ED 066 878 EM 010 158

AUTHOR	Leaf, William A.
TITLE	SLIP: A Symmetric List Processing Language in PL-I.
INSTITUTION	Educational Testing Service, Princeton, N.J.
REPORT NO	RB-71-52
PUB DATE	Sep 71
NOTE	74p.
EDRS PRICE	MF-$0.65 HC-$3.29
DESCRIPTORS	Computer Science; *Data Processing; *Electronic Data Processing; *Programing Languages
IDENTIFIERS	PL 1; SLIP; *Symmetric List Processing

ABSTRACT

SLIP (Symmetric List Processing) is a list processing system designed to be added to a higher order language (PL-1 in this version) so that the user has available to him list processing powers. The primary value of such a system is its data handling power. Through SLIP, one can set up lists of data, scan those lists, alter them, and read or write them via external devices with minimal concern for space allotment, data types, or data structure organization. It is possible, for example, to write general programs which create and manipulate list structures whose shape, size, and contents are completely defined only during execution, by the shape, size, and contents of the data. SLIP exists as a set of library subroutines which do the actual manipulations. Thus the user simply writes a normal PL-1 program in which some statements refer to SLIP functions. These subroutines are explained here. (Author/JK)

RB-71-52

RESEARCH BULLETIN

SLIP: A SYMMETRIC LIST PROCESSING LANGUAGE IN PL-I

William A. Leaf
Carnegie-Mellon University

Educational Testing Service
Princeton, New Jersey
September 1971

SLIP: A Symmetric List Processing Language in PL-I

William A. Leaf

Carnegie-Mellon University

Introduction

This description of SLIP is written for the reader who has at least a basic knowledge of PL-I terminology, conventions, and programming.

Since this SLIP (and PL-I) are written for IBM-360 computers, the machine-oriented details in the following pages are 360 details. As much detail as possible has been provided so that the interested reader can understand the physical representation of SLIP cells and lists. The user interested only in the features of the language can skip such hardware information without loss of understanding.

The version of SLIP described here is written for the OS version of PL-I, specifically version 4, release 17 of the PL-I(F) compiler. Only minor changes in the routines are necessary to adapt them to other releases or versions of the (F) compiler.

SLIP: A Symmetric List Processing Language in PL-I

William A. Leaf*

Carnegie-Mellon University

General Features of SLIP

SLIP is a list processing system originally developed by Joseph Weizenbaum (1963). It is designed to be added to a higher order language, PL-I in this version, so that the user has available to him, in addition to the host algebraic language, list processