Symbol Manipulation Languages

PAUL W. ABRAHAMS

Courant Institute of Mathematical Sciences
New York University
New York, New York

1. What Is Symbol Manipulation?

Symbol manipulation is a branch of computing concerned with the manipulation of unpredictably structured data. Most scientific and business data processing is characterized by the manipulation of data of known length and format. Thus, in the numerical solution of a partial differential equation the representations of the input and output parameters and intermediate results (fixed, floating, double-precision, etc.) are fixed at the time the program for solving the equation is written:

the dimensions of the arrays involved are usually also known in advance or at least do not vary during the running of the program. Similarly, in the preparation of a payroll the exact size and layout of the input and output records and intermediate working storage is given and fixed, and in fact may be stated explicitly in forms such as the data division of COBOL. A little more generally, the length of a payroll may not be known in advance but may be supplied in the course of the problem; or the size of the array of grid points for the differential equation may be changed in accordance with intermediate results of the problem itself. In all these cases, however, the general format of data is fixed and at most some parameters related to size are varied from time to time during the computation.

In contrast, the size and format of the data involved in symbol manipulation are not known in advance and vary greatly during the running of a program. These data are in the form of variable-length *lists*. A list is a sequence of elements, each of which is a data item. A *multilevel list* is one in which the data items may themselves be lists; the latter are called *sublists* of the multilevel list. For instance, a verbal text might be represented as a list of the characters in it. An algebraic expression, after suitable insertion of parentheses, might be represented as a multilevel list; the representation would consist of a list whose elements are the main operator and the list representations of the subexpressions to which this operator is to be applied. Thus the elements of one of these lists would consist of a mixture of sublists and elementary items such as operators, variables, and constants. The number of levels of sublists, i.e., of lists within lists, would correspond to the number of levels of nesting of parentheses.

Symbol manipulation languages vary in regard to the generality of the lists upon which they operate. List processing languages such as LISP, SLIP, IPL, and L6 process lists in their most general form. String processing languages use one-level lists only; these lists are called *strings*, and their items are called *constituents*. The constituents are usually single characters, as in SNOBOL; but they also may be groups of several characters, as in COMIT [*41*]. The distinguishing feature of such languages is that a list cannot itself be an item on a list. Algebraic manipulation languages operate on algebraic expressions; though these expressions are multilevel rather than single-level lists, they are nevertheless a very specialized form. Examples of algebraic manipulation languages are FORMAC [*36*] and Formula ALGOL [*29*, *31*]; since this subject is treated by Sammet [*36*], it will not be further discussed here. Both string processing languages and list processing languages have been used for algebraic manipulation. In general, the more specialized languages take advantage of the specialization by utilizing linguistic

features and implementation techniques that do not work in the more general cases.

A general exposition of symbol manipulation languages, using LISP as an example, was written by this author in 1965 [*1*]. An overview of the state-of-the-art in symbol manipulation about a year later can be gotten by reading the August 1966 issue of the *Communications of the ACM* [*3*], which contains selected papers from the ACM Symposium on Symbolic and Algebraic Manipulation that was held in Washington, D.C. in March 1966. Several papers from that symposium are cited in this article.

Any programming language for symbol manipulation must meet two major requirements. First, there must be appropriate ways of representing lists both on paper (the external representation) and in the memory of a computer (the internal representation). Second, there must be appropriate functions, statement types, subroutines, and other linguistic devices for specifying operations on lists.

1.1 Representation of Lists

We first consider the external representation of lists. For specialized lists such as character strings and algebraic expressions, there are natural written representations. Thus a character string may be written by writing down the characters one after another, enclosing the entire group in quote marks to show where it begins and ends. An algebraic expression may be written, for example, in one of the forms used for arithmetic expressions in scientific programming languages.

For more general lists, the most frequently used written representation of a list consists of the elements of the list written in sequence, delimited by blanks and enclosed in parentheses. Thus,

(CAT 4 DOG)

represents the list whose three elements are the character string CAT, the number 4, and the character string DOG.

((CAT 4) (CENTIPEDE 100))

represents a list whose elements are two sublists. Each of these sublists in turn has two elements.

In representing a list within the memory of a computer, we must indicate both what items are on the list and in what sequence they occur. First, consider the sequencing problem. The simplest way to indicate the sequence of items in a list would be to allocate a block of storage words, store one item per word, and then use a special item to indicate the end of the list. But what size block should we use? Since

the length of the list is not known in advance, we might allow the maximum length—but to do so for every variable would quickly exhaust storage on almost any computer. Even worse, the number of lists needed cannot be predicted in advance, for lists can appear as members of lists, and in addition, in systems such as SNOBOL and LISP, new variables can be created at run time. So clearly some form of dynamic storage allocation is needed. We will return to this point shortly.

We also have to be able to represent the items on a list. If all of the items are of the same kind—single characters or floating-point numbers, say—that is not much of a problem. But if the contents of a list can be heterogeneous, then a problem can arise with "data puns," i.e., different items that happen to be represented by the same configuration of bits. So we must either associate a tag with each item on a list that says what kind of an item it is, or represent items in such a way that data puns cannot occur.

By dynamic storage allocation, we mean that the amount of space allotted to storing the values of variables varies at run time. Thus we need to have a way of obtaining more space when it is needed; and since we will surely run out of space sooner or later, we need to have a way of recovering space that is no longer needed.

Usually, the value of a list variable is stored as a pointer to, i.e., the address of, the machine location where the list actually starts. Some of the possible ways of representing a list in a computer memory are illustrated in Fig. 1. In Fig. 1*a* we see a list represented as an array, with the first cell of the array giving its dimension and the succeeding cells containing the representations of the successive items of the list. A list variable whose value was this particular list would contain in its assigned storage location the address of the first cell of the array. This address would thus be a token of the list. (For lists of characters, the array might pack several to a word and give the number of characters

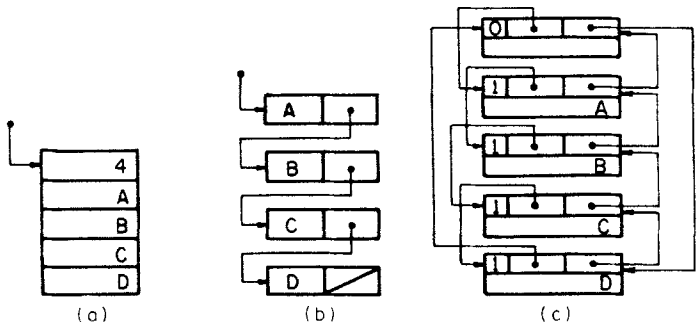


FIG. 1. Three representations of the list (A B C D): (*a*) array, (*b*) one-way list, and (*c*) two-way list.

rather than the number of words in the header.) In Fig. 1*b* we see the representation of the same list as a sequence of linked cells. Each cell contains an item and a pointer to the next cell, i.e., the location of that cell. A token of the list would be the address of the top cell. The last cell has a special indicator for end-of-list, as shown. In Fig. 1*c* we see the list represented in a doubly linked form with a header cell. Each element of this list occupies two words. The header contains pointers to the first and last elements of the list, and each element of the list contains a pointer to its predecessor and a pointer to its successor. The header is the predecessor of the first element and the successor of the last one. A type code distinguishes the list elements from the header, and also distinguishes different kinds of elements from each other. A token of this list would be the location of the first word of its header.

The array representation is convenient in situations where a list, once generated, is never modified directly. In this situation, a list is modified by making a new copy of it with the wanted modifications. The linked-cell representation is convenient when lists are subject to direct modification.

In either representation, there is always a reservoir of available space, often in the form of a *list of available space*. As new lists are created or old ones are enlarged, space is taken from this reservoir for the purpose. Of course, the space has to be available in an appropriate form; if we want a block of 300 cells in a row, it does not suffice to have 150 disjoint blocks of 2 cells each.

The action of returning storage to the reservoir is known as *erasure*; the criterion for erasing a block of storage is that the data stored there will never be used by the program in the future. This criterion will be satisfied if the contents of the storage are inaccessible to the program. For example, suppose that a certain list is the value of a variable and is not the value of any other variable, nor is it a sublist of any other list. Then if the value of that variable is changed, the previous value is rendered inaccessible and thus the list can safely be erased.

Thus the central issue in storage recovery is the determination of whether or not a given block of storage is inaccessible. Depending on the particular language, this determination may be made by the programmer, by the system, or by a combination of the two. If the determination is left to the programmer, then an erasure subroutine is provided; this subroutine is given the address of a list (or other storage to be erased) and returns the storage occupied by this list to the reservoir. Sublists of the list may or may not be erased also, depending on the system. If the system is the determiner of inaccessibility, then a program called the *garbage collector* (cf. Section 2.3) is provided. When the reservoir is exhausted, the system will invoke the garbage collector.

The invocation will take place without any explicit action on the part of the programmer. The garbage collector will then search out all inaccessible lists and return them to the reservoir. The garbage collector can usually be invoked explicitly as well, and in some cases garbage collection may be performed even though the reservoir is not exhausted. Another approach is the one taken by SLIP, where the system accounts for references to lists as elements of other lists, while the programmer accounts for all other references to lists. In general, leaving the responsibility for erasure to the programmer requires the programmer to do more work but leads to a simpler system. A significant disadvantage of leaving erasure to the programmer is that if a list is erased when it is not yet inaccessible, then the resulting program misbehavior may be extremely difficult to debug. Furthermore, failure to erase erasable lists may lead to the rapid exhaustion of storage, and this situation also will be difficult to debug.

Character strings are a special case of lists because in some internal representations several characters may be packed into a list item. There are a number of possible internal representations for character strings which differ in the density of packing and the extent to which pointers are used. In general, the methods that pack information more densely also increase the cost of insertion and deletion of characters. A useful discussion of the alternative internal representations of character strings is given by Madnick [*26*].

Generally speaking, symbol manipulation systems have not had any effective methods for utilizing secondary storage devices. Some facilities of this sort are included in IPL [*28*], and most systems permit input and output to and from named files, which may reside on secondary storage. Some ideas on the efficient use of secondary storage have been published by Bobrow and Murphy [*4*] and by Cohen [*10*].

1.2 Language Features

The operations common to all symbol manipulation languages are those involving the creation and decomposition of lists. At a minimum, it must be possible to create a list by combining existing elements of lists, and to extract a portion of a list. Beyond that, the linguistic treatment of symbol manipulation varies enormously from language to language and is quite difficult to generalize. Further discussion on this topic is therefore left to the discussion of the individual symbol manipulation languages.

Increasingly, symbol manipulation languages have tended to include more general computational facilities. Some of them, e.g., SLIP, Formula ALGOL, and the PL/I list processing facilities, have been achieved by embedding, that is, by adding list processing to an existing language.

For SLIP, there have been several host languages, notably FORTRAN and MAD. Formula ALGOL is an extension of ALGOL, and the list processing features of PL/I were a later addition to that language. On the other hand, LISP and SNOBOL reached the same result via a different path; they started out as pure symbol manipulation languages with only the most rudimentary facilities for anything else, but user requirements pushed them further and further into general computation. LISP 2 and SNOBOL4 are the results.

A useful, though now somewhat outdated, comparison of several symbol manipulation languages is given by Bobrow and Raphael [5].

2. LISP 2

LISP 2 is the most recent version of the list processing language LISP (an acronym for LISt Processing). Its immediate predecessor, LISP 1.5, is described in Section 3. Although only one implementation of LISP 2 exists at the time of this writing, and furthermore that implementation is on a one-of-a-kind computer, LISP 2 is nevertheless a good starting point because of its resemblance to the well-known language ALGOL. LISP 2 was developed jointly by the System Development Corporation (SDC) and Information International, Inc.; the present implementation on the Q32 time-sharing system at SDC was completed in 1967. A general description of LISP 2 is given by Abrahams *et al.* [2]; a more precise definition appears in a series of technical notes issued by SDC [38]. The description here is based on those technical notes.

LISP 2 was developed in order to correct some of the deficiencies of LISP 1.5, most importantly its inconvenient input language and its gross inefficiency in numerical calculations. In order to remedy the difficulties with the input language, LISP 2 adopted an ALGOL-like source language (SL); an intermediate language (IL) resembling LISP 1.5 was also provided. The difficulties with numerical calculations were remedied through the introduction of type declarations and an optimizing compiler.

The advantages of LISP 2 are its symbol manipulating capabilities, its ability to treat programs as data, its flexibility in handling many different types of data, and the ease with which the basic system can be modified. Its disadvantages are its excessive space consumption and the complexity of its specifications; these disadvantages are probably responsible for the difficulties encountered in implementing it.

2.1 Data

LISP 2 data are of two kinds: elements and ntuples. Elements, which we discuss first, consist of numbers, Boolean values, strings, symbols, and

functions. Ntuples consist of nodes (a generalized form of list), arrays, and additional programmer-defined data types.

There are three kinds of numbers in LISP 2: reals, integers, and binary numbers. Integers and binary numbers differ primarily in their external representations. The external representations of real and integer numbers are similar to those of FORTRAN; the external representation of a binary number consists of a sequence of octal digits followed by the letter *Q* followed by an optional scale factor.

The Boolean data consist of TRUE and FALSE.

A string is externally represented by a sequence of characters enclosed by the character "#", e.g., "#STRING#". Within the string, any sequence "'*c*", where *c* is any character, is equivalent to the character *c* by itself. Thus it is possible to include the characters "#" and "'" within a string.

Symbols consist of identifiers, characters, special spellings, and mark-operators. The external representation of an identifier consists of a sequence of letters, digits, and periods starting with a letter. The external representation of a character consists of "¢" followed by the desired character, e.g., "¢+" or "¢A". The external representation of a special spelling consists of a string preceded by "%," e.g., "%# THIS IS AN IDENTIFIER #". The external representation of a mark-operator consists of a sequence of operator characters such as "+" or "**".

In LISP 2, functions are a kind of datum. The external representation of a function depends upon whether the function is being read in or printed out. The external representation of a function to be read in is of the form "[FUNCTION *name*]" where *name* is an identifier that names the function. The actual datum thus denoted is a compiled subroutine. Functions as data are one of the unusual features of LISP. For instance, in LISP 2 it is possible to form an array whose elements are functions; this is not possible in ALGOL, FORTRAN, or PL/I without the use of elaborate artifices.

Lists are represented externally in the notation described in Section 1.1, and internally in the form of one-way lists. The identifier NIL is used as the list terminator. The cells that compose a list are called *nodes*; each node contains the location of an item and the location of the next node in the list. For multilevel lists, the item may itself be a node. Thus a node is really a datum containing the location of two other data. The LISP function CAR, when applied to a node, yields the first component of the node, i.e., the list item; the LISP function CDR,[1] when

[1] These names originated with the early implementation of LISP on the IBM 704; CAR stands for Contents of Address Register and CDR stands for Contents of Decrement Register.

applied to a node, yields the second component of the node, i.e., the remainder of the list.

