SHARE PROGRAM LIBRARY AGENCY

PROGRAM NUMBER

068002

University of Miami

1365 MEMORIAL DRIVE - CORAL GABLES, FLORIDA

(305) - 284-6257

CONTRIBUTED PROGRAM LIBRAF
(for IBM S/360, 1130 anc

SHARE Program Library Agency
Triangle Universities Computation Center
P. O. Box 12076
Research Triangle Park, N. C. 27709

This form should be completed and submitted with the program p
instructions for submitting programs are in your *User Group Refe*
Standards Manual available from PID.

(1) **Program Order Number** *(to be filled in by PID)* 360D-06.8.002

(2) **System Type** *(machine)* S/360

(3) **Search Key** . . . . . LPI-/A//SYSTEM/ /FOR/
/FORTRAN/ LIST PROCESSING

(4) **Name of Author** *(if different than submitter's)* . . . . . Dean Ritchie

(5) **Submitter's Name** *(direct technical inquiries to)* . . . . [redacted]

(6) **Submitter's Address** . . . . . Computing Center
Washington State University
Pullman, Washington 99163

(7) **Title of Program** . . . . LPI

(8) **Submitter's User Group Affiliation Code and Installation Code** . . . . . S UY

(9) **Submitter's Own Program Identification and Suffix** *(optional)* . . . . .

(10) **Primary Subject Code** 06.8

(11) **Secondary Subject Codes** . . . . .

(12) **Operating or Monitor System Required** OS/360

(13) **New or Revision Code** *(if revision, show prior Program Order Number in item 1)* . . . . . N

(14) **Year Completed** 67

(15) **Date of Submittal** 062569

(16) **Documentation** *(number of original pages submitted)* 37

(17) **Abstract** *(should contain sufficient information for a reader to determine the value of the program). Listed on the reverse side of this form are subjects which may serve as a guide for a descriptive abstract.*

CONTRIBUTED PROGRAM LIBRARY SUBMITTAL FORM

Subject Guide

a. **Purpose**
b. **Programming Language used**
c. **Version and modification level or release number of IBM Programming System used, or program order number for non-IBM authored program used**
d. **Field of application**
e. **Type of routine (main program, subroutine, etc.)**
f. **Specific description of machine requirements**
g. **Engineering Changes (EC) level of equipment (if pertinent)**

ABSTRACT

LPI is a small set of subprograms for use by Fortran programmers to perform the basic functions of list processing. This paper describes and evaluates LPI, comparing it specifically with SLIP, a similar system.

LPI requires only sufficient hardware to compile and execute Fortran programs.

DISCLAIMER

Triangle Universities Computation Center (TUCC) serves solely as the distribution agent for contributed programs and does not test or maintain them. They are distributed essentially in the original form submitted by the author. Neither TUCC nor SHARE, INC., makes any warranty, expressed or implied, as to the documentation, function, or performance of the contributed programs.

(Please attach additional pages if necessary) . . . . . **Total pages attached** _____

Permission to Publish

"I hereby give anyone permission to reprint, reproduce, and distribute this program to anyone else."

(18) **Signature of Submitter and Date** H. Blair Benner, 6/26/69

(19) **Signature of Installation Addressee** Jesse S. Watkins

T4SF

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

CARD DECK KEY

One Deck - Assembler language source deck

209 cards

CHAPTER I

LIST PROCESSING SYSTEMS

Introduction

Most algebraic programming systems provide for a sequence or array of data, with the convention that adjacent items of a sequence occupy adjacent memory locations. This approach has proved adequate for a very large set of computer applications. In fact, the long-standing popularity of Fortran is evidence of a great need for just such a programming system.

However, as computer problems have become more diverse, a need has arisen for data representation techniques which are more flexible. The sequential storage representation presents severe problems whenever the order of a long sequence of data must be changed frequently, because each subsequent item must be moved whenever one is moved. Other problems arise when it is necessary to process several sequences of data, each of which might vary widely in length during the process, because algebraic systems generally must fix the length of each array before the process begins.

Definitions

Threaded lists[1].--The threaded list concept is one way to represent a sequence of data in computer memory so that the location of the successor of an item is independent of the location of the item. Under this system, each item of data has associated with it the information necessary to find its successor.

Sublists and list structures[2].--Whenever a single item of a threaded list indicates another threaded list, we refer to the second list as a sublist of the first. Such a representation allows the grouping of data structures, much like the outline of a book. That is, the main list might be a list of the chapters, each of which is a list of paragraphs. The advantage here is that it is not necessary to search through all the paragraphs in the book to find a particular paragraph, but only the paragraphs of the desired chapter, after it has been found by searching through the chapters. In modern data processing problems, it is frequently necessary to process highly complex structures of data, with many levels of sub-structuring, and a high degree of cross-referencing.

A familiar example of such a structure is a complete library card catalogue. A card catalogue might be represented by a group of lists which contain the information normally present

---

[1] The concept of "threaded lists" is quite widely known and used among computer programmers. I believe its discovery is usually attributed to Professor Wilkes of Cambridge University.

[2] Donald Knuth, in [1], refers to list structures as "trees." Further illustrations of such structures may be found in section 2.3, pp. 305-313 of [1].

on a catalogue card, where there is one such list for each book. The title file could then be represented as a list of pairs of items, the first item of each pair being the title of the book, and the second item being a reference to the sublist which contains the complete description of the book. The author file could be constructed similarly. The subject file might be a list structure, the higher levels of which are lists of more general headings coupled with sublists, which are subheadings coupled with still lower sublists, and whose lower levels are simply lists of references to the book description lists which fall into that particular category.

Many of these structures submit rather easily to representation as threaded lists and list structures. However, the fact that a structure submits easily to such representation does not mean that the representation is the most practical to use (or even that there is a practical representation). In the above example, it might be more practical to construct the author file as a list of author-sublist pairs, where the sublist is a list of the works by that author.

List processing systems.--A programming system is a list processing system if it specifically includes features for building and processing threaded lists. This includes the facility to read and process external inputs (i.e., from cards, magnetic tape, etc.), build lists from these data, analyze and modify these lists, and write or print results of the processing. Such a

system usually involves some sort of memory pool[3], which furnishes space for lists, and a scheme for returning space to that pool as the data become obsolete or unneeded.

Examples

IPL-V [2].--IPL-V is a rather low-level programming language, which produces threaded lists in much the same fashion as an assembly-language produces machine code. Figure 1 is an example of input to IPL-V, which causes a list structure to be generated. The system automatically fills in the "LINK" fields to indicate the following item, whenever it is left blank. The symbols 9-1, 9-2, and 9-3 are defined in the figure as sublists, and each occurrence of one of them in a SYMB field constitutes a sublist reference. The other symbols are assumed to refer to data areas somewhere in the computer memory. The end of a list is indicated by a zero successor address. IPL-V programs are similar to list structures, except the items of a program also have prefixes, which indicate the operation to be performed on the SYMB parameter[4].

Symmetric list processor [5].--SLIP is a large set of subprograms which are compatible with Fortran, in the sense that they can be called as Fortran subprograms. Each subprogram is designed to perform some function associated with list processing.

---

[3]The concept of a memory pool, like that of a threaded list, is quite well known and widely used. The reference [3] is generally credited with its first application in a list-processing system.

[4]The reference [2] has several examples of IPL-V programming as well as a much broader discussion of the list structures.

NAME	SYMB	LINK
L6	0	
	S1	
	9-1	
	S2	
	9-2	0
9-1	0	
	S3	
	9-3	
	9-3	
	S4	0
9-2	0	
	S5	
	S6	0
9-3	0	
	S7	
	S8	
	S9	
	S10	0

Fig. 1.--An IPL-V list structure, adapted from Allen Newell, ed., Information Processing Language-V Manual (Englewood Cliffs, New Jersey: Prentice-Hall, 1961), p. 46.

SLIP lists are constructed from two-word cells, which have two addresses in the first word, and the datum in the second word. The addresses are both used to thread the list, in the sense that one is the address of the predecessor and the other is the address of the successor of the cell. The end of the list is indicated by the fact that it is succeeded by a special cell called a header cell. Another special type of cell is used when the datum is to be a sublist, and the datum portion contains the address of the sublist's header cell. Figure 2 illustrates these concepts.

Summary

Threaded lists constitute a fairly general means of representing complex data structures, and list processors generally reduce the programming effort required to construct programs which handle such structures. Among list processing systems in existence, there is a wide range of choices available[5], from a completely independent programming language such as IPL-V, to a simple extension of Fortran like SLIP.

[5] A list of list-processing systems and related programs, along with short descriptions, can be found in [4].

