able to manipulate them with so few extensions of the above rather simple primitive operations. Secondly, the methods used in the programs correspond fairly closely to the methods used by people in handling these structures. These facts would seem to be closely related. The three straightforward yet basically different modes of sequencing are defined simply enough, due to the threaded feature of these list structures. They allow us to look at the large-scale or small scale structure of lists, or at their contexts, by means of very direct instructions. And, in fact, this is what people do in processing algebraic expression. Thus, we are enabled to count, or rearrange, or what we will, using "natural" instructions. The coding of the above programs was, in fact, relatively easy, corresponding to the way the programmer would "like" to do them.

7. Implementation

The system, essentially as outlined above, is currently being implemented on the 650 computer system at Carnegie Tech. The TASS assembly system, developed at Carnegie, is being used as the compiler. The powerful macro and subroutine facilities in the assembler make the coding of the primitives and more general codes quite easy to accomplish.

A more general compiler will be developed during the coming summer for a different computer but using the macro assembler as the basis for the compilation technique.

REFERENCES

1. Shaw, J. C., et al. A command structure for complex information processing. *Proceedings of the 1958 Western Joint Computer Conference, May 1958.*
2. McCarthy, John. Recursive functions of symbolic expressions and their computation by machine. Progress Report, RLE, MIT.
3. Robinson, R. M. Primitive recursive functions. *Bull. Amer. Math. Soc. 53* (1950), 925.
4. Robinson, Julia. General recursive functions. *Proc. Amer. Math. Soc. 1* (1950), 703.