Nodes can be used to represent more general structures than lists, since the CDR component of a node is not restricted to be another node or NIL. These generalized lists are actually binary trees with the restriction that only the end points can be labeled. They are represented externally in a notation utilizing dots; "$(\alpha \,.\, \beta)$" represents the node whose CAR component is α and whose CDR component is β. This notation can be generalized; thus "$(\alpha_1 \alpha_2 \ldots \alpha_n \,.\, \beta)$" represents a structure obtained from the list $(\alpha_1 \alpha_2 \ldots \alpha_n)$ by replacing the terminating NIL by β. In Fig. 2 we see some examples of these generalized lists, together with

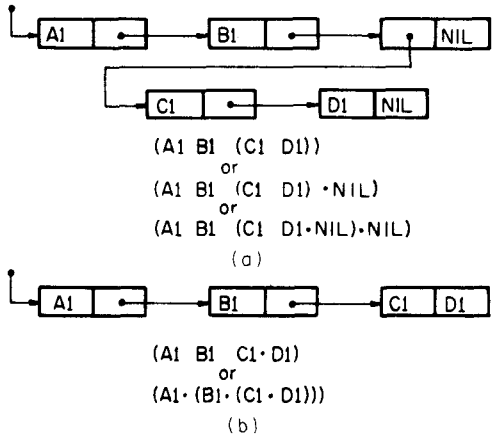


FIG. 2. Examples of node structures.

some (but not all) of their alternative external representations. An ordinary list can be represented externally in the dot notation as well as in the notation introduced earlier. It follows, then, that the external representation of a list is not unique. When a node is printed, the external representation used is the one with the minimum number of dots; thus, ordinary lists are printed in the usual way.

In Fig. 3 we see another example of a node structure as represented both internally and externally. In Fig. 3*b* there is one node that is pointed to from two places. This node is used to represent a merge point of the binary tree descending from the top node and corresponds to a repeated part of the external representation. Note that the node structure of Fig. 3*a* has the same external representation as that of Fig. 3*b* but uses more storage.

An array is represented internally as a sequential block of cells preceded by a header word giving the type of the elements and the

dimensionality. All of the elements must have the same type. Externally, an array is represented as a sequence of elements enclosed in brackets and preceded by the array type. For multidimensional arrays, several levels of bracketing are used.

The programmer may define *ntuples* in addition to nodes and arrays. In general, an ntuple is an ordered collection of data. Associated with each kind of ntuple is a collection of coordinate functions by means of which the individual components of a particular ntuple may be extracted. In the case of a node, the coordinate functions are CAR and CDR.

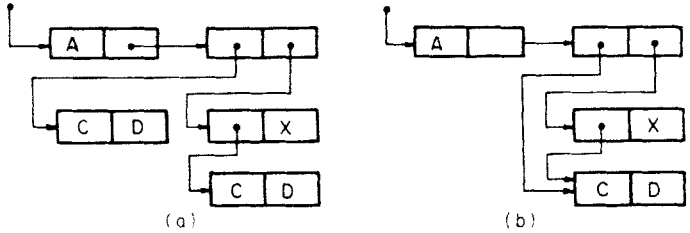


FIG. 3. Two representations of (A · ((C · D) · ((C · D) · X))): (*a*) no common storage and (*b*) common storage.

In the case of an array, there are as many coordinate functions as there are elements in the array; the application of a coordinate function to an array is expressed in the usual subscript notation. In the case of a programmer-defined ntuple, the coordinate functions are specified by the programmer in much the same way as a data structure is specified in COBOL, PL/I, or JOVIAL. Since LISP 2 was developed at SDC, which also developed JOVIAL, the influence of JOVIAL data structure specification on LISP 2 has been strong.

For any datum except a function, a number, or a Boolean, the token of the datum is a location. Thus the problem of data puns arises only for functions, numbers, and Booleans. The solution adopted by LISP 2 is somewhat complicated, and is based upon the use of type declarations, both implicit and explieit. We use the case of the integer 479 as an example. There are two possible internal representations for 479, as shown in Fig. 4. One representation consists of the number itself; the other consists of a pointer to a one-word array whose header indicates that the number is in fact an integer (as distinguished from a real, say). If we wish to add two variables whose values are integers given in the pointer representation, then we must trace down the pointers, locate the actual numbers, and add them. If the resulting value is to be in the same form, then we must create a new array and pass along the pointer to it as the result of the addition. On the other hand, if the variables in question had their values represented directly, then two fetches, an

add, and a store would suffice. The advantage of the direct representation is that it leads to efficient calculation; the advantage of the indirect representation is that the data are self-descriptive.

The type of a data token is a rule for interpreting it, i.e., for determining the datum that the token represents. A field is a location within LISP's storage that is capable of holding a data token; every such field has a type associated with it. Examples of such fields are storage words reserved for variables, the pushdown stack used for temporary storage, the CAR and CDR portions of a node, and the elements of an array. If the type associated with a field is INTEGER, say, then 479 would be stored in that field in its direct representation—but that field could only be

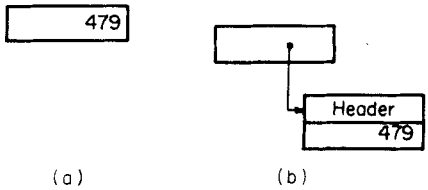


FIG. 4. Two representations of 479: (*a*) direct and (*b*) indirect.

used to hold integers, and could not be used to hold nodes, arrays, functions, etc. If the type associated with a field is GENERAL, then any datum whatsoever can be stored there—but any such datum must be in the form of a pointer. Thus GENERAL is used to describe fields where any kind of datum might be stored.

The type of a variable is determined by a type declaration made for the variable, or by default. The type of a part of an ntuple is determined by the definition of that kind of ntuple. In particular, the CAR and CDR portions of a node are always of type GENERAL (and thus cannot be used to hold numbers in their direct representation). For arrays, the type of the elements is determined by the header; the array as a whole is treated as being GENERAL. Thus a datum declared REAL ARRAY will contain elements of type REAL, i.e., actual numbers; an array containing those same numbers in the indirect representation would have type GENERAL ARRAY.

2.2 Program Structure

A LISP program is specified as a sequence of declarations of various kinds. The most important kind of declaration is the function definition, which is equivalent to a procedure declaration in ALGOL. Declarations are made under the aegis of the LISP Supervisor, which recognizes two kinds of actions: declarations and evaluation of expressions. In order to run a program in the usual sense, one first defines a

function that carries out the desired operations and then invokes this function by evaluating an expression. Since functions can themselves call functions, one can construct hierarchies of functions in the same way that one constructs hierarchies of procedures or subroutines. Recursion is permitted and indeed (in the LISP community) encouraged and admired.

Although LISP 2 introduces quite a number of extensions to ALGOL in program structure as well as in data structure, remarkably few of these extensions are peculiar to the needs of list processing. Most of these needs are met purely through the introduction of appropriate data types and conversion rules among data types. Therefore, although we will at least mention most of these extensions, we will not dwell upon them in detail.

A LISP 2 function definition consists of two parts: the heading and the body. The heading gives the name of the function, the names and types of the formal parameters, and the type of the value returned by the function. The body is (unlike ALGOL) an expression; evaluation of this expression gives the value of the function. A simple example of a LISP 2 function definition (in SL) is the recursive definition of the factorial function:

```
FACTORIAL FUNCTION(N) INTEGER; N INTEGER:
  IF N = 0 THEN 1 ELSE N* FACTORIAL(N — 1)
```

The corresponding definition in IL is

```
(FACTORIAL FUNDEF (FUNCTION (FACTORIAL INTEGER) ((N INTEGER))
(IF ( = N 0) 1 (* N (FACTORIAL (— N 1))))))
```

In both of these examples, the first line is the heading and the second line is the body. (Since SL and IL are both written in a free-field format, this arrangement is not required.) This particular function happens to have a recursive definition.

In LISP 2, evaluation of an expression yields a *valuation*; valuations are characterized by a type and a reference mode. If the reference mode is NOVALUE, then the valuation consists of nonsense information and the expression may only be evaluated for its side effects. If the reference mode is anything else, then the valuation has as part of it a value, which is a data token. The reference mode then determines how the value is to be obtained from the valuation. The UNFIELDED, DIRECT, and INDIRECT reference modes are illustrated in Fig. 5. Evaluation of an UNFIELDED expression yields a value only; the location containing that value is not accessible to the program. Constants always have the UNFIELDED reference mode. Evaluation of a DIRECT expression yields a pointer to a field containing the value. Evaluation of an INDIRECT

expression yields a pointer to a field whose contents are in turn a pointer to a field containing the value. The interpretation of this value is, of course, determined by the type of the valuation. The type and reference mode of an expression are determined completely by the expression itself and the context in which it appears; they do not vary from one evaluation of the expression to the next. The actual value and the various pointers involved may very well vary from one evaluation to the next.

The rationale behind this particular generalization was to permit assignments to be made to components of list structures and other ntuples as well as to arrays and variables. In both ALGOL and FORTRAN, the left side of an assignment statement must be either an ordinary variable or an array reference; PL/I permits certain more general forms,

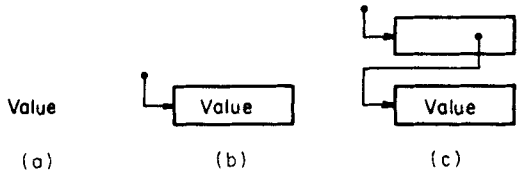


FIG. 5. Reference modes of an expression: (a) unfielded, (b) direct, and (c) indirect.

but does not permit functions to appear on the left side of an assignment statement. In LISP 2, the reference mode of the left side of an assignment must be either DIRECT or INDIRECT. (The default reference mode of a variable is DIRECT.) Expressions with either of these reference modes satisfy the essential requirement for the left side of an assignment, namely, they provide a location where a value can be placed. An expression such as "CAR(A)", where A is a variable whose value is a node, has the DIRECT reference mode and thus designates the actual field which is the CAR part of A. The introduction of reference modes also permits more general expressions, e.g., conditionals, to appear on the left side of assignments.

A *block* consists of a sequence of *statements* enclosed in the statement brackets "BEGIN" and "END", and preceded by some declarations of variables local to the block. The declarations are separated from each other by semicolons, and the last one is followed by a colon. If there are no declarations, "DO" without the colon is used instead of "BEGIN:" to avoid certain syntactic ambiguities. The following kinds of statements are permitted:

(1) *Expressions.* When an expression is encountered in a context where a statement is expected, the expression is evaluated and the

resulting valuation is simply discarded. In particular, assignments are accomplished through the evaluation of assignment expressions, described below, rather than through a distinct statement type.

(2) *Block statements.* These are as in ALGOL.

(3) *Compound statements.* These are as in ALGOL.

(4) *Go statements.* These are of the form "GO α", where α must be a label. Label variables such as those used in PL/I are not permitted.

(5) *Conditional statements.* These are as in ALGOL.

(6) *Case statements.* These resemble the computed GO TO of FORTRAN and the GO TO statement with a designational expression in ALGOL.

(7) *Return statements.* The statement "RETURN α" has two effects: it causes exit from the block containing it, and it causes that block to have a valuation, namely, the valuation of α. Under certain circumstances, execution of a return statement may cause an exit from surrounding blocks as well. The return statement is one of the more pleasant features of LISP 2, and in fact numerous modifications to ALGOL have introduced similar facilities.

(8) *Code statements.* These are as in ALGOL.

(9) *Try statements.* A try statement has the form "TRY v, s_1, s_2" where v is a variable and s_1 and s_2 are statements. First the statement s_1 is executed. If during the execution of this statement the function EXIT (of one argument) is evaluated, then control reverts through as many levels as necessary to return to the try statement. The value of v becomes the value of the argument of EXIT, and the statement s_2 is executed. If no EXIT is encountered during the execution of s_1, then control simply proceeds to the statement following the try statement.

(10) *For statements.* For statements are similar to those in ALGOL, but include additional special forms appropriate to list processing. For instance, the statement:

$$\text{FOR } v \text{ IN } l : s$$

causes the statement s to be executed once for each element in the list l; during these successive executions, the successive values of v are the successive elements of l.

Assignments are performed by assignment expressions, which are of the form:

$$\alpha \leftarrow \beta$$

Here α must have reference mode DIRECT or INDIRECT and β may have any reference mode except NOVALUE. The valuation of the entire assignment expression is simply the valuation of β; however, evaluating the assignment expression has the side effect of replacing the value of α by the value of β, i.e., changing the contents of the field that contains

the value of α. Since assignments are expressions, they can be embedded within actual parameters of function calls, for instance. Nested assignment expressions are permitted.

Operations on lists are accomplished primarily through the application of appropriate functions rather than through special syntactic devices (the for statement being an exception). The basic operations on lists are performed by the functions CAR and CDR, by the infix operation "$\circ$" (read as "cons" for *cons*truct), and by equality testing. The functions CAR and CDR each expect one argument, which must be a node, and return as value respectively the CAR and CDR parts of that node. The expression "$\alpha \circ \beta$" creates a new node whose CAR part is α and whose CDR part is β. Equality testing is accomplished by extending the definition of the equality operator "$=$" to ntuples; two ntuples are equal if their external representations are equal. (There is a different equality test for actual identity of pointers; this other test will distinguish different copies of the same list with different internal representations but the same external representation.) In addition to these basic facilities, LISP provides a library of other useful functions, e.g., APPEND(x,y) which expects two lists as arguments and returns a new list whose elements consist of the elements of x followed by the elements of y.

In order to use identifiers and nodes as constants within a program, a quotation convention is needed. Otherwise there would be no way to distinguish the identifier "ABC", used as a constant, from the variable "ABC". Therefore, any constant may be preceded by a "'", and constant identifiers must be preceded by a "'". Thus "127" and "'127" are both the same numerical constant; "ABC" is a variable; and "'ABC" is a constant, as is "'(A B C)".

As we mentioned earlier, functions can be used as data. For example, consider the following sequence of statements:

```
A ←FUNCTION(X,Y); X,Y REAL: X↑2 + Y↑2 − X*Y;
X ←A(2,5);
A ←FUNCTION(X,Z); X,Z REAL: X↑2 + Z↑2 + X*Z;
Y ←X + A(1,2)
```

Evaluation of the right side of the first assignment expression yields a function. (Recall the remark earlier that an expression can be used in any context where a statement is expected.) Within this function X and Y are dummy variables and bear no relationship to the X and Y appearing on the left side of the assignment expressions. After the sequence of statements is executed, the value of the variable X is 19 and the value of the variable Y is 26.