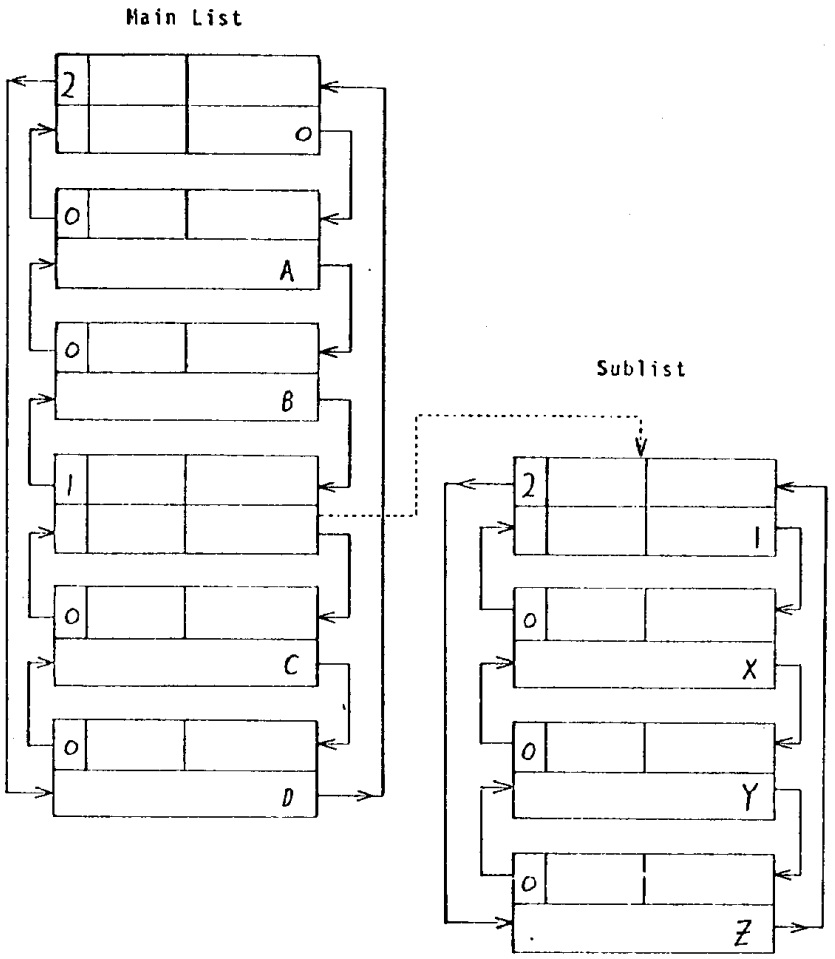

Fig. 2.--A simple SLIP list structure (A,B,(X,Y,Z),C,D), adapted from J. Weizenbaum, "Symmetric List Processor," Communications of the ACM, Vol. VI, No. 9 (1963) 526, Fig. 3. Copyright, 1963, Association for Computing Machinery, Inc.

CHAPTER II

THE LPI SYSTEM

Definitions

Introduction.--LPI, like SLIP, is a set of subprograms which may be invoked by Fortran programs to furnish various list-processing functions. These programs have been implemented for the IBM System/360 computer, so we define the basic structures on which the LPI subprograms operate in terms of the IBM data structures. Clearly, the definitions may be modified to operate as well for almost any other computer.

Cell.--An LPI cell is a unit of eight bytes of computer memory which is logically divided into the prefix, the pointer, and the datum fields. The prefix is one byte, the pointer is three bytes, and the datum is four bytes, which is the same size as the standard Fortran data item.

List.--An LPI list is a collection of cells c1, ..., ck, such that: The prefix of c1 contains 2, and none of c2, ..., ck has a prefix value of 2; the pointer of ci contains the memory address of ci+1 for i=1, ..., k-1, and the pointer of ck contains the memory address of c1. Figure 3 illustrates these ideas.

Special cells.--The cell c1, whose prefix value is 2, is called the list header, and is used to mark both the beginning and the end of the list. Any cell whose prefix has the value 1

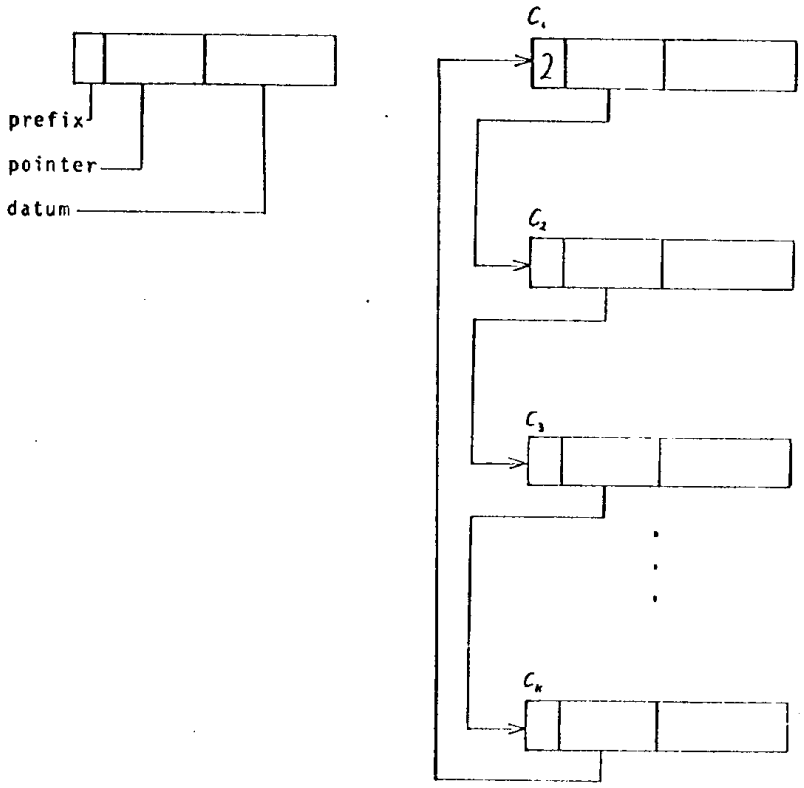

Fig. 3.--A cell and a list

is called a sublist reference, and must contain the memory address of a list header in its datum field. The list whose header is indicated is a sublist of the list which contains the sublist reference. If L2 is a sublist of L1, and L3 is a sublist of L2, then we say that L3 is also a sublist of L1.

The list of available space.--During the operation of a program which uses LPI, the list of available space contains all of the cells which are available for building lists or list structures. The list of available space is similar to any LPI list, in the sense that its cells are threaded together by the values of their pointer fields. However, instead of using header cells, the beginning and end of the list of available space are indicated by internal pointers, which are automatically updated by LPI.

Markers.--In LPI, the concept of a marker is quite important. This is a device for keeping track of a position in a list, much as a bookmarker is a device for keeping a position in a book. In LPI, any Fortran integer variable can be used as a marker simply by assigning it the value of the memory address of the cell in the position to be marked. For example, if the value of a variable is the memory address of a list header, then it marks the head of that list.

The Programs

LPI consists of five Fortran subprograms, written for the IBM System/360 computers in Assembly Language. In the following paragraphs, the use of these programs will be illustrated by model Fortran statements, using dummy parameter names to illustrate their use.

Initializing the list of available space.--Any program which uses LPI must set aside a Fortran array for the exclusive use of the LPI subprograms as a list of available space. Further, the list of available space must be initialized before any other LPI programs can be used. The statement CALL INIT(A1,A2), where A1 is the first element of the array to be used, and A2 is an integer whose value is the array size (as specified in the DIMENSION statement which defined it), will cause the array to be initialized as the LPI list of available space. WARNING: This statement should be executed only once for any particular program execution.

Building a list.--The statement V=ADD(A1,A2,2) will generate a list which consists of one cell. From the definition of a list, we see that this means that the cell must be a header cell (its prefix has a value of 2), and its pointer must contain its own address. The datum portion of the cell will be set to the value of A2, and the values of both A1 and V will be set to the memory address of the header cell. This introduces some amount of redundancy, but it is necessary, because the ADD function is used for another purpose, where its first argument is significant. The redundancy is not unpleasant, however, since it has been found to be an advantage to record the address of the list header in two places in certain situations.

Expanding a list.--If M is a marker, then a cell whose prefix is P and whose datum is D can be added just after the position marked by M, by execution of the statement V=ADD(M,D,P). The value of P must be less than 256, unequal to 2, and if it is

1, then the value of D must be the address of the header of a sublist, in which case the added cell is a sublist reference. V becomes a marker which marks the newly added cell.

Scanning a list.--Given the concept of a marker, it is necessary to be able to cause a marker to be advanced along a list, retrieving the contents of each cell as the marker reaches it. If A1 is a marker, the statement V=ADVANC(A1,A2) will cause A1 to be advanced to the next cell, and will set A2 to the contents of the prefix of that cell, and V to the contents of its datum field. Note that by checking the value of A2, it is possible to recognize the header cell of the list, which signifies that the end of the list has been reached.

Deleting a cell.--If A1 is a marker, then the next cell after the position marked by A1 may be deleted by execution of the statement V=DELETE(A1,A2). A2 will be set to the value of the prefix of the deleted cell and V to the value of its datum. A1 remains unchanged. EXCEPTION: Execution of the above statement will not cause a header cell to be deleted. In case the cell after the position marked by A1 is a header cell, then the effect of the DELETE statement will be the same as if an ADVANC had been issued instead of the delete.

Erasing a list.--Whenever the data of a list are no longer needed for the process, the cells of that list should be returned to the list of available space to be reused. Under conditions described in the next section, this may be accomplished by executing the statement V=ERASE(A1), where A1 is a marker to the header of the list. The resultant value of V is not significant.

Collecting Unused Cells

LPI is constructed so that a cell is only retrieved from the list of available space when it is to be used as a list header, or when it is to be inserted into some list. Similarly, whenever cells are deleted from any list, they are automatically returned to the list of available space. Thus the only concern of the programmer is that lists and sublists be accounted for, and all the space allowed for the list of available space can be used in active lists.

The reference counter[1].--In order to help keep track of sublists, LPI uses the datum portion of each list header as a reference counter. This count is initialized to the value of A2 in the ADD instruction that causes the list to be generated. Subsequent ADD instructions that generate sublist references to the list automatically cause its reference counter to be incremented by 1. Further, each time a sublist reference is returned to the list of available space, or a list is referenced by an ERASE instruction, the reference counter is automatically decreased by 1. A list's cells are actually returned to the list of available space only when its reference counter becomes zero in this fashion.

Use of the reference counter.--A good general rule for using the LPI reference counter is to always initialize the

[1]The reference counter is a feature of LPI which has been borrowed directly from SLIP. For a good discussion of its reason for being, see [5], p. 526, starting with the last paragraph, and continuing through p. 527.

reference counter of a list to 0 when it is to be used primarily as a sublist, and to initialize all other lists' reference counters to 1. Then, when the data of a list become obsolete, the ERASE program should be invoked only if the list was generated with a reference count of 1. Sublists which are generated with a reference count of 0 will automatically be erased when they are no longer referenced as sublists.

Restriction.--If the reference counters are to work properly, then no list may be generated in such a way that it is a sublist of itself. Of course, such a list may be generated, but it can never be erased automatically. In fact, the reference to itself must be deleted before it can be erased at all.

CHAPTER III

EVALUATION OF LPI AS A LIST-PROCESSING SYSTEM

Criteria of Evaluation

In order to evaluate a list-processing system, we must first review the qualities generally expected in such a system. Since the input-output, arithmetic capabilities, and program logic facilities of LPI are contained in Fortran, we pass over these features, not because they are insignificant, but rather because they are well known. The remaining items to be discussed are dynamic memory allocation, representation of complex data structures, retrieval of information from lists, garbage collection, and recursive programming capabilities.

Since SLIP is also a Fortran-based system, it is quite natural to compare LPI to SLIP. On the surface, LPI is a very minimal system in comparison to SLIP, since SLIP comprises well over 100 separate subroutines, and LPI consists of 5. Furthermore, SLIP operates in a symmetric environment, in that its lists are threaded both ways, i.e., each cell has both a forward and a backward pointer.

Comparison of LPI and SLIP

We first consider those qualities mentioned above, and how LPI compares to SLIP in regard to them.

Dynamic memory allocation.--Since both LPI and SLIP operate under Fortran, the capabilities of each for dynamic memory allocation are rather severely limited. What these systems do is to take a Fortran array, which has been dedicated to that purpose, and allocate its elements to threaded lists as they are required. When the array is exhausted, either system simply terminates the program run.

Representation of data structures.--The structure of SLIP allows the representation of a bi-directional threaded list structure, which is somewhat more general than the lists created by the LPI system. For most applications, the SLIP structures are considered as sequences ordered in one direction, with the convenience that one can "back up" a marker or reader. For those applications, LPI adequately represents the same data structures, as illustrated by Fig. 4. The facilities for building these structures are similar. LPI furnishes a single program to create the list header cell and insert elements into a list. SLIP uses several separate subprograms to accomplish these tasks, with the added convenience that SLIP can insert a new element on either side of a known entry in a list, where LPI requires new elements to be inserted after a known cell.

Retrieval of information.--SLIP provides two mechanisms for scanning a list structure. The first, called a sequence reader, is similar to the marker concept of LPI, except for the fact that a sequence reader may be advanced in either a forward or a backward direction. The second, called a reader, involves a special pushdown stack which can be advanced through an entire

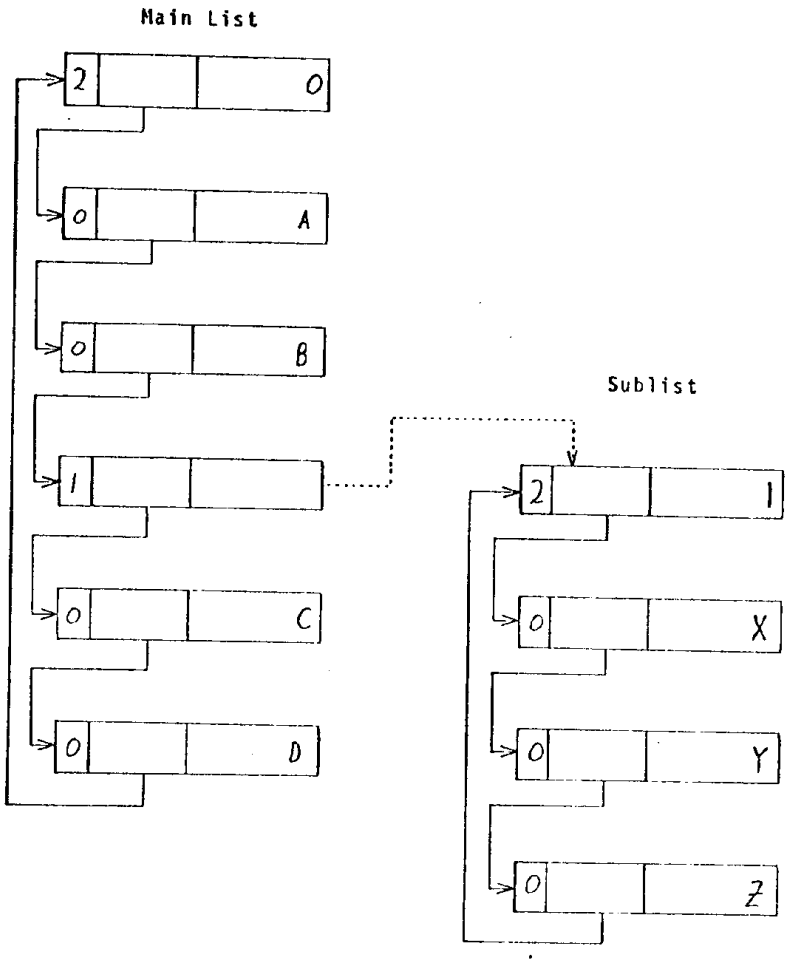

Fig. 4.--The LPI representation of (A,B,(X,Y,Z),C,D), the structure of Fig. 2.

list structure by pushing down information each time a sublist reference is encountered, and popping up information each time the end of a sublist is encountered. This process takes place automatically in SLIP, but obviously it can be duplicated using LPI. To illustrate this, suppose that some program requires access to each item of data on a list structure. We will say that when we have retrieved a new item from the structure we will branch to statement 10, where the processing takes place. The program should return to statement 1 each time it needs a new item from the structure, and control is to be passed to statement 11 when the entire structure has been examined. Using SLIP, the program on the left will accomplish this, and using LPI, the program on the right will suffice.

```
    LRD=LRDROV(LIST)
1   X=ADVSWR(LRD,1)
    IF(1.EQ.0) GO TO 10
    1=IRARDR(LRD)
    GO TO 11
```

```
    LRD=LIST
    PSHDWN=ADD(PSHDWN,1,2)
1   X=ADVANC(LRD,PREF)
    IF(PREF.EQ.1) GO TO 2
    IF(PREF.NE.2) GO TO 10
    LRD=DELETE(PSHDWN,PREF)
    IF(PREF.NE.2) GO TO 1
    X=ERASE(PSHDWN)
    GO TO 11
2   Y=ADD(PSHDWN,LRD,0)
    LRD=X
    GO TO 1
```

Garbage collection.--The garbage collection feature of LPI is entirely similar to the feature of SLIP. These systems have automatic garbage collection, in the sense that deleted cells are automatically entered into the list of available space, and each sublist is automatically erased upon the deletion of all references to it.