In Fig. 6 we see a LISP 2 program that computes the longest common segment (i.e., subsequence of elements) of two lists. It uses a version of the for statement, governed by " ON ", in which the controlled variable assumes as successive values the initial list, the initial list less its first element, the initial list less its first two elements, etc. In the block

```
% R LCS FINDS THE LONGEST COMMON SEGMENT OF TWO LISTS
% R L1 AND L2
LCS FUNCTION(L1, L2):
  BEGIN X, Y, BEST GENERAL; K, N, LX INTEGER; LX ← LENGTH(L1);
    FOR X ON L1 WHILE LX > K:
      BEGIN INTEGER LY; LY ← LENGTH(L2);
        FOR Y ON L2 WHILE LY > K:
          DO N ← COMSEGL(X,Y);
            IF N ≤ K THEN GO A;
            K ← N;
            BEST ← COMSEG(X, Y);
          A: LY ← LY - 1;
          END;
        LX ← LX -1;
        END;
      RETURN BEST;
    END;

% R COMSEGL FINDS THE LENGTH OF THE LONGEST INITIAL COMMON
% R SEGMENT OF TWO LISTS X AND Y
COMSEGL FUNCTION(X,Y) INTEGER: IF NULL(X) ∨ NULL(Y)
    CAR(X) ≠ CAR(Y) THEN 0 ELSE COMSEGL(CDR(X), CDR(Y)) + 1;

% R COMSEG FINDS THE LONGEST INITIAL COMMON SEGMENT OF TWO
% R LISTS X AND Y
COMSEG FUNCTION(X,Y): IF NULL(X) ∨ NULL(Y) ∨ CAR(X) ≠ CAR(Y)
   THEN NIL ELSE CAR(X) ∨ COMSEG(CDR(X), CDR(Y));
```

FIG. 6. A LISP 2 program.

declarations, the assignments are used to specify initial values to be used upon entering the block. The function NULL yields TRUE as value if its argument is the empty list and FALSE otherwise. The function LENGTH(x) has as its value the length of the list x. The types of all formal parameters and of all functions except COMSEGL are assumed to be GENERAL by default. Initial values not specified explicitly are determined by default; in particular, the default initial values for the INTEGER variables are all 0 and for the GENERAL variables are all NIL.

The basic concept in LISP 2 input–output is the file. A file is a named data stream associated with an input–output device such as tape, a disk, or a printer. Many files may exist at the same time; of these, one is selected to be the input file and one to be the output file. The input file acts as a source of single characters, and the output file acts as a sink of single characters.

Input and output are defined in terms of two basic functions: READCH and PRINCH. READCH( )[2] has as its value the next character. that can be read from the input file; PRINCH(x) writes the character x into the output file, and incidentally has x as its value. The functions READ and PRINT are defined in terms of READCH and PRINCH, respectively, and these read or write the external representation of a single datum. (Since a datum may be a complicated list structure, it may occupy several lines.) There no are input or output statements as such; all input and output is done by means of functions.

LISP 2 does not have any formatting in the usual sense. Because of its variable length, symbolic data raise unusual problems in formatting; however, since LISP 2 is also intended for numerical use, the lack of formatting is a serious drawback. LISP 2 does provide for the handling and adjustment of margins. For any file, a left and right margin may be specified, subject to the physical line length limitations of the device associated with the file. The first character on any line is read from or written onto the left margin position, which need not be the first available character position on the line. When the character position moves past the right margin, a user-specified margin overflow function is invoked. Similar functions exist for page position, and there is also a tab function for positioning within a line. These formatting functions are independent for different files, and may be modified dynamically during input or output.

There are selection functions, INPUT(f) and OUTPUT(f), that select an input file or an output file, deselecting the previous input or output file. The value of each of these functions is the previously selected file. When a file is deselected, its entire state is preserved, and again restored

[2] The "( )" notation indicates that READCH is a function of zero arguments.

when the file is reselected. Thus the appearance of simultaneous input and output on several files can be maintained with no difficulty; the user merely selects the file he wishes to operate upon before performing the operation. The function OPEN associates a file with a physical device; the function SHUT breaks this association.

2.3 Implementation of LISP 2

The LISP 2 system provides an environment in which LISP 2 programs can be read in, compiled, and executed. There is no sharp division between these activities, and the user may shift back and forth among them. The principal components of the system are:

(1) *Supervisor*—handles over-all control and processes requests for action.
(2) *Syntax-directed translator*—translates SL to IL.
(3) *Compiler*—translates IL to assembly language.
(4) *Assembler*—translates assembly language into relocatable binary code.
(5) *Input–output functions*—handle reading, printing, and file manipulation.
(6) *Garbage collector*—recovers abandoned storage.
(7) *Library*—provides a collection of useful functions.

Programs may be brought into the system either by typing them in on a terminal device or reading them in from a file. Under user control, programs may then be translated successively from SL to IL by the syntax-directed translator, from IL to assembly language by the compiler, and from assembly language to code by the assembler. They may be called by giving the function name and a list of arguments. (All of the translators are themselves callable programs.) There are two reasons for the division of the translation process into several stages. First, the various intermediate forms are themselves useful languages. In particular, programs that operate on programs work much more easily with IL than with SL. Second, the task of translating from SL to IL is primarily one of pattern-matching in one-level lists, while the task of translating from IL to assembly language is primarily one of complex structure manipulation. The tools that are appropriate for one task are not the best for the other.

The garbage collector is perhaps one of the most interesting features of the LISP 2 implementation. Many of the ideas used in it are due to Ross [*34*, *35*]. Storage areas are set aside for various kinds of data structures used by LISP. Some of these are arranged in pairs, where one member takes space from the bottom up and the other takes space from

the top down. When any area is exhausted, or when the boundaries of two paired areas meet, a garbage collection is necessary. Garbage collection proceeds in four phases:

(1) *Marking.* All active data structures are traced, and a mark is associated with each. The mark may be placed either directly in the structure or in a bit table. Any data not marked are known to be inaccessible to the program and therefore may be safely erased.

(2) *Planning.* The planning phase is a preparation for the moving phase which follows it. During the moving phase, various data structures are relocated. In the planning phase, the new location of each data structure is determined and recorded.

(3) *Fixup.* During the fixup phase, all pointers to data structures are updated to reflect the new location of the data structure. These pointers will occur both in temporary storage areas of the program itself and within data structures. The fixup phase also includes the modification where necessary of executable code. This modification is directed by a bit table associated with the code.

(4) *Moving.* Storage is rearranged in order to pack active data and recovered space into solid blocks.

The garbage collector is actually initiated when one of the basic LISP structure-creating functions cannot obtain the storage that it needs. After the garbage collector is finished, control returns to the function that called it, and this function then proceeds to create the structure that it could not create previously. Although the garbage collector can be invoked explicitly by the user, it never needs to be; the LISP system itself will invoke it when it is needed.

3. LISP 1.5

LISP 1.5 is, historically, the programming language that led to LISP 2. LISP 1.5 is in turn derived from the original LISP 1 as described by McCarthy [*24*]. LISP 1 was characterized by great elegance, but in practice it turned out to be a language in which it was impossible to write useful programs. This situation led to many additions to LISP 1, and the result of these additions has become known as LISP 1.5 (since it was believed to be halfway between LISP 1 and LISP 2). The definition of LISP 1.5 is somewhat imprecise, in that there exist a number of implementations of LISP which are considered by both their authors and users to be LISP 1.5 but which differ in many details. The two best-documented versions are LISP 1.5 for the IBM 7090 [*25*], developed at MIT, and LISP 1.5 for the System Development Corporation time-sharing system [*39*]. A collection of articles describing a number of

LISP 1.5 applications, implementations, and improvements has been published by Information International, Inc. [*16*] and subsequently reprinted by the MIT Press. Unlike LISP 2, LISP 1.5 has been widely implemented and widely used.

3.1 Pure LISP

Pure LISP exists as a language for defining functions of symbolic expressions, known as S-expressions. An S-expression is a particular case of the node structures of LISP 2. These S-expressions are built up from atoms, which are the same as the identifiers of LISP 2; S-expressions are defined as follows:

(a) Every atom (i.e., identifier) is an S-expression.
(b) If α_1 and α_2 are S-expressions, then $(\alpha_1 \,.\, \alpha_2)$ is an S-expression.

In other words, if α_1 and α_2 are S-expressions, then the node whose CAR component is α_1 and whose CDR component is α_2 is an S-expression. The various alternative notations for nodes are acceptable, e.g., $(\alpha_1\ \alpha_2 \ldots \alpha_n)$ is equivalent to

$$(\alpha_1 \,.\, (\alpha_2 \,.\, \ldots \,.\, (\alpha_n \,.\, \text{NIL}) \ldots))$$

There are five basic functions in pure LISP that operate on symbolic expressions: CAR, CDR, CONS, EQ, and ATOM. CAR, CDR, and CONS are as in LISP 2. EQ(x,y) is defined if and only if at least one of its arguments is an atom. Its value is the atom T (for "true") if x and y are the same S-expression, and F (for "false") otherwise; ATOM(x) has as its value T if x is an atom, and F otherwise. It is defined for all S-expressions.

A LISP function is itself represented as an S-expression, in a form quite similar to that of the intermediate language of LISP 2. The form

$$(f\ \alpha_1\ \alpha_2 \ldots \alpha_n)$$

indicates the application of the function f to the arguments $\alpha_1, \alpha_2 \ldots \alpha_n$. The application is carried out by evaluating $f, \alpha_1, \alpha_2, \ldots, \alpha_n$ in sequence and then applying the value of f (which must be a function of n arguments) to $\alpha_1, \alpha_2, \ldots, \alpha_n$. The form

$$(\text{COND}\ (p_1\ e_1)\ (p_2\ e_2) \ldots (p_n\ e_n))$$

resembles the conditional expression of LISP 2. It is evaluated by evaluating the p_i in turn until one is found whose value is T. The value of the entire form is then obtained by evaluating the corresponding e_i. None of the other e_i's are evaluated, nor are any of the p_i following the first true one.

A function is represented in the form:

$$(\text{LAMBDA } (x_1\ x_2 \ldots x_n)\ \alpha)$$

where the x_i are atoms representing dummy variables appearing in the expression α. Application of a function to arguments is carried out by substituting the value of each argument for the corresponding x_i in α and then evaluating the result of this substitution

In order to permit recursive functions to be expressed in closed form, an additional device is needed. Evaluation of the form

$$(\text{LABEL } f\ \alpha)$$

yields the function α (which must be a LAMBDA expression) and in addition associates the function name f (which must be an atom) with α so that during the application of α to arguments, any occurrence of f evaluates to α. Thus a function may be made recursive by naming it via LABEL and then using this name within the definition, i.e., within the LAMBDA expression. Although the LABEL device is necessary in pure LISP, it has virtually no application in actual programming because of much more convenient mechanisms for defining recursive functions.

Given the apparatus just described—the five basic functions, application of functions to arguments, LAMBDA, LABEL, and conditional expressions—it is possible to write an interpreter for LISP in LISP. This interpreter is the analog of a universal Turing machine, in that its input is a LISP function together with arguments, its output is the result of applying that function to the arguments, and the interpreter itself is written in the same language as the program that it is interpreting. Much of the interest in LISP from the standpoint of the theory of computation devolves from the fact that LISP is universal in this sense.

3.2 Features Added to Pure LISP

While pure LISP was a thing of beauty to computational theorists, it turned out in practice to be inadequate to the needs of writing programs. At the same time, its simplicity, self-interpretive properties, and symbol manipulating capabilities made it a desirable basis on which to develop a usable language. The main improvements that characterize LISP 1.5 are:

(1) *Definitions and permanent values.* It is possible to associate a value with any identifier. In the case of an identifier whose value is a function, the association is created through use of the LISP function DEFINE. Normally, a LISP program consists of a sequence of applications of functions to arguments. Thus, in order to create a complicated

function with many subfunctions, DEFINE is used to associate the definition of each function with its name. Any of these functions may refer to any other function or to itself by name within its definition.

In addition, it is often useful to assign constant symbolic expressions as values of certain atoms. The function CSET has two arguments: an identifier and a value. Evaluation of CSET causes the value to be associated with the identifier; any subsequent evaluation of the identifier will yield this associated value.

More generally, an identifier has associated with it a *property list.* Specific properties of an identifier (of which its CSET-assigned value is one) are indicated by placing the name of the property on the property list followed by the associated value. Properties without associated values are also permitted. As an example, in an interpreter-based system one of the properties of a function name will be EXPR. The identifier EXPR will appear as an element of the property list of the function name, and will be followed immediately by the S-expression giving the function definition.

(2) *Numbers.* LISP 1.5 has a full complement of arithmetic facilities, although their use is still somewhat awkward because of the parenthesized prefix notation, e.g., "(TIMES A (PLUS B 5))" for "A*(B + 5)". Because of the problem of "data puns" alluded to in Section 1.1, the LISP 1.5 system has needed to adopt an artifice in order to deal with numbers. The most common artifice is the use of indirect storage, sometimes called *boxing*; a less common artifice is a combination of indirect storage for large numbers and direct storage for numbers whose magnitude is smaller than that of the lowest address usable for list structure storage. Indirect storage is, however, quite costly in both time and space, and one of the major drawbacks of LISP 1.5 has been its inefficiency in arithmetic calculations.

(3) *Sequential programs.* A major addition was the "program feature" to permit programs to be defined as a sequence of statements rather than in the purely functional form. A program is written in the form

$$(\text{PROG } (v_1\ v_2\ \ldots\ v_n)\ s_1\ s_2 \ldots s_m)$$

where the v_i are local variables and the s_i are statements. The local variables are assigned storage when evaluation of the PROG form commences, and this storage is released when evaluation is completed. Each statement is interpreted as an expression to be evaluated. The statements are evaluated in turn, and the values are then thrown away, so that the evaluation is always performed for the sake of its side effects. The expression

$$(\text{SETQ } x\ \alpha)$$

evaluates the expression α and assigns this value to the variable x.

The expression

(RETURN α)

terminates evaluation of the PROG form and causes the value of the PROG form to be the value of α. Labels, in the form of identifiers, may be intermixed with statements; evaluation of the expression

(GO x)

causes execution to continue with the statement following the label x. If a conditional expression is evaluated and none of the p_i are true, then control proceeds to the next statement, and the fact that the value is undefined does not (in this context) cause an error.

(4) *Compilation.* The original LISP system was interpreter-based. However, compilers have been added to several LISP 1.5 system. In some cases, the compiler has been used as a replacement for the interpreter; in others, the compiler and the interpreter coexist. Compilation appears to improve the running speed of LISP programs by a factor of about 50.

In LISP, the interpreter ordinarily exists as a function callable by the programmer. The function EVAL1 is the most useful form of the interpreter; given a symbolic expression, EVAL1 determines its value. Interpretation is required because the functions occurring in the expression to be evaluated must be applied to their arguments in an appropriate manner. In systems with a compiler only, an interesting approach has been taken to the implementation of EVAL1; namely, the expression to be evaluated is transformed into a function of no arguments, this function is then compiled, the compiled function is applied to its null set of arguments, and the result of this application is the desired value of the expression. The compiled program is then thrown away.

(5) *Character manipulation.* It was found desirable in many cases to manipulate data in an arbitrary format. For this purpose, character manipulation functions were provided in LISP. These functions permitted input or output of a character at a time, termination of an input or output line, designation of any character as an atom, formation of a sequence of characters into either an identifier or a number, and decomposition of either an identifier or a number into its component characters.

(6) *Macros.* Macros are included in many (but not all) versions of LISP 1.5. Each macro is associated with a particular identifier and consists of a transformation function. Let m be the name of a macro with transformation function f (which must be a function of one argument). Suppose that during evaluation of an expression a subexpression α is encountered whose first element (i.e., the element in the function posi-

tion) is m. Then the entire subexpression α is replaced by the result of applying f to α, and this new expression is then evaluated. Since the new expression may itself contain macro names, macro definitions may effectively be recursive.