Recursive programming.--Since both LPI and SLIP are wedded to Fortran, and Fortran subroutines cannot be called recursively,

there is really very little that can be done to facilitate recursive programming. About the only way that recursive programs can be written is to use the assigned go-to feature to set return points, and then program an internal subprogram which saves the return point indicator and any parameters that it might change on a pushdown stack, popping them off again before executing the go-to that will return control to the calling point. SLIP operates somewhat differently from this, but not significantly. To illustrate the differences and similarities of the two systems in this regard, we present the example in Fig. 5.

Advantages of SLIP over LPI

In processing threaded list structures, SLIP's primary advantages over LPI are its symmetry, which allows it the advantages mentioned in the paragraphs on information retrieval and building structures, and the following additional features.

Comparing or copying lists.--SLIP has programs for comparing two list structures, or for generating a copy of a list structure. We define equality of list structures by the following recursive definition: Two lists are equal if they have the same number of cells, and corresponding to each sublist reference in one is a sublist reference in the other, corresponding non-sublist reference items are equal, and corresponding sublists are equal lists.

Concatenating and truncating lists.--In SLIP, if we have two lists, L1 and L2, and C1 is an element of L1, then all the elements of L2 may be inserted into L1 on either side of C1, by a


```
      FUNCTION NFACT(N)
      COMMON W(100)

      ASSIGN 1 TO NCALL
      NTEMP=N
      X=VISIT(NCALL)

      RETURN
    1 IF(NTEMP.EQ.1)GO TO 2

      I=NEWTOP(W(1),NTEMP)
      NTEMP=NTEMP-1

      X=VISIT(NCALL)
      X=POPTOP(W(1))
      X=STRDIR(X,NTEMP)
      NFACT=NFACT*NTEMP
      CALL TERM
    2 NFACT=1
      CALL TERM
      END
```

```
      FUNCTION NFACT(N)
      IMPLICIT INTEGER(A-Z)
      X=ADD(PSHDWN,1,2)
      ASSIGN 11 TO NRET
      NTEMP=N
      GO TO 1
   11 X=ERASE(PSHDWN)
      RETURN
    1 X=ADD(PSHDWN,NRET,0)
      IF(NTEMP.EQ.1)GO TO 2
      X=ADD(PSHDWN,NTEMP,0)
      NTEMP=NTEMP-1
      ASSIGN 12 TO NRET
      GO TO 1
   12 NTEMP=DELETE(PSHDWN,PREF)
      NFACT=NFACT*NTEMP
      NRET=DELETE(PSHDWN,PREF)
      GO TO NRET,(11,12)
    2 NFACT=1
      NRET=DELETE(PSHDWN,PREF)
      GO TO NRET,(11,12)
      END
```

Fig. 5.--An example of a recursive process, as programmed in both SLIP and LPI.

A program to implement the recursive formula NFACT(N)= N*NFACT(N-1) if N > 1, NFACT(1)=1, using both SLIP, on the left, and LPI, on the right.

single program. Similarly, if L1 and C1 are as described above, then all the elements on either side of C1 may be removed from L1 and formed into a new list, L2.

Description lists.--The SLIP description list is a special appendage to a list, whose entries occur in attribute-value pairs. Description lists may be used to describe the contents of the list in any way desired.

List markers.--Associated with each SLIP list is a binary integer from 0 to 7, called the marker value of the list. These are frequently used when a list structure is being processed, so that sublists need only be processed once.

Advantages of LPI over SLIP

The principle advantages of LPI over SLIP seem to fall into two categories. The first of these consists of corrections of certain difficulties found in SLIP. The second category includes advantages which are generally conceded to a simple system or language over a more complex one, such as easier use, quicker mastery, and more efficient use of computer time and memory.

The extended prefix.--In SLIP, the ID section of a list cell is used to indicate whether the cell is a list reader cell, a list header, a sublist reference, or a datum. This idea has been extended in LPI to allow the prefix to range from 0 to 255. Of that range, only the values 1 and 2 have any special significance to the LPI system. Any others may be used by the programmer to indicate whatever he might wish about the datum portion of the

cell. One very useful example of this is in parsing an array of symbolic data, such as a computer program or a sentence. A special prefix value might be used to indicate that the symbol is not complete within this datum, and is continued in the next cell of the list.

Explicit specification of a reference count.--In using SLIP, when creating a list header, the programmer can specify that a sublist is to be created with an initial reference count of zero, by using a parameter whose value is 9. Alternately, he can use a variable parameter whose value is not 9, and a list with reference count of 1 will be generated, and the address of its header will be stored in the parameter. The trouble with this is that the programmer must check the parameter to make certain its value is not 9 in the latter case. LPI removes the possibility of coincidence by requiring that the initial value for the reference counter be included as a parameter of the ADD instruction which creates the list header.

Explicit specification of sublist references.--Another possibility for coincidence exists under SLIP, because the programs that insert data into a list automatically decide if a sublist reference is intended. LPI requires explicit specification of the prefix, and therefore will not create a sublist reference unless the prefix parameter is 1.

Conclusion

LPI is a very compact system of Fortran subprograms, each of which actually performs a basic operation peculiar to list

processing; i.e., add a cell to a list, delete a cell from a list, advance a marker along a list, and erase a list. This fact makes LPI very useful in a training situation, such as a university classroom, because its features correspond directly to what may be called basic elements of list processing.