Figure 7 gives the LISP 1.5 program for LCS that corresponds to the LISP 2 program for the same function as given in Fig. 6. The function GREATERP yields T if its first argument is numerically greater than its second argument, and F otherwise. The function SUB1 subtracts 1 from its argument. All the remaining functions and operators have already been explained.

4. L6

A highly machine-oriented (though moderately machine-independent) list processing language developed by Knowlton at Bell Telephone Laboratories in 1966 is L6 (Laboratories Low Level Linked List Language) [*19*]. In contrast to LISP, it gives the programmer very precise control over the allocation of storage, but at the cost of much more detailed programming.

In L6, the representation of data is determined entirely by the programmer. Storage is allocated in units called *blocks*; a block consists of 2^n machine words. In the 7094 implementation, n ranges from 1 to 7. Part of an L6 program consists of the definition of *fields*; field definitions may be changed dynamically as a program is run. A field is defined by designating its name (a single letter or digit), a word of a block, and a group of bits within that word. Fields may overlap or be contained within one another, and their contents are quite arbitrary. One possible content of a field is a pointer, which is the address of the 0th word of another block. The length of such a field must be greater than or equal to the address size of the machine

The programmer has available to him a set of 26 base fields, called *bugs*. These are designated by the letters of the alphabet, and constitute the explicit variables of L6. These variables may be operated upon by field names. Thus the sequence "W9WQAAR" refers to a field that is obtained by taking the block which is the value of the bug W, then taking the block pointed to by the 9 field within block W, then taking the block pointed to by the W field within block W9, etc. Note that W is used both as a bug and as a field, and these uses are independent. Note also that all the fields in the sequence except for R must be pointers.

The use of blocks, fields, pointers, and bugs is illustrated in Fig. 8. A pointer from a field to a block indicates that the field contains the location of the 0th word of the block. There are two 2-blocks and a 4-block in this diagram, and two bugs: T and R, where T refers to the leftmost

block and R refers to the rightmost block. The J field of the rightmost block may be referred to as RJ, TBJ, or TCBCJ (among the many possibilites). Note that the two leftmost blocks have the same division of their first two words into fields, but that the rightmost block has a

```
DEFINE((
(LCS (LAMBDA (L1 L2) (PROG (X Y BEST K N LX)
     (SETQ LX (LENGTH L1))
     (SEQT K 0)
     (SETQ X L1)
A1   (COND ((OR (NULL X) (NOT (GREATERP LX K)))
        (GO A4)))
     (SETQ LY (LENGTH L2))
     (SETQ Y L2)
A2   (COND ((OR (NULL Y) (NOT (GREATERP LY K)))
        (GO A3)))
     (SETQ N (COMSEGL X Y))
     (COND ((NOT (GREATERP N K)) (GO A)))
     (SETQ K N)
     (SETQ BEST (COMSEG X Y))
     (SETQ LY (SUB1 LY))
     (SETQ Y (CDR Y))
     (GO A2)
A3   (SETQ LX (SUB1 LX))
     (SETQ X (CDR X))
     (GO A1)
A4   (RETURN BEST)  )))
(COMSEGL (LAMBDA (X Y) (COND
     ((OR (NULL X) (NULL Y) (NOT (EQUAL (CAR X) (CAR Y)))) 0)
     (T (ADD1 (COMSEGL (CDR X) (CDR Y)))) )))
 (COMSEG (LAMBDA (X Y) (COND
      ((OR (NULL X) (NULL Y) (NOT (EQUAL (CAR X) (CAR Y)))) NIL)
      (T (CONS (CAR X) (COMSEG (CDR X) (CDR Y)))) )))
```

FIG. 7. LISP 1.5 program for LCS.

different division. Thus the B field occupies the same space as the J and K fields.

An L6 program consists of a sequence of macro calls. A macro call may contain elementary tests, elementary operations, and a label that specifies where control is to go when the operations are completed. A macro call may itself be labeled.

The elementary operations of L6 are concerned with setting up storage, defining fields, obtaining blocks of various sizes, freeing blocks that

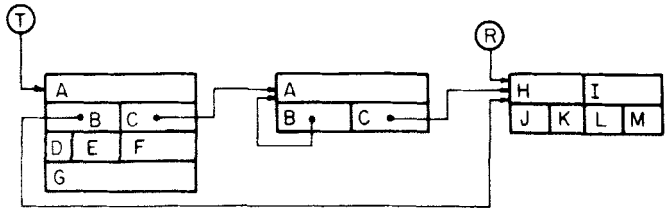


FIG. 8. A group of linked blocks.

are no longer needed, performing arithmetic and logical operations, doing input–output, pushing down and popping up certain fixed stacks, and calling subroutines. Each elementary operation is expressed as a parenthesized list of elements; the first element of the list ordinarily indicates the field affected by the operation, the second element specifies the operation itself, and the remaining elements are the operands.

For example, the operation

(5,DB,21,35)

defines field B to consist of bits 21 through 35 of word 5 of a block. The operation

(0,DX,PE,PF)

defines field X to consist of a group of bits in word 0. The starting bit is given by the contents of the E field of the block pointed to by bug P; the ending bit is the contents of the F field of the same block.

The operation

(CG,GT,8,D)

causes the storage allocator to get an 8-word block of storage, store its location in CG (i.e., the G field of bug C), and store the previous contents of CG in the D field of the newly obtained block. The rest of this block is 0. Had the operation been instead

(CG,GT,8)

then the new block would have its initial contents entirely 0. The more

elaborate form is useful in the creation of pushdown lists. The operator FR is used in order to free blocks; the decision as to when a block is to be freed must be made entirely by the programmer, and it is his responsibility to handle correctly such matters as the erasure of blocks that are pointed to by several other blocks.

An example of an arithmetic operation is

(V,A,5)

which adds 5 to the contents of bug V; another example is

(RY,M,QY)

which multiplies the contents of field RY by the contents of field QY, leaving the result in field RY. The operation

(C,XH,XYYZ)

replaces the contents of bug C by the exclusive OR of the old contents of C and the Hollerith literal "XYYZ".

As an illustration of the input–output operations, the operation

(A,IN,6)

brings 6 characters from the system input device into the contents of bug C. The characters are brought in via a left shift, so that existing characters in C are shifted off the left end. The operation

(C,PR,4)

prints 4 characters taken from the right end of C. The special character whose octal representation is 77 is used as an end-of-line signal; on input, no characters are brought in after a 77 is encountered, and on output, the transmission of a 77 causes the end of a line and the beginning of a new one.

The system provides two pushdown stacks to the programmer: the field contents pushdown and the field definition pushdown. A third stack, not visible to the programmer, is used for storing subroutine entries so that subroutines can be recursive. The operation

(S,FC,B)

saves the contents of B on the field contents pushdown (leaving the contents of B undisturbed), and the operation

(R,FC,B)

restores these contents. Similar operations exist for the field definition pushdown.

There are two kinds of tests: numerical and logical. The test

(BG,G,CG)

tests whether the contents of field BG are numerically greater than the contents of field CG. There are also tests for "less than," "equal," and "not equal." The test

(R,O,T)

tests whether the contents of R has a one bit in every position where the contents of T has a one bit; the command

(R,Z,T)

does the same for zero bits.

Instructions are made up from tests and operations. For example, the instruction:

L1 IFANY (GX,E,GY)(GW,N,GY) THEN (XR,E,3)(XS,SD,5) L5

has the label L1. It is interpreted to mean that if any of the conditions preceding the "THEN" are satisfied, the operations following the "THEN" are to be carried out, and control is then to go to the instruction labeled L5. Otherwise control goes to the next instruction. An instruction may omit any of the three parts, namely, the tests, the operations, and the go-to. An unconditional instruction thus starts with "THEN".

5. PL/I String and List Processing

The primary design aim of the programming language PL/I was to satisfy the requirements of commercial users, scientific users, and systems programmers simultaneously. Consequently, PL/I has borrowed heavily from FORTRAN, ALGOL, and COBOL; in addition, it includes many features that are found in none of these languages. Some of these features, such as the ability to specify parallel computations, were included in the original design; others, such as the list processing features, were added in subsequent revisions. Originally, PL/I was specified by a joint committee of IBM and SHARE, the IBM users' organization, and publicly released in March 1964; subsequent responsibility for the specifications was taken over by IBM. As of this writing, no computer manufacturer other than IBM has made PL/I available as a standard software package.

The list processing features of PL/I were included primarily to satisfy the needs of compiler writers, particularly those who were interested in "writing PL/I in PL/I." These features have in fact proved difficult to implement, and though they are included in the full language specification [*17*], they are not included in F-level PL/I [*18*], which is IBM's

present version. A description of PL/I list processing was published by Lawson [*20*]. The essence of the approach is to introduce pointers as a class of data and to provide facilities for referencing the data that they point to. No special functions for list processing are provided; thus, housekeeping responsibilities such as erasure are entirely the programmer's responsibility.

5.1 String Processing

Facilities exist in PL/I for processing both strings of bits and strings of characters; we shall consider here only strings of characters. A *character string* (henceforth usually referred to merely as a *string*) consists of a sequence of zero or more characters. Character string constants are written by enclosing them in primes. Primes within the string are indicated by two consecutive primes; repetition of a string can be indicated by a preceding repetition factor in parentheses. Thus,

'ABCDEF'

'IT' 'S ME'

(3)'CHA'

are all strings; the last of these is equivalent to 'CHACHACHA'.

In PL/I, variables are described by means of the DECLARE statement. Thus, the statement

```
DECLARE A FIXED, B(15,100) CHARACTER (30), C POINTER;
```

declares the variable A to represent a single fixed-point number, the variable B to represent a 15 by 100 array of 30-character strings, and the variable C to represent a pointer. A DECLARE statement may specify many different attributes of a variable; those not specified are determined either by the context in which the variable is used or by a default assumption.

Strings may be declared with either fixed or variable length, e.g.

```
DECLARE A CHARACTER (25), B CHARACTER (17) VARYING,
        C CHARACTER(*) VARYING;
```

In this case A is a string of exactly 25 characters, B is a variable-length string with a maximum of 17 characters, and C is a string whose length will be determined at the time storage is allocated for it. (Storage might be allocated either by using C as a formal parameter of a procedure or by using the ALLOCATE statement, described below.)

A collection of functions is provided for operating on strings. The operator "| |" is used to indicate the concatenation of two strings, i.e., the string consisting of all the characters of the first string followed

by all the characters of the second string. The function SUBSTR expects three arguments: a string s, an integer i, and an integer j. The value of SUBSTR consists of a sequence of j characters extracted from s beginning with the ith one, with appropriate definitions for exceptional cases. j may be omitted, in which case the entire string from the ith character onward is obtained.

The function INDEX expects two strings as arguments. If either of the two arguments is a null string, then INDEX returns 0. Otherwise it searches the first argument for an occurrence of the second argument as a substring. If such an occurrence is found, then the value of INDEX is the starting position within the first string of this substring. If such an occurrence is not found, INDEX returns 0. The function LENGTH expects a string as argument and returns as value the length of that string. The function REPEAT takes a string s and an integer n as arguments; its value is a string consisting of s repeated n times.

In addition to these functions, PL/I provides various methods for converting data of other types to and from character strings. Such conversions may be accomplished through assignments, through the use of an explicit conversion function, or through input–output functions that transmit external representations of data to character strings or take external representations of data from character strings.

Rosin [*33*] has proposed some interesting modifications to the PL/I string processing capability. He replaces SUBSTR(S,I,J) by S(: I ... J + I − 1), where the " ... " is actually part of the notation. He uses the following related auxiliary notations:

$$\text{X(: I)} \equiv \text{X(: I ... I)}$$

$$\text{X(: ... J)} \equiv \text{X(: 1 ... J)}$$

$$\text{X(: I ...)} \equiv \text{X(: I ... LENGTH(X))}$$

$$\text{X(A : I) ... J)} \equiv \text{SUBSTR(X(A), I, J − I + 1)}$$

(Actually, Rosin proposed the last of these as the basic notation, and defined the others in terms of it.) In the case where J < I, the resulting string consists of the characters from the Ith to the Jth in reverse order. Rosin also defines five new operators:

X UPTO Y returns X(: 1 ... INDEX(X,Y) + LENGTH(Y) − 1)

X BEFORE Y returns X(: ... INDEX(X,Y) − 1)

X AFTER Y returns X(: INDEX(X,Y) + LENGTH(Y) − 1 ...)

X FROM Y returns X(: INDEX(X,Y) ...)

Y IN X returns X(: INDEX(X,Y) ... INDEX(X,Y) + LENGTH(Y) − 1)

If, in any of these operations, Y does not occur in X, the scan is said to fail and a pseudo-variable is set to indicate this. There is some ambiguity in Rosin's proposal as to just what happens when a scan fails. He also proposes that the various string operators be permitted to appear on the left side of an assignment and cause modification of the string X when they appear in that context.

5.2 List Processing

The *storage class* of a PL/I variable determines the mechanism by which storage is assigned to it. There are four possible storage class attributes, as illustrated in the statement

```
DECLARE(A STATIC, B AUTOMATIC, C CONTROLLED,
        D BASED(P)) FIXED;
```

The parenthetical notation used here indicates that A, B, C, and D are all fixed-point variables. Storage for A is allocated exactly once, at the time when the program begins execution. Storage for B is allocated upon entrance to the block or procedure in which B is declared, and released upon exit from that block or procedure. Storage for C is allocated explicitly when the statement

```
ALLOCATE C
```

is executed, and freed when the statement

```
FREE C
```

is executed. A, B, and C are *nonbased variables*; D is a *based variable*. In this case, P is contextually declared as a pointer (it could, but need not, also be declared as a pointer explicitly); D serves as a prototype for the location that P points to. The expression

```
P->D
```

represents a fixed-point variable located at the address given by P. It is the programmer's responsibility to make sure that P actually points to a fixed-point variable. The symbol "->" is read as "qualifying," and in this example D is said to be *qualified by* the pointer P. If Q is another pointer, then Q->D would be the fixed-point variable pointed to by Q; P->D and Q->D may both exist at the same time, and may well be different. A reference to D by itself is taken to mean P->D, since P was the pointer declared with D. If the declaration

```
DECLARE D BASED FIXED;
```

had been used instead, then all references to D would need to be qualified explicitly.

The function ADDR, provided by PL/I, has as its single argument a variable name; its value is a pointer to that variable. Thus, if we write

```
DECLARE (A AUTOMATIC, B BASED(P)) FIXED;
P = ADDR(A);
B = 5;
```

the net effect will be to set the value of A to 5, since B really means the variable pointed to by P. Pointer qualifications may be nested, so that if we write

```
DECLARE P POINTER, Q BASED(R), A FLOAT,
  B FLOAT CONTROLLED(Q);
R = ADDR(P);
P = ADDR(A);
R -> Q -> B = 5.3;
```

then the net effect is to set A to 5.3. The rules of PL/I state that R -> Q -> B is to be interpreted as (R -> Q) -> B. To understand this example, note that R -> Q designates the pointer that is pointed to by R; that pointer is P. Qualifying B by R -> Q designates the floating variable pointed to by R -> Q, i.e., by P. Since P points to A, it is A that is set to 5.3.

A constant pointer NULL is provided, which can be used as a list terminator; NULL does not point to any data.

A *structure* is a hierarchical collection of variables, arrays, and substructures. The term *structure* as used in PL/I has no connection at all with list structures. A typical structure might be created by the statement

```
DECLARE 1 DATE, 2 YEAR FIXED(4), 2 MONTH FIXED (2),
  2 DAY, 3 DAY_OF_MONTH FIXED(2), 3 WEEKDAY CHAR (3);
```

The integers appearing in this declaration identify level numbers. A substructure of a given structure is indicated through the use of a higher level number. Thus the total structure (which is a variable) is DATE. A date consists of a 4-digit year, a 2-digit month, and a day. The day in turn consists of a 2-digit day of the month and a 3-character day of the week.

Suppose now that we wish to construct a one-way one-level list of

fixed-point numbers. Such a list can be organized using structures declared as follows:

```
DECLARE 1 ELEMENT BASED(P), 2 NEXT POINTER,
        2 CONTENT FIXED;
```

This declaration establishes the format of each list element, which consists of a fixed-point number and a pointer to the next element. A procedure for adding a number to the head of a list L and returning as value a pointer to the next list would be:

```
ADDNUM: PROCEDURE(L, N);
  DECLARE L POINTER, N FIXED, 1 ELEMENT BASED(P),
    2 NEXT POINTER, 2 CONTENT FIXED;
  ALLOCATE ELEMENT SET(P);
  NEXT = L;
  CONTENT = N;
  RETURN(P);
  END ADDNUM;
```

Initially, the value of L would be NULL. The ALLOCATE statement causes storage to be set aside to hold the structure ELEMENT, i.e., to hold a pointer and a fixed-point number. The "SET(P)" clause causes the pointer P to point to the beginning of this newly allocated storage area. The variable used in the SET clause need not be the same as the pointer declared with ELEMENT. Had the SET clause been "SET(R)" instead, then R would point to the newly allocated element, NEXT would have to be replaced by "R—>NEXT", and "CONTENT" would have to be replaced by "R—>CONTENT".

A procedure to erase a list of the kind created by ADDNUM would be

```
ERASE: PROCEDURE(L);
DECLARE L POINTER, 1 ELEMENT BASED(L),
  2 NEXT POINTER, 2 CONTENT FIXED, M POINTER;
DO WHILE L¬ = NULL;
M = NEXT;
FREE ELEMENT;
L = M;
END ERASE;
```

In this example, NEXT and ELEMENT are implicitly qualified by L. The END statement ends both the DO and the procedure. The symbol "¬" means NOT. The FREE statement has the effect of returning to the system the storage allocated for the element pointed to by the current value of L.

It is fairly clear how a two-way list could be created instead of a one-way list by using a different structure definition for ELEMENT. It is not quite so clear how a multilevel list or a list with nonhomogeneous elements could be created. In order to create such lists, we need to introduce yet another possible attribute of a variable, namely, CELL. CELL is used to specify storage equivalence between different data, and resembles the EQUIVALENCE statement of FORTRAN. Consider the declaration:

```
DECLARE 1 ELEMENT BASED(P), 2 NEXT POINTER,
        2 CONTENT CELL, 3 X FIXED, 3 Y FLOAT, 3 Z POINTER,
        2 TYPE FIXED(1);
```

By declaring CONTENT to have the attribute CELL, we specify storage equivalence among its substructures, namely, X, Y, and Z. In other words, the storage layout for an ELEMENT will permit the CONTENT part to be either a variable as described by X, a variable as described by Y, or a variable as described by Z. Any particular ELEMENT will have as its CONTENT just one of these alternatives. In this particular list organization TYPE is intended to be a one-digit code indicating which of the alternatives is the one actually present. Thus we can determine the type of a particular list element by testing the integer TYPE. Since CONTENT can be a pointer, we can have a list as an element of a list (the pointer merely need point to another ELEMENT); since CONTENT can also be a number, we can terminate a list at any level with a number.

From these examples it can be seen that PL/I does not impose any particular method of list organization upon the programmer; in this sense it resembles L6. It does, of course, require the programmer to specify how his lists are to be arranged, and this task is accomplished through the various declarations shown here.

6. SLIP

SLIP (for Symmetric LIst Processor) is a list processing language developed by Weizenbaum in 1963 [*40*]; an excellent updated description has been published by Smith [*37*]. SLIP is a descendant of at least four earlier languages: Gelernter's FLPL [*13*], IPL-V [*28*], Perlis's threaded list system [*30*], and Weizenbaum's earlier KLS system. SLIP actually consists of a collection of subroutines to be used by a

FORTRAN program; most of these subroutines, being themselves written in FORTRAN, are machine-independent. Thus SLIP provides to its users several advantages at the outset. Since SLIP is embedded in FORTRAN, the SLIP user has the full facilities of FORTRAN, and in particular its numerical and array-handling facilities, available to him. If he already knows FORTRAN, then the burden of learning SLIP is eased. SLIP programs are essentially as transferable from one machine to another as are FORTRAN programs; and from the implementer's viewpoint, SLIP is quite easy to install on a new machine with a FORTRAN compiler. FORTRAN, by the way, is not the only host language that has been used for SLIP; a MAD version also exists at Yale University.

6.1 SLIP Data

SLIP data, which are in addition to the usual FORTRAN data, consist of two-way lists. SLIP lists can be traversed either forward (i.e., left to right) or backward (i.e., right to left); hence, the name *symmetric lists*. The general form of a SLIP cell is shown in Fig. 9. The cell actually

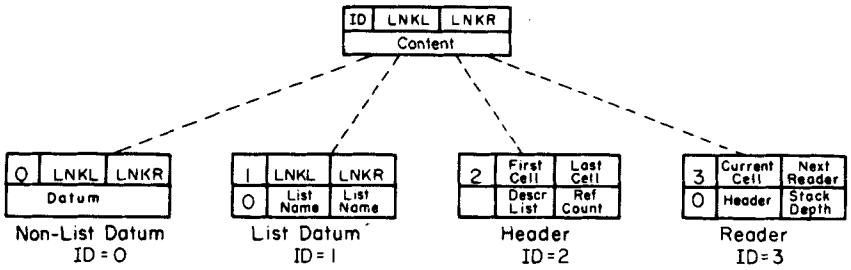


FIG. 9. The SLIP cell.

consists of two words in most computers. The first word always contains three fields, called ID, LNKL (link left), and LNKR (link right). If the ID is not 0, the second word is also subdivided in the same way. The *name* of a list is a word whose ID is zero and whose LNKL and LNKR fields both contain pointers to, i.e., the address of, the list. A FORTRAN variable whose value is a list will normally contain the name of that list.

The ID field of the first word of a cell determines how the cell is to be interpreted. A cell representing a list item will have an ID of 1 if the item is a sublist and an ID of 0 otherwise. For sublists, the second word contains the name of the sublist. Every list has a header that serves as a starting point for the list and also as a way of referring to it; the ID of a header is 2. The ID field of the second word of a header can be used by the user for any purpose; the LNKL field of that word contains a pointer to a description list (of Section 6.2.5) if one is desired, and the

LNKR field contains a reference count, discussed below. Readers, which are used for traversing lists, have an ID of 3.

An example of a SLIP multilevel list is given in Fig. 10. For each list, the header indicates the first and last cells of the list. The LNKL field of a list cell points to the cell to the left of the given one; the LNKR field of a cell points to the cell to the right of the given one. The header is the left neighbor of the leftmost cell and the right neighbor of the rightmost

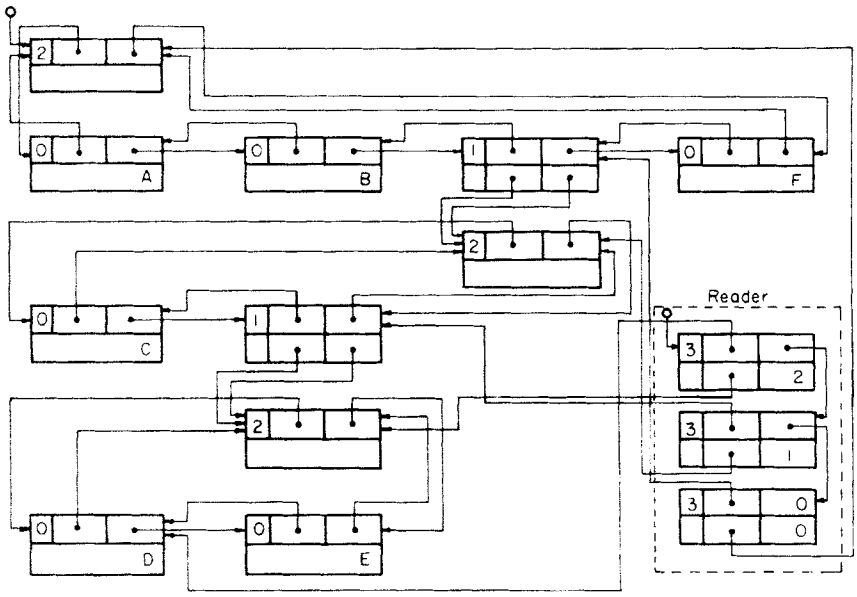


FIG. 10. SLIP representation of (A B (C (D E)) F) with reader at D.

cell; for an empty list, the header is its own left and right neighbor, and LNKL and LNKR of the first header word both point back to the header itself.

Unlike the languages discussed so far, SLIP does not treat segments of lists as lists. In other words, CDR of a SLIP list is not a SLIP list; one would need to copy it in order to make it into a SLIP list. This situation is a necessary consequence of SLIP's two-way linkages; if two lists were to share the same CDR, then the left neighbor of the CDR could not be uniquely defined. The main consequence of this restriction is that part of one list cannot appear as a sublist of another list.

Readers are a device for traversing lists; their use is discussed in Section 6.2.4. From the storage standpoint, a reader is a list of cells that indicates a particular item embedded within a list (not necessarily on the top level) and the path that leads from the list header to this item. Figure 10 includes a reader of the list shown there. The first cell of the

reader points to the item. The successive cells of the reader point to the successive higher level lists that contain the item. A reader cell contains a pointer to a list item in LNKL of its first word and a pointer to the head of the corresponding list in LNKL of its second word; LNKR of the first word contains a pointer to the next higher cell of the reader (0 for the top one) and LNKR of the second word contains the level number (0 for the outermost list, increased by 1 for each level of nesting). It can be seen, then, that if a reader points to an item embedded in a list, then we can trace our way back to the outermost list and can also tell how deep we are within that list.

One of the notable and original contributions of SLIP lies in its approach to storage management and recovery. SLIP divides the responsibility for erasure between the user and the system. In general, a list can be erased if and only if there are no references to it either as the value of a variable that will be used later on or as an element of a higher level list. SLIP gives the user responsibility for external references, i.e., those that occur as values of FORTRAN variables, and takes upon itself the responsibility for internal references, i.e., those that occur as parts of other (not erased) lists.

Storage management is implemented through the use of a reference count contained in the header of every list. The reference count for a list is either the number of internal references to it or the number of internal references to it plus one, depending on how the user created it. Whenever a list α is inserted as a sublist of a list β, then the reference count of α is increased by one. Whenever a list is explicitly released by the user, its reference count is decreased by one. When the reference count of a list becomes zero, that list can be erased. A list is erased by appending it to the List of Available Space (LAVS) from which new cells are obtained. References to sublists of an erased list are dealt with by the procedure that makes cells available from LAVS. When a cell is taken from LAVS, a check is made to see if the ID of the cell is 1. If the ID is not 1, the cell is simply made available. If the ID is 1, the second word of the cell must contain the name of a list. The reference count of this list is then decreased by one. If as a result the reference count becomes zero, then this list is in turn added to LAVS.

When a new list is created or cells are added to an existing list, the cells required are taken from LAVS according to the procedure just described. The user may, at his option, set the initial reference count of a newly created list to one or zero. If he sets the initial reference count to one, then the list will never be returned to LAVS until he releases it explicitly (by decreasing its reference count by one). If he sets the initial reference count to zero, then the list will be erased as soon as the last internal reference to it is erased.

This scheme has several pleasant consequences. The user has responsibility for storage control for those references to lists that are most visible to him, namely, the external ones, while being freed of the responsibility for those references that are much less visible to him, namely, the internal ones. Since appending a list to LAVS at the time of erasure requires examining only the two ends of the list, the time required to erase a list is independent of its length. Also, the bookkeeping for erasing sublists is postponed until the last possible moment.

6.2 SLIP Programs

The SLIP system exists in the form of a large number of subroutines. We shall not attempt to describe them all here, but rather we shall discuss their major groupings and give some illustrations of the members of each group. The grouping used here follows Smith.

6.2.1 Storage Allocation

The two routines normally used by the programmer are INITAS (*space*, *n*), which converts the array *space* of dimension *n* into the the initial LAVS, and IRALST(l), which decreases the reference count of the list l by one. The routine NUCELL, which obtains a new cell from LAVS, is normally called only by the SLIP subroutines themselves; its main relevance for the user is that if no space is available, it will issue an error complaint and terminate the entire program.

6.2.2 Manipulating Data on Lists

These routines add, delete, change, and reference data on lists. The routine LIST(l) creates an empty list; if its argument is the literal "9", the reference count is initially zero. Otherwise the reference count is initially one, and the name of the created list is placed in l. In either case, the value of LIST is the name of the created list. NEWTOP(d,l) inserts the datum d as the first element of l, and NEWBOT(d,l) inserts d as the last element of l. NXTLFT(d,c) inserts the datum d to the left of the cell c, and INLSTL(l,c) inserts the list l to the left of the cell c; NXTRGT and INLSTR, which insert on the right, are analogous. LSSCPY(l) creates a new copy of the list l, and has as its value the name of this new copy. The value of TOP(l) is the datum stored as the first item in the list l; the value of BOT(l) is the datum stored as the last item of l. DELETE(c) deletes the item c from the list containing it. Note that since a SLIP cell can be a member of only one list, its left and right neighbors are uniquely defined even if the header of the list is not given.

As an example, the following program will create a list whose elements consist of the integers from 1 to 10:

```
      CALL LIST(ILIST)
      DO 1 K = 1,10
    1 CALL NEWBOT(I,ILIST)
```

The value of ILIST will be the name of this list.