The features of SLIP which are missing can be either simulated, as in the first example, or a separate subroutine can be written, using the LPI functions, to perform the equivalent function for each. Such expansions have deliberately been omitted from the development of LPI, in the hope that any researcher who might wish to use LPI will develop a system of programs which are adapted specifically to his needs, using the LPI functions as a base. Thus each experimenter might develop a system of Fortran subprograms, each of which performs a function as basic to his field as the LPI functions are to list-processing; such a system should be very useful to anyone who might wish to apply the results of the research.

Experience has shown LPI to be a very powerful, compact tool. It has been used at Washington State University in teaching classes in List Processing, Systems Programming, Compiler Languages, and Computational Linguistics, both as the direct object of instruction, and as a tool for implementing concepts discussed in class. As an example of the level of use in classroom work, the Computational Linguistics class wrote three small programs using LPI as classroom assignments, and most of the class wrote term projects using LPI, as well.

LITERATURE CITED

1. Knuth, Donald E. The Art of Computer Programming. Vol. I. Reading, Massachussetts: Addison-Wesley, 1968.

2. Newell, Allen (editor). Information Processing Language-V Manual. Englewood Cliffs, New Jersey: Prentice-Hall, 1961.

3. Newell, Allen, and H. A. Simon. "The Logic Theory Machine--A Complex Information Processing System," IRE Transactions on Information Theory, Vol. II, No. 3 (1956), pp. 61-79.

4. Raphael, B., D. G. Bobrow, L. Fein, and J. W. Young. A Brief Survey of Computer Languages for Symbolic and Algebraic Manipulation. Menlo Park, California: Stanford Research Institute, July, 1966.

5. Weizenbaum, J. "Symmetric List Processor," Communications of the ACM, Vol. VI, No. 9 (1963), pp. 524-544, copyright 1963, Association for Computing Machinery, Inc.

APPENDIX A

A SAMPLE PROGRAM, WITH STORAGE MAPS

The schematic representation of lists and list structures used in Fig. 4 is a very useful way to illustrate the logical relationships between elements of a threaded list structure, but tends to confuse the actual arrangement of data in computer memory. To illustrate the use of the Fortran array as a memory pool for LPI cells, we present the following program, and the "maps" of the available-space array at three points of the operation of the program, each marked by a Roman numeral to the right. For the sake of simplicity in the maps, we suppose that the memory address of the element, LSPACE(I), of the array is simply I. Also shown in each map are the values of the internal pointers to the beginning and end of the list of available space, called LAVS1 and LAVS2, respectively. The program follows:

```
      IMPLICIT INTEGER(A-Z)
      DIMENSION LSPACE(20)
      CALL INIT(LSPACE,20)          I
      X=ADD(L1,1,2)
      X=ADD(L2,1,2)
      X=ADD(L1,1,3)                 II
      X=ADD(X,L2,1)
      X=ERASE(L1)                   III
      STOP
      END
```


MAP I

LAVS1 = 1, LAVS2 = 19

Loc.	Pref.	Pointer	Datum
1,2	0	3	0
3,4	0	5	0
5	0	7	0
7	0	9	0
9	0	11	0
11	0	13	0
13	0	15	0
15	0	17	0
17	0	19	0
19	0	LAVS1	0

MAP II

LAVS1 = 7, LAVS2 = 19

Loc.	Pref.	Pointer	Datum
1	2	5	1
3	2	3	1
5	3	1	1
7	0	9	0
9	0	11	0
11	0	13	0
13	0	15	0
15	0	17	0
17	0	19	0
19	0	LAVS1	0

MAP III

LAVS1 = 9, LAVS2 = 1

Loc.	Pref.	Pointer	Datum
1	2	LAVS1	0
3	2	3	1
5	3	7	1
7	1	1	3
9	0	11	0
11	0	13	0
13	0	15	0
15	0	17	0
17	0	19	0
19	0	5	0

APPENDIX B

A SAMPLE PROGRAM, WITH ANNOTATIONS

It is always difficult to conceive a sample problem to illustrate a programming system which is neither too simple to illustrate the system nor so complex as to require a major document to explain the problem. In an attempt to provide such an example, I propose to discuss a rather well-known problem, which has reasonably well-known conventional solutions already available. The purpose of this discussion is not to present a better way of solving this particular problem, but rather to couch a discussion of possible techniques for using LPI in terms familiar to nearly everyone.

The problem is the evaluation of a simple polynomial, such as "a+b*c*d+e+f," where "*" is the symbol used for multiplication, as in Fortran programming. For the sake of simplicity, we will assume that the polynomial consists of a sum of products of one or more single-letter variables. Although it is clearly not difficult to solve this problem in a single step, or "pass" over the polynomial, we will use three separate steps, to illustrate the use of LPI.

First, a list of terms will be generated, where a term may be either a cell whose prefix is 4 and whose datum is the variable name, or a sublist reference in case the term contains

more than one factor. Any sublist referenced will be a list of factors, or cells with a prefix value of 6, and whose data are variable names. Second, each variable name will be deleted from the structure, and replaced by its value (which we assume can be obtained by using the mythical function LOOKUP). Finally, the list structure is scanned, and a result is computed, which is the sum of the values of the terms, where a sublist term's value is given by the product of its data.

This program is presented purely for illustrative purposes, and will be left incomplete wherever completeness has no instructive value. Therefore, it has never been tested by a computer, and may have programming mistakes in it.