6.2.3 *Sequencing through Lists*

In order to simplify sequential processing of the items on a list, SLIP provides a collection of sequencing functions. The function SEQRDR(l) has as its value a newly created *sequencer* for the list l; the sequencer is initially a pointer to the header of l. The function SEQLL(s,f) advances the sequencer s to the next cell to the left of the current one, i.e., causes the sequencer to point to that cell. Its value is the datum stored in the new cell, and as a side effect it sets f to $+1$ if the new cell is a header, 0 if the new cell is a list name (i.e., has an ID of 1), and -1 if the new cell is a nonlist datum. The analogous function for sequencing to the right is SEQLR. As an example, assume that the list LL contains a mixture of floating point numbers and sublists. The following program will set SUM to the sum of the floating point numbers:

```
      SL = SEQRDR(LL)
      SUM = 0
    1 X = SEQLL(SL,F)
      IF (F) 2,1,3
    2 SUM = SUM + X
      GO TO 1
    3 CONTINUE
```

6.2.4 *Tracing through Lists Using Readers*

By means of *readers*, the user can operate on or examine the elements of a list (and also those of its sublists) in a more elaborate way than he can using a simple sequencer. We define the *progeny* of a list to consist of its elements plus the progeny of its sublists. A reader of a list l then consists of a pointer to α, one of the progeny of l, together with the path from α to the header of l. The internal representation of a reader was discussed in Section 6.1. A reader of a list l is created by calling LRDROV(l);

the value of this function is the name of the newly created reader, which points to the header of *l*. There are 12 functions for advancing readers. A reader may advance to the left or the right; it may advance structurally (descending into sublists) or linearly (not descending into sublists); and it may advance to the next element, the next name, or the next word (where an element has an ID of 0, a name has an ID of 1, and a word has an ID of either 0 or 1). Thus ADVSWR(*r*,*f*), for instance, will ADVance Structurally Word Right. Here *r* is the name of the reader, and *f* is a flag. If the present cell contains the name of a sublist, then the advance will take the reader to the rightmost cell of that sublist; otherwise the advance will take the reader to the cell directly to the right of the present one. In this case, since the advance is to the next word, any item other than a header will be acceptable; had the advance been by element, then any name found would be skipped (though descent into the sublist designated by the name would still take place). The flag *f* is made zero if a cell of the given type is found, and nonzero if the search for such a cell encounters the header of *l*.

As an example, assume that the terminal nodes of the list LL (i.e., those members of the progeny of LL that are not themselves sublists) are all floating-point numbers. Then the following program will compute the sum of the terminal nodes of LL:

```
      K = LRDROV(LL)
1     X = ADVSER(K,F)
      IF (F.NE.0) GO TO 2
      SUM = SUM + X
      GO TO 1
2     CONTINUE
```

6.2.5 *Description Lists*

A *description list* is a sequence of linked cells that can be attached to the header of a list and contains information describing that list. The description list is composed of pairs of cells; the first cell of a pair contains an *attribute* and the second contains a *value*. Description lists in SLIP are thus like property lists in LISP, except that property lists are attached to elementary items (i.e., identifiers) while description lists are attached to lists.

Attribute-value pairs are added to a description list by means of the function NEWVAL; NEWVAL(*at*, *val*,*l*) searches the description list of the list *l* for the attribute *at*. If *at* is found, then the corresponding value is

replaced by *val*, and the value of NEWVAL is the replaced value. If *at* is not found, then the (*at*, *val*) pair is added to the bottom of the description list and the value of NEWVAL is zero. Pairs are removed from a description list by NOATVL(*at*,*l*), which removes *at* and its associated value from the description list of *l*. The function ITSVAL(*at*,*l*) has as its value the value paired with *at* on the description list of *l*. Other functions exist for copying, removing, and performing other manipulations on description lists.

6.2.6 Recursion

Since the FORTRAN language as usually defined and implemented does not allow recursion, SLIP has provided special functions to make recursion possible. A recursive function is normally written as a block of code beginning with a statement label rather than as a FORTRAN function. The block is entered by a FORTRAN ASSIGN statement and a call to the function VISIT; it is left by a call to the function TERM. For example, executing the statements

```
ASSIGN 60 TO LOC
X = VISIT(LOC)
```

will cause the recursive function defined by the block of code beginning at statement 60 to be entered; the value of that function will be returned as the value of VISIT and thus will become the value of the variable X. Execution of the block will be terminated by

```
CALL TERM(Z)
```

which will have the dual effect of returning control to the place from which the corresponding VISIT was called, and setting the value of VISIT to Z. It should be noted that the call to VISIT actually causes a transfer of control to LOC, and that control does not return to VISIT until TERM is called.

In order to save and restore arguments of recursive functions, SLIP uses an array of 100 public lists, placed in COMMON storage and designated W(1), ..., W(100). Upon entrance to a recursive function, these lists are pushed down; upon exit, they are popped up. Pushdown is done by the function PARMTN($p_1, p_2, \ldots, p_n$) which expects a variable number of arguments and saves these arguments on the first n public lists. The pushdown for a particular argument is done by placing the argument into the second word of a cell obtained from LAVS and adding that cell to the head of the corresponding public list. Popping up is done by RESTOR(n), which removes the first cell from each of the public lists, W(1) through W(n).

In order to simplify saving and restoring of arguments, VISIT and TERM are both permitted to accept a second argument. This argument will be evaluated during the function call, but the value obtained will be discarded. If PARMTN is called as the second argument of VISIT, then the FORTRAN statement that enters a recursive function can also save the arguments of that function; similarly, if RESTOR is called as the second argument of TERM, then the FORTRAN statement that leaves a recursive function can also restore the arguments of that function. In those versions of FORTRAN that do not permit a variable number of arguments in subroutine calls, the treatment of VISIT, TERM, and PARMTN will be slightly different.

6.2.7 *Input–Output*

Weizenbaum's original article on SLIP gave no information on input–output, though Weizenbaum's system does in fact contain functions for that purpose. Smith describes two functions RDLSTA and PRLSTS for reading and printing lists. RDLSTA reads lists in essentially the same format as LISP, except that the dot notation is not meaningful and numbers are treated as character strings. PRLSTS prints lists in quite a different format; no parentheses are used, one item appears on each line, and the beginning and end of a sublist is indicated by indentation and a special message. Thus the list (A B (C D) E) would print as

```
BEGIN LIST
A
B
        BEGIN SUBLIST
        C
        D
        END SUBLIST
E
END LIST
```

7. SNOBOL

SNOBOL[3] is a programming language designed primarily for the manipulation of strings. Historically, SNOBOL was inspired primarily by COMIT; the first version of SNOBOL was described by Farber *et al.* in 1964 [*11*]. There have been several subsequent modifications, of which the best known has been SNOBOL3 [*12*], completed in 1966. A more

[3] The meaning of the acronym "SNOBOL" has never been explained publicly.

recent version, SNOBOL4 [*14*], is presently being implemented and is gradually replacing SNOBOL3. The description given here is based on SNOBOL3, but includes a section on the changes introduced by SNOBOL4.

It is interesting to observe that although SNOBOL is a string processing language rather than a list processing language, many of its applications are the same as those of list processing languages. For instance, Farber *et al.* [*12*] give three examples of the use of SNOBOL. Of these, two, namely, the Wang algorithm for the propositional calculus and a program for the differentiation of algebraic expressions, are classic examples used to illustrate LISP. (The third example is an editor that does right justification of printed lines.)

The structure of a SNOBOL3 program is quite different from that of a program in any of the languages we have discussed so far. A SNOBOL3 program consists of a sequence of statements, of which there is only one type. Variations are achieved through omission of components of the statement. The statements are executed sequentially until the sequence is altered by a go-to. In general, execution of a statement causes all or a portion of a string to be replaced by a different string. However, many different kinds of side effects can also occur as a result of the execution.

The data of SNOBOL3 are all strings; a constant string is represented by enclosing it in quotes. Numbers, too, are treated as strings; they also must be enclosed in quotes, though arithmetic is possible. The variables of SNOBOL3 are called *names*, and they all have strings as values.

One of the simplest subcases of a SNOBOL3 rule is a pure assignment such as

CAT = " SIAMESE "

which causes the value of the name CAT to be the string SIAMESE. Concatenation of strings is represented by writing them successively; thus, the two statements

PREP = " OF ",

SHIP = " MAN " PREP " WAR "

will cause the value of SHIP to be the string " MAN OF WAR ". Note that in this case we have concatenated a constant. a name, and another constant. The statement

STRING =

will cause the name STRING to assume the null string as its value. Quotes cannot be written as part of a constant. However, the name QUOTE has as its initial value a single quote. Thus strings containing quotes can be created and manipulated.

The most general form of assignment causes just a portion of a string to be replaced. Thus, if the value of STRING is "CANDY IS DANDY" and we write

STRING "ANDY" = "HOCOLATE"

then the new value of STRING will be

"CHOCOLATE IS DANDY"

The first item on the left side of the assignment (ignoring labels for the moment) is the *string reference*, which specifies the string to be modified; the remaining items (in this case only one) specify the portion of the string reference that is to be replaced and are called the *pattern*. If there is no pattern, as in the simple assignment given earlier, then the entire string as specified in the string reference is replaced. Note that only the first instance of the pattern within the string reference is affected. The pattern may or may not be found within the string reference. If it is found, the statement is said to *succeed*; if it is not found the statement is said to *fail*.

A statement containing all the possible components is

L1 STRING S1 "OF" S2 = S2 "FOR" S3 /S(L1)F(L5)

Here L1 is the statement label, which must begin in column 1; if the statement is unlabeled, then column 1 will be blank. The components of the statement are separated by blanks. A period in column 1 indicates a continuation card. STRING is the string reference, S1 "OF" S2 is the pattern, S2 "FOR" S3 is the *replacement*, and the material following the slash is the go-to.

Execution of this statement proceeds as follows: S1 and S2 are both names whose values are specific strings. The pattern being searched for consists of S1 (more precisely, the value of S1) concatenated with "OF" concatenated with S2. If this pattern is found within STRING, then the first occurrence of it is replaced by S2 concatenated with "FOR" concatenated with S3. Assuming the pattern is found, control returns to statement L1, as specified by the S (success) alternative of the go-to; if the pattern is not found, control goes to statement L5 as specified by the F (failure) alternative of the go-to. If a statement has no S alternative specified in the go-to, then control will pass to the next statement if execution succeeds; lack of an F alternative is treated analogously. Because the statement in this example loops back to itself if the pattern is found, the effect of executing it will be to change every occurrence of the pattern to the replacement and then to transfer control to L5. Note that in this case both the pattern and the replacement contain names.

If a statement does not contain an "=", then the statement will still

succeed or fail according as the pattern is or is not found within the string reference, but no replacement will be done. Statements of this sort are useful for testing. Thus,

STRING "W" /S(L4)

will transfer control to statement L4 if STRING contains a "W" and will pass control to the next statement otherwise.

The usual arithmetic operators are available in SNOBOL3, though only integer arithmetic is permitted. Numeric constants must be quoted (which is a nuisance), operators must be separated from their operands by blanks, and expressions with more than one operator must be fully parenthesized (also a nuisance). For SNOBOL3, an integer is defined to be a character string that represents an integer. Parenthesized arithmetic expressions may appear as part of a pattern. Thus,

C = A + (B * "4")

multiplies B by 4, adds A, and leaves the result in C. If the value of N is "21", then

STRING "W" (N − "8") = "Y12"

will replace the first occurrence of "W13" in STRING by "Y12".

Strings may be searched for patterns that are not entirely known in advance by means of *string variables*. An *arbitrary string variable* is designated by surrounding a name by asterisks. It will match any string whatsoever (including the null string). Thus, if we write

STRING = "PETER PIPER PICKED A PECK OF PICKLED PEPPERS"

STRING "PIPER" *ACTION* "A PECK"

then *ACTION* will match "PICKED". Moreover, the name ACTION will be assigned "PICKED" as its value. A name used this way can appear later in the statement, either as part of the pattern or as part of the replacement or both. Thus, if we write

STRING1 = "A ROSE IS A ROSE IS A ROSE"

STRING2 = "A HORSE IS A HOUSE IS A HOSS"

STRING1 "A" *NOUN* "IS A" NOUN "IS A" NOUN

= "A" NOUN "IS SURELY A" NOUN

STRING2 "A" *NOUN* "IS A" NOUN "IS A" NOUN

= "A" NOUN "IS SURELY A" NOUN

then replacement will occur for STRING1 but not for STRING2. The value

of NOUN, however, will be "ROSE" rather than "HORSE" as a consequence of the failure of the fourth statement.

A pattern may contain fixed-length string variables and balanced string variables. A *fixed-length string variable* is written by following the name by a slash and a string specifying a length in characters. In the statement

```
STRING "A" *PART/"7"*
```

the seven characters of STRING following the first "A" will be named PART. The "7" could be replaced by any string expression evaluating out to an integer. A *balanced string variable* is indicated by surrounding the name by parentheses; the variable will only match a string that is balanced in the usual algebraic sense. Thus, if we write

```
EXPR = "A + (B**C)* C"

EXPR "A" *(STR)* "C"

EXPR "A" *STR1* "C"
```

then the value of STR will be "+(B**C)*", while the value of STR1 will be "+(B**".

Both system-defined and user-defined functions are available. An example of a system-defined function is SIZE(s), whose value is the number of characters in the string s. A function may, instead of returning a value, signal failure; the failure signal will cause the statement containing the function call to fail. This feature is useful for testing. For example, the function EQUALS (x, y) returns the null string if x and y are identical character strings, and signals failure otherwise; thus,

```
N = EQUALS(SX, SY) N + "1"
```

will increment N by 1 if and only if SX and SY are the same character string. In interpreting an expression containing both concatenation and arithmetic operations, the concatenations are done first. In this example, if SX and SY are identical, the null string will be concatenated with N, which leaves N unaffected; then 1 will be added to N. If SX and SY are not identical, EQUALS will signal failure and no replacement will be done. Various arithmetic tests use the same mechanisms. Thus .LT(x, y) returns the null string if $x < y$ and signals failure otherwise. The arithmetic test for equality is not quite the same as EQUALS, since .EQ("0069", "69") will succeed while EQUALS ("0069", "69") will fail.

Often it is necessary to match a pattern at the beginning of a string rather than at some arbitrary place in the middle. Executing the function call

```
MODE("ANCHOR")
```

will cause all subsequent pattern matches to start at the beginning of the string being matched. Executing

```
MODE("UNANCHOR")
```

will cause the usual mode of pattern matching to be resumed. The value of both of these function calls is the null string.

A user may define a function by a call to DEFINE. The function definition consists of a block of code. Thus,

```
DEFINE("REVERSE(X)","REV")
```

defines a function named REVERSE; its formal parameter is X, and the defining code block begins at the statement labeled REV. The value returned by REVERSE will be the character string X in reverse order. The defining block might be

```
REV X   *CHAR/1* =  /F(RETURN)

        REVERSE = CHAR REVERSE /(REV)
```

Here the second statement transfers control unconditionally to REV. The special label RETURN is used to indicate return from the function. When the function is entered, the value of the name REVERSE will be the null string. The value returned by the function will be the value of the name REVERSE at the time of the return transfer. Thus

```
Z = REVERSE("ABCDE")
```

sets the value of Z to "EDCBA". The second argument of DEFINE may be omitted, in which case it is taken to be the same as the name of the function.

The character "$" is used to indicate indirect references. Thus if the value of the name AUTHOR is "MELVILLE" and the value of the name MELVILLE is "MOBY DICK", then the value of $AUTHOR will be "MOBY DICK". More complicated cases are possible; thus, the statements

```
WORD *CH/"1"*

$("LIST" CH) = WORD " " $("LIST" CH)
```

will add WORD to one of LISTA, LISTB, ..., LISTZ according to what the first character of WORD is. Indirect references can be used in the go-to as well as in the pattern or replacement.

Input and output are accomplished through the use of the special names SYSPIT and SYSPOT.[4] Every time that a value is assigned to

[4] According to the usage at Bell Laboratories in SNOBOL3 days, SYSPIT stands for System Peripheral Input Tape and SYSPOT stands for System Peripheral Output Tape.

SYSPOT, the value so assigned is printed as a new line in the standard output file. Every time that SYSPIT is evaluated, a new line is read from the standard input file and that line is the required value. Thus,

SYSPOT = SYSPIT

will cause a line to be copied from the input file to the output file.

The implementation of SNOBOL3 involves both a compiler and an interpreter. The compiler translates the original source program into an internal language suitable for the interpreter; the interpreter operates on this internal representation at run time. Storage recovery is completely automatic.

7.1 SNOBOL4

SNOBOL4 is a significant improvement over SNOBOL3. Most of the changes have been in the direction of generalizing existing SNOBOL3 concepts; the remaining changes are primarily concerned with eliminating nuisances.

One nuisance that has been eliminated is the requirement that integers be enclosed in quotes. Also, sequences of arithmetic operators need not be fully parenthesized, so SNOBOL4 will interpret correctly the statement

C = A + B * D

In SNOBOL4, more than one statement may be written on a line; successive statements are separated by semicolons. There are also a number of minor syntactic changes.

The concept of a pattern is greatly generalized in SNOBOL4 over what it is in SNOBOL3. In SNOBOL4, a pattern is a type of datum, and a name can designate a pattern. Patterns may be composed from simpler ones in a number of ways. Thus,

OPER = "+" | "−" | "*" | "/"

creates a pattern (not a string) named OPER; this pattern will be matched by any of the arithmetic operators. Patterns may be concatenated, so that

DOPER = "." OPER

would associate with DOPER a pattern consisting of a period followed by an arithmetic operator. The string variables of SNOBOL3 are replaced by patterns. Thus ARB replaces an arbitrary string variable, BAL replaces a balanced string variable, and LEN(n) replaces a string variable of length n. For instance,

"A" ARB "B"

is a pattern that will match any string that starts with an "A" and ends with a "B". Also,

"SIN" BAL

will match any string that consists of "SIN" followed by a parenthesis-balanced string of characters, and

"A" LEN(4) "BC"

will match a string consisting of "A" followed by four arbitrary characters followed by "BC".

Since in SNOBOL3 the matching of a string variable can be used to assign a value to a name, SNOBOL4 requires a corresponding facility. Two operators are used for this purpose: "." and "$". If we write

STRING1 = "THREE BLIND MICE"

STRING1 " " ARB . ADJ " "

then ADJ will be assigned the value "BLIND" since ARB will match "BLIND" and the period will cause the value of ARB to be assigned to ADJ. Any component of a pattern may be named by appending a value assignment to it in this way. Since a component may consist of several subcomponents enclosed in parentheses, groups of components may also be named.

If value assignment is done by ".", then the assignment will be made after the entire pattern has been matched and not before. Consequently the "." will not work for back referencing, i.e., for matching a component named earlier in a pattern. Assignment made by "$", on the other hand, takes place immediately, whether or not any further matching is successful. Thus,

STRING = "X + ALPHA = ALPHA"

STRING "X+" ARB $ VAR "=" VAR

will cause VAR to match "ALPHA" and the entire pattern to match STRING. Had STRING been "X+ALPHA=BETA", VAR would still have received the value ALPHA even though the entire pattern match would have failed. A "." used in place of "$" would cause the pattern match to fail and would leave the previous value of ARB undisturbed.

Ordinarily, all the components of a pattern are evaluated before any pattern matching takes place. However, a "*" may be used preceding a pattern component to indicate that the component is not to be evaluated until it is needed in a match. This means of pattern evaluation

is known as *deferred pattern definition*, and it permits the definition in a simple way of recursive patterns. Thus, the pattern

P = "B" | "A" *P

will match any of

B
AB
AAB
...

The ANCHOR and UNANCHOR modes of SNOBOL3 are replaced by a more general method in SNOBOL4. The pattern-valued function POS(n) matches a null string n characters from the beginning of the string being matched. In particular, POS(0) at the beginning of a pattern will force the rest of that pattern to be matched against the beginning of the target string. A similar function RPOS(n) matches a null string n characters from the end of the target string. Among the other useful pattern-valued functions are ARBNO(p), which matches an arbitrary number (including zero) occurrences of the pattern p; ANY(s), which matches any character in the string s; and BREAK(s), which will match a nonnull sequence of characters containing no character in s but followed by a character in s.

SNOBOL4 includes arrays as a type of datum. An array is created by calling the function ARRAY(d, v), where d specifies the dimensionality of the array and v, which may be omitted, specifies the initial value to be assigned to the array elements. Thus,

BOARD = ARRAY("8,8", "X")

causes the value of BOARD to be an 8 by 8 array, each element of which is initially the string "X". An array element is referenced by enclosing the subscripts in angle brackets; e.g.,

BOARD < 3,5 >

In order to permit more flexible calling sequences, SNOBOL4 includes a name operator. If a name (which in general is anything that denotes an object) is prefixed by the operator ".", then the resulting object indicates the name of the object so prefixed rather than the object itself. With this mechanism, one can for instance write functions that pass results back through their arguments (a frequently used device in FORTRAN programming).

All SNOBOL programs, being character strings, are themselves in the form of SNOBOL data. In SNOBOL4, the user can take advantage of this fact, much as he can in LISP, by creating a program and then executing

it. The mechanism for accomplishing this is the function CONVERT (p,"CODE") where p is a character string representing a sequence of SNOBOL4 statements. The statements are separated by semicolons within the character string. Evaluation of CONVERT with "CODE" as its second argument causes p to be compiled, and returns as value a data object of type CODE that represents the compiled code. This code can then be executed either by transferring to it directly using a special kind of go-to or by transferring to a label within it in the ordinary way.

SNOBOL4 uses data types, but not in the way that the other languages discussed here use them. The user can ascertain the data type of any object, but he need not (and indeed cannot) declare that a given name will always have as its value an object of a given type.[5] The principal use of the data-type predicates is in writing functions whose behavior depends on the type of their arguments. There is also a facility for creating new data types and then using them; these new data types resemble the ntuples of LISP 2 and the structures of PL/I (sans level numbers).

Figure 11 gives a SNOBOL4 program analogous to the LISP function LCS given previously. Since the operations being performed are essentially linear, this program is somewhat simpler than the LISP program. The program defines the function LCS for computing the longest common substring of two strings, and then applies LCS to two strings on successive cards in the input data stream. The result is then printed out. The LISP functions COMSEG and COMSEGL are combined into a single function COMSEGL(x,y) that returns a length as value and sets L3 to the initial common segment of x and y. Note that tests are used in several of the replacements. If such a test fails, the corresponding replacement is not done; if such a test succeeds, it generates a null string that can be concatenated with the replacement without changing it. The system function TRIM removes trailing blanks from its argument.

8. Other Symbol Manipulation Languages

The previous sections have discussed those symbol manipulation languages that in the author's opinion are the most significant ones at this time. In this section we shall examine briefly a number of other languages, though even the list given here is far from exhaustive and reflects to a great extent the author's own biases.

[5] Compare this situation with LISP 2 and PL/I, where all variable names have fixed types associated with them either by explicit or implicit declaration. The lack of declarations adversely affects the efficiency of SNOBOL4 exactly in the way that it adversely affects the efficiency of LISP 1.5 versus that of LISP 2.

```
      DEFINE ("LCS(L1,L2)")
      DEFINE("COMSEGL(X,Y)")
           S1 = TRIM(INPUT)
           S2 = TRIM(INPUT)
           OUTPUT = LCS(S1,S2)                              :(END)
      * LCS COMPUTES THE LONGEST COMMON SEGMENT OF TWO STRINGS L1
      * AND L2
           K = 0
           X = L1
      A1   DIFFER(X,"") GT(SIZE(X),K)                       :F(RETURN)
           Y = L2
      A2   DIFFER(Y,"") GT(SIZE(Y),K)                       :F(A3)
           N = COMSEGL(X,Y)
           K = GT(N,K) N                                    :F(A)
           LCS = L3
      A    Y POS(0) LEN(1) =                                :(A2)
      A3   X POS(0) LEN(1) =                                :(A1)
      * COMSEGL(X,Y) RETURNS THE LENGTH OF THE LONGEST INITIAL
      * COMMON SUBSEGMENT OF X AND Y AND AS A SIDE EFFECT SETS
      * L3 TO THAT COMMON SUBSEGMENT
           COMSEGL = 0
      B1   X LEN(COMSEGL) . X1                              :F(RETURN)
           Y LEN(COMSEGL) . Y1                              :F(RETURN)
           COMSEGL = IDENT(X1,Y1) COMSEGL + 1               :F(RETURN)
           L3 = X1                                          :(B1)
```

FIG. 11. Definition of LCS in SNOBOL4.

8.1 IPL-V

The most recent version of IPL[6] is IPL-V [28]. Historically, IPL is of great importance, having pioneered the use of list processing techniques. The original IPL was developed by Newell *et al.* in 1957 [27] for use in connection with their explorations of problem-solving. It was implemented on the JOHNNIAC computer at the Rand Corporation and

[6] For "Information Processing Language."

also on the IBM 650, and to this day much of the format of the language results from the characteristics of the IBM 650 and its assembly program. Although IPL still has its adherents, it appears to have been superseded by the newer languages.

Like L6 programs, IPL programs are heavily oriented toward machine-language programming. An IPL program is divided into routines, each of which consists of a set of instructions. The routines can themselves be expressed as lists of instructions, and the instructions in turn can be expressed as data. Thus, IPL programs are self-descriptive in the same way that LISP and SNOBOL programs are.

Symbols are used to designate *storage cells*. A storage cell is capable of holding a datum such as a list name, a number, or an alphanumeric string; a pushdown stack is implicitly associated with every storage cell. The instructions make implicit use of two special storage cells: H0, the *communication cell*, and H5, the *test cell*. The H0 cell is used to pass inputs to routines and to return results from routines[7]; H5 is used to indicate the results of tests. The contents of a storage cell are considered to be the top item on the stack rather than the whole stack. The IPL storage cells resemble the public lists of SLIP; in fact, the SLIP public lists are derived from the IPL storage cells.

An IPL instruction has four parts: the *name*, the *prefixes*, the *symbol*, and the *link*. The name is simply a label for the instruction, though in addition (as in SNOBOL) it can be used to name a routine. Names in IPL consist of either a letter followed by up to four digits or "9-" followed by up to four digits, e.g., "J521", "R2", "9-10".[8] The prefixes, called P and Q, specify the operation to be performed on the symbol and the degree of indirectness involved. The symbol represents a storage cell, whose contents may be a routine. The link, if specified, names a symbol to which control is to be transferred. Let S designate the storage cell named by the symbol. Then the actions caused by the various values of P are

0 Execute the routine in S.
1 Push down H0 and copy S into H0.
2 Copy H0 into S and pop up H0.
3 Pop up S.
4 Push down S, leaving the previous contents of S in S.
5 Copy S into H0.
6 Copy H0 into S.
7 Branch to S if H5 is negative.

[7] Arguments are passed to routines by stacking them in H0, so that IPL routines do not have formal parameters in the usual sense.

[8] These forms are quite unmnemonic, though experienced IPL programmers seem to be able to remember them.

A copy operation causes neither pushing down nor popping up; since the contents of a storage cell are considered to be the top item on its stack, a copy affects only that item.

Externally, IPL data and IPL programs are written in the same form; items in successive lines represent successive items of a list. Sublists must be named explicitly rather than through any parenthetic notation. The internal representation of IPL data resembles that of LISP, with the symbol corresponding to CAR and the link corresponding to CDR. The prefixes represent additional information not contained in LISP cells and are used to indicate the type of the datum designated by the symbol. Interestingly, prefixes were used in the very earliest (and unpublished) version of LISP, but were later dropped. Prefixes have been revived in some recent LISP implementations.

IPL provides to its users a large collection of primitive routines called *processes*; all of these have names starting with "J". As in L6, the user must handle most of the bookkeeping. Erasure is entirely the responsibility of the programmer; in particular, it is his responsibility to guarantee that an erased list is not part of another list.

There are facilities in IPL for utilizing auxiliary storage. Lists (which, of course, may represent programs) can be transferred to and from auxiliary storage fairly easily. The programmer can arrange to have his program trapped when storage is exhausted and then transfer data and programs to auxiliary storage.

The IPL implementation is based on an interpreter rather than on a compiler. Consequently, the system is generally not very efficient.

8.2 COMIT

COMIT is a string processing language originally developed by Yngve [*41*] in 1957 for use in mechanical translation. COMIT was the first major language to introduce pattern-matching statements, and its central ideas have had a strong influence in the whole field of symbol manipulation. Pattern-matching statements are now included in many different languages, several of which are discussed in this article.

The primary data depository of COMIT is the *workspace*. The contents of the workspace are a sequence of *constituents*, each of which is a symbol i.e., an arbitrary sequence of characters (unlike SNOBOL, where each constituent is a single character). Notationally, the constituents are separated by plus signs. Characters other than letters, periods, commas, and minus signs are all preceded by "*", and spaces are replaced by minus signs. An example of a sequence of constituents in a workspace is

THE + *2 *7 + MEN + ARE + HERE +.

Constituents may have *subscripts* attached to them. A *numerical sub-*

script is an integer. A *logical subscript* is a name with up to 36 possible associated *subscript values*, which are also names; any subset of the values may be present. The order of subscripts is immaterial. An example of a subscripted constituent is

MAN/.365,PART-OF-SPEECH NOUN VERB, CLASS HUMAN

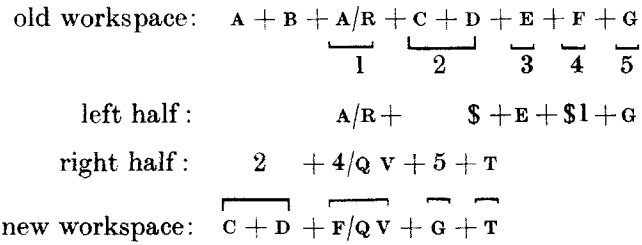
Here the numerical subscript is 365 (not .365), the logical subscripts are PART-OF-SPEECH and CLASS, the values of PART-OF-SPEECH are NOUN and VERB, and the value of CLASS is HUMAN.

A COMIT program consists of a set of *rules*, each of which in turn consists of a set of *subrules*. There are two kinds of rules: *grammar rules* and *list rules*. Grammar rules are used primarily to detect and transform complex patterns of constituents in the workspace, while list rules are used to operate on a single constituent on the basis of a dictionary lookup.

The first subrule of a grammar rule corresponds more or less to a SNOBOL statement. The five parts of the first grammar subrule are the *rule name* (corresponding to the SNOBOL label), the *left half* (corresponding to the SNOBOL pattern), the *right half* (corresponding to the SNOBOL replacement), the *routing* (which produces side effects not producible in the right half) and the go-to (like SNOBOL). As in SNOBOL, parts of a rule may be omitted. If a rule has more than one subrule, the second and remaining subrules contain alternate right halves, routings, and go-to's. A grammar rule with only one subrule is executed by matching the left half against the workspace, replacing the matching sequence of constituents as specified by the right half, executing the routing, and transferring to the go-to (or to the next rule if the match failed). If there is more than one subrule, then each subrule must have a *subrule name*. An area called the *dispatcher* will have space set aside corresponding to the rule name, with a logical value entered for each subrule. After the left half is matched, the subrule corresponding to the only true value (if there is only one such subrule) will be executed. If there is more than one, a pseudo-random choice is made among the true ones; if there is none, a pseudo-random choice is made among all the possible ones. This choice may, however, be made instead by the go-to of the previously executed rule. Logical subscripts may be used as rule names, and there are ways to use the values of a logical subscript of a constituent in order to select a rule and a subrule.