The preliminaries:

```
      IMPLICIT INTEGER(A-Z)
      DIMENSION SPACE(1000)
      DATA PLUS,STAR, BLANK/'+','*',' '
      DIMENSION CARD(80)
      CALL INIT(SPACE,1000)
```

A list of available space is now available to LPI, which contains 500 cells. Next, read the expression into CARD, one character per word.

```
      READ (5,1) CARD
1     FORMAT(80A1)
      I=1
      EXPR=ADD(EXPR1,1,2)
```

EXPR and EXPR1 are now markers to a list header. That list will be generated to contain a list of terms, each of which will be either a single variable name inserted with prefix value of 4, or a sublist inserted with prefix value of 1. EXPR will be

left as a marker to the list header, but EXPR1 will be used to mark the last cell of the list.

```
2     IF(CARD(I).NE.BLANK) GO TO 3
      I=I+1
      IF(I.GE.80) GO TO 100
      GO TO 2
3     SYMB=CARD(I)                    (skip over blanks)
4     I=I+1
      IF(I.GE.80) GO TO 99
      IF(CARD(I).EQ.BLANK) GO TO 4
      IF(CARD(I).EQ.STAR) GO TO 10
      EXPR1=ADD(EXPR1,SYMB,4)
```

Since the operator was + (assuming that either + or * must follow a variable name, if anything), we know that the term was a single variable, so we put it into the expression with a prefix value of 4. Note how EXPR1 is updated to mark the new "last" cell of the list.

```
      I=I+1
      GO TO 2
10    TERM=ADD(TERM1,0,2)
```

Note that the newly created list is to be used as a sublist, so its initial reference count is set to 0.

```
      EXPR1=ADD(EXPR1,TERM,1)
      TERM1=ADD(TERM1,SYMB,6)
```

The variable whose name is the value of SYMB was followed by a * operator, so we know that both it and the next variable are factors of the term. They are put into the sublist with a prefix value of 6, to distinguish them from terms.

```
11    I=I+1
      IF(CARD(I).EQ.BLANK) GO TO 11
      SYMB=CARD(I)
      TERM1=ADD(TERM1,SYMB,6)
12    I=I+1
      IF(I.GE.80) GO TO 100
      IF(CARD(I).EQ.BLANK) GO TO 12
      IF(CARD(I).EQ.STAR) GO TO 11
```

```
      I=I+1
      GO TO 2
```

If the operator is not *, then it must be +, so skip past it, and go back to statement 2 and get the next terms. Control drops to statement 99 if the last symbol should be a term, and needs to be added to the expression. If control passes to statement 100, then the symbol was a factor, and has already been added to the last term of the expression. Therefore, the statement 99 is the end of the input process, and statement 100 begins the substitution phase. We assume that we have a function LOOKUP, which has as a parameter a variable name, and returns as its value the number to be substituted for that variable in the expression.

```
 99   EXPR1=ADD(EXPR1, SYMB,4)
100   EXPR1=EXPR
```

Here, EXPR and EXPR1 will be used differently from above. We advance EXPR1, to check the next item of the expression, and if it is a sublist, the factors of the sublist must be checked. Otherwise, we use the fact that EXPR still marks the cell preceding the term just checked, and delete that term from the list, then add in its place the number to be substituted for it.

```
      SYMB=ADVANC(EXPR1,I)
      IF(I.EQ.2) GO TO 200     (if the substitution is finished)
      IF(I.EQ.1) GO TO 110
      I=LOOKUP(SYMB)
      X=DELETE(EXPR,X)
      EXPR=ADD(EXPR,I,4)
      GO TO 100
```

In case the term was a sublist, then we use TERM and TERM1 similarly, to substitute for the factors of the term.


```
110   TERM=SYMB
111   TERM1=TERM
      SYMB=DELETE(TERM1,I)
      IF(I.EQ.2) GO TO 112
      SYMB=LOOKUP(SYMB)
      TERM=ADD(TERM,SYMB,6)
      GO TO 111
112   EXPR=EXPR1
      GO TO 100
```

When control falls to statement 200, the substitution is complete. All that is left is the final computation.

```
200   SUM = 0
201   X=ADVANC(EXPR1,I)
      IF(I.EQ.2)  GO TO 300
      IF(I.EQ.1) GO TO 210
      SUM=SUM+X
      GO TO 201
210   TERM=X
      PROD=1
211   X=ADVANC(TERM,I)
      IF(I.EQ.2) GO TO 212
      PROD=PROD*X
      GO TO 211
212   SUM=SUM+PROD
      GO TO 201
300   X=ERASE(EXPR1)
```

SUM contains the desired result. Note that the entire list structure is returned to the list of available space by the one ERASE command.

APPENDIX C

LPI ERROR HANDLING

LPI makes several checks on the parameters of the LPI functions, hence there are several possible conditions which might erroneously arise. When any of these occur, LPI terminates the job with an ABEND macro-instruction. Following is the list of user completion codes which might occur in such an ABEND dump, and a description of the error condition each signifies.

User Code	Condition
4	The ADD function was invoked with the prefix value unequal to 2, and a marker value which does not indicate a legal cell address.
8	The ADD function was invoked with a prefix value greater than 255.
12	The ADD function was invoked with a prefix value of 1, but the value of the datum is not a valid sublist reference.
16	The ERASE program was invoked with an argument whose value is not a list reference.
20	The list of available space has been exhausted.