The possible constituents of a left half are full constituents, $\$n$ constituents, indefinite constituents, and integers. A *full constituent* is a symbol that matches an identical symbol in the workspace. A $\$n$ *constituent* matches n arbitrary successive constituents. An *indefinite*

constituent, indicated by "$", represents an arbitrarily long sequence of constituents. An integer j represents the constituent or constituents that match the jth constituent of the left half, and is used for back referencing. Constituents of a left half may be modified by subscripts, thus specifying additional requirements for a match. A right half may contain only full constituents and integers, where the integer j represents the jth constituent of the left half. Again, subscripts may be used in the right half to modify, replace, or delete existing ones and to insert new ones. An example of a pattern match and its result is



A *list rule* consists of two or more *list subrules*, and corresponds to a dictionary and its entries. The list subrules resemble the first grammar subrule, except that each left half is a single symbol. The subrules are automatically alphabetized by their left halves to facilitate rapid search, and their number is not restricted to 36. Control can only reach a list rule from the go-to of another rule; the selection of a subrule of the list rule will ordinarily be determined by a constituent of the pattern that was matched by the previously executed rule.

Since it is inconvenient to keep all the data in the workspace, COMIT provides a numbered set of temporary storage locations called *shelves*. The contents of the workspace may replace, be exchanged with, or be added to the contents of a shelf. This operation is performed by the routing of a rule.

COMIT has been almost entirely superseded by SNOBOL, though COMIT, like IPL, still has a few adherents. It does have two minor advantages over SNOBOL. First, for linguistic processing the ability to have constituents of more than one character is often convenient. Second, the dictionary search operations of COMIT have no exact parallel in SNOBOL and therefore cannot be done quite as efficiently in SNOBOL.

8.3 EOL

EOL is a low-level language for manipulating strings of characters. It was originally designed by Lukaszewicz in Poland and later revised and implemented by Lukaszewicz and Nievergelt [*22*, *23*] at the University of Illinois in 1967. Conceptually, an EOL program should be

thought of as a machine-language program running on a hypothetical computer called the EOL machine. An EOL program is built up from machine language through macro definitions and subroutine calls, so that EOL programs are quite hierarchical.

The EOL computer is equipped with inputs, outputs, files, and stacks. An *input* is a source of characters and an *output* is a sink of characters. In practice, inputs and outputs correspond to such devices as card readers, line printers, or magnetic tapes. A *file* is used to provide mass storage, and may correspond to core, drum, disk, or tape. Internal processing in the EOL computer is done mostly on the *stacks*, which are linear lists of *constituents*. Each constituent in a stack is a string of characters; a special mark preceding each constituent indicates its type: word (i.e., alphanumeric), number, or address. A one-bit register H, similar to H0 in IPL, is used to hold the results of tests.

An EOL program consists of a sequence of macro definitions followed by a sequence of external procedures. An external procedure may itself contain macro definitions. External procedures together with their required macro definitions may be compiled independently; macro definitions may also be compiled independently. A procedure consists of a sequence of statements, each of which may be a machine-language instruction, a macro-instruction, a procedure definition, a declaration, or a comment. A procedure is external if it is not contained within any other procedure.

There are about 50 basic instructions in the EOL machine, and their format is reasonably mnemonic. The stack instructions permit, for instance:

(1) Moving a specified number of constituents from the beginning of one stack to either end of a different stack in either the same order or in reverse order.
(2) Compressing several words into one word or splitting one word into several words of one character each.
(3) Testing whether the initial or the final constituent of a stack is equal to a given word or number.

Instructions may be made conditional on whether their operands start with characters from particular character classes. Input instructions are used for reading, and output instructions for writing; formatted output is possible. Files can be broken down into records, and records can be labeled.

Although rather difficult to program in, EOL appears to be a quite flexible language. In particular, the fact that all executable statements are in the format of instructions is a significant handicap; one would very much like to have infix and prefix operators, and to be able to

compose expressions from them. Also, the macro definition facility does not permit operations on the macro parameters other than direct substitution, so that there is no way to write macros that analyze their arguments.

8.4 A Few More

AMBIT (Algebraic Manipulation By Identity Transform) [*8*] is a language developed by Christensen at Computer Associates in 1965. The language has been applied to symbolic manipulation problems other than algebraic manipulation. Essentially, AMBIT is a block-structured language in which the statements consist of replacement rules as in COMIT. A replacement rule has two parts: the *citation*, corresponding to the left half, and the *replacement*, corresponding to the right half. A novel feature is the use of pointers as place markers in matching the workspace. Pointers may appear in both the citation and the replacement, and matching always begins with a pointer. This convention is used as the basis for some interesting implementation techniques [*9*].

CONVERT [*15*], developed by Guzman and McIntosh at the University of Mexico, is an augmentation of LISP to include pattern matching facilities. Its two central functions are RESEMBLE, which matches patterns, and REPLACE, which replaces them. Matching can be carried out against segments of lists and against sublists of lists, using patterns similar to those of SNOBOL4; RESEMBLE creates a dictionary associating variables with pattern components, and REPLACE uses this dictionary in the replacement. A similar augmentation of LISP called FLIP [*6*] has been developed by Bobrow and Teitelman. FLIP was intended to lead to pattern-matching facilities in LISP 2, but these facilities of LISP 2 were never fully specified, much less implemented.

Lombardi [*21*] uses list processing as the basis of his approach to incremental computation. His *incremental computer* is a simulated computer in which programs are specified with gaps, using incompletely defined functions. During the evaluation of such a function, any undefined variables or subfunctions that appear can then be provided at the time they are needed from, say, an on-line terminal. Lombardi's treatment of list processing is more formal than the one used in this article.

PANON-1B [*7*] is a symbol manipulation language developed by Caracciolo and his associates at the University of Pisa. It is based on a particular extension of Markov normal algorithms and consists of a sequence of transformation rules to be applied to an argument string according to appropriate sequencing rules.

COGENT (COmpiler GENerator and Translator) [*32*] is a programming system designed primarily as a compiler–compiler, i.e., a compiler that produces other compilers. However, it is also applicable to more general symbolic and linguistic applications. Its basic approach is to describe the data to be operated on in terms of syntax equations, and then to specify transformations on this data in terms of these syntax equations. It thus is a pattern-matching language, where the constituents of the match are syntax terms and the matching process may well involve recursive computations.

9. Concluding Remarks

In reviewing the collection of symbol manipulation languages given here, two divergent approaches become apparent. On the one hand, LISP, SNOBOL, SLIP, and PL/I are higher-level languages that include symbol manipulation facilities. As we pointed out in Section 1.2, such languages may arise either through the embedding of symbol manipulation facilities in a general-purpose language or through the expansion of a symbol manipulation language to include general computation. L6 and EOL, on the other hand, are low-level languages. Their simplicity contrasts sharply with the complexity of the higher-level languages, but this simplicity is obtained at the cost of making the user do more work.

Pattern-matching is a recurrent theme in symbol manipulation languages. Pattern-matching provides a nonprocedural method of specifying transformations on symbolic data, and it promises to be one of the dominant features of symbol manipulation languages in the future. Already, pattern-matching facilities have been embedded in LISP, a language that originally lacked these facilities. However, it is not easy to embed pattern-matching in an arbitrary language. In PL/I, for instance, the diversity of data types and the use of structures makes it difficult to define a standard data form, comparable to SNOBOL strings and LISP lists, on which matching and replacement could be done.

The techniques of symbol manipulation are finding increasing application in such specialized fields as computer graphics and compiler construction. Though the languages discussed here have been applied in these areas to only a limited degree, the concepts and implementation techniques of these languages have been applied extensively.

Symbol manipulation is a rapidly expanding branch of computing, but it is still considered somewhat exotic by the mass of computer users. Consequently, there has been relatively little pressure for standardization of symbol manipulation languages, and the "let a hundred flowers bloom" view has prevailed. New languages continue to appear, and old

ones are constantly being revised. It appears likely that symbol manipulation languages will stabilize as their use becomes more widespread, in much the same way as scientific and commercial languages have stabilized. At present, LISP and SNOBOL dominate the field (algebraic manipulation excepted), and LISP 1.5 has been reasonably stable for several years. Although experimentation is sure to continue, the day when symbol manipulation is just another way of massaging data is probably not far off.

ACKNOWLEDGMENT

The writing of this article was supported by the AEC Computing and Applied Mathematics Center, Courant Institute of Mathematical Sciences, New York University, under Contract AT (30-1)-1480 with the U.S. Atomic Energy Commission.

REFERENCES

1. Abrahams, P., List-processing languages, in *Digital Computer User's Handbook* (M. Klerer and G. Korn, eds.), 1-239–1-257. McGraw-Hill, New York, 1967.
2. Abrahams, P., *et al.*, The LISP 2 programming language and system. *Proc. AFIPS Fall Joint Computer Conf., San Francisco*, **29**, 661–676.
3. Association for computing machinery. *Comm. ACM* **9**, entire issue (1966).
4. Bobrow, D. G., and Murphy, D. L., Structure of a LISP system using two-level storage. *Comm. ACM* **10**, 155–159 (1967).
5. Bobrow, D., and Raphael, B., A comparison of list processing languages. *Comm. ACM* **7**, 231–240 (1964).
6. Bobrow, D., and Teitelman, W., Format-directed list processing in LISP. Tech. Rept., Bolt, Beranek, and Newman, Inc., Cambridge, Massachusetts, 1966.
7. Caracciolo di Forino, A., and Wolkenstein, N., On a class of programming languages for symbol manipulation based on extended Markov algorithms. *Centro Studi Calcolatrici Elettroniche del C.N.R.* [2], **21**, (1964).
8. Christensen, C., Examples of symbol manipulation in the AMBIT programming language. *Proc. ACM Natl. Conf., 20th, Cleveland, Ohio, 1965*, pp. 247–261. Thompson, Washington, D.C.
9. Christensen, C., On the implementation of AMBIT, a language for symbol manipulation. *Comm. ACM* **9**, 570–572 (1966).
10. Cohen, J., A use of fast and slow memories in list-processing languages. *Comm. ACM* **10**, 82–86 (1967).
11. Farber, D. J., Griswold, R. E., and Polonsky, I. P., SNOBOL, a string manipulation language. *J. Assoc. Comput. Mach.* **11**, 21–30 (1964).
12. Farber, D. J., Griswold, R. E., and Polonsky, I. P., The SNOBOL3 programming language. *Bell System Tech. J.* **45**, 895–944 (1966).
13. Gelernter, H., A FORTRAN-compiled list processing language. *J. Assoc. Comput. Mach.* **7**, 87–101 (1960).
14. Griswold, R. E., Poage, J. F., and Polonsky, I. P., Preliminary report on the SNOBOL4 programming language (II). Rept. S4D4, Bell Telephone Labs., Holmdel, New Jersey, November 1967.
15. Guzman, A., and McIntosh, H., CONVERT. *Comm. ACM* **9**, 604–615 (1966).

16. Information International, Inc., *The Programming Language* LISP*: Its Operation and Applications* (E. C. Berkeley and D. G. Bobrow, eds.), Inform. Intern., March 1964 [reprinted by M.I.T. Press, Cambridge, Massachusetts, 1966].
17. International Business Machines Corporation, IBM System/360 Operating System: PL/I Language Specifications. Form C28-6571-4. IBM, New York, 1966.
18. International Business Machines Corporation, IBM System/360: PL/I Subset Reference Manual. Form C28-8202-0. IBM, New York, 1967.
19. Knowlton, K. C., A programmer's description of L6. *Comm. ACM* **9**, 616–625 (1966).
20. Lawson, H. W., PL/I list processing. *Comm. ACM* **10**, 358–367 (1967).
21. Lombardi, L. A., Incremental computation. *Advan. Computers* **8**, 247–333 (1967).
22. Lukaszewicz, L., and Nievergelt, J., EOL programming examples: a primer, Rept. No. 242, Dept. of Computer Sci., Univ. of Illinois, Urbana, Illinois, September 1967.
23. Lukaszewicz, L. and Nievergelt, J., EOL report, Rept. No. 241, Dept. of Computer Sci., Univ. of Illinois, Urbana, Illinois, September 1967.
24. McCarthy, J., Recursive functions of symbolic expressions and their computation by machine, pt. I. *Comm. ACM* **3**, 184–195 (1960).
25. McCarthy, J., *et al.*, *LISP 1.5 Programmer's Manual.* M.I.T. Comput. Center and Res. Lab. of Electron., 1963 [reprinted by M.I.T. Press, Cambridge, Massachusetts].
26. Madnick, S., String processing techniques. *Comm. ACM* **10**, 420–423 (1967).
27. Newell, A., and Shaw, J. C., Programming the logic theory machine. *Proc. Western Joint Computer Conf.*, 1957.
28. Newell, A., *et al.*, *Information Processing Language-V Manual*, 2nd ed. Prentice-Hall, Englewood Cliffs, New Jersey, 1964.
29. Perlis, A., and Iturriaga, R., An extension to ALGOL for manipulating formulae. *Comm. ACM* **7**, 127–130 (1964).
30. Perlis, A., and Thornton, C., Symbol manipulation by threaded lists. *Comm. ACM* **3**, 195–204 (1960).
31. Perlis, A. J., Iturriaga, R., and Standish, T., A definition of Formula ALGOL, Carnegie Inst. of Technol., Pittsburgh, Pennsylvania, August 1966.
32. Reynolds, J. C., COGENT programming manual, AEC Res. and Develop. Rept. ANL-7022, Argonne Natl. Lab., Argonne, Illinois, March 1965.
33. Rosin, R. F., Strings in PL/I, PL/I Bull. No. 4, SICPLAN Notices **2**, Assoc. for Comput. Machinery.
34. Ross, D. T., A generalized technique for symbol manipulation and numerical calculation. *Comm. ACM* **4**, 147–150 (1961).
35. Ross, D. T., The AED free storage package. *Comm. ACM* **10**, 481–492 (1967).
36. Sammet, J. E., Formula manipulation by computer. *Advan. Computers* **8**, 47–102 (1967).
37. Smith, D. K., An introduction to the list processing language SLIP, in *Programming Systems and Languages* (S. Rosen, ed.), pp. 393–418. McGraw-Hill, NewYork, 1967.
38. System Development Corporation, LISP 2 for the IBM S/360, Doc. Ser. TM-3417, System Develop. Corp., Santa Monica, California, April 1967.
39. Weissman, C., *LISP 1.5 Primer.* Dickenson, Belmont, California, 1967.
40. Weizenbaum, J., Symmetric list processor. *Comm. ACM* **9**, 524–544 (1963).
41. Yngve, V. H., *COMIT Programmer's Reference Manual.* M.I.T. Press, Cambridge, Massachusetts, 1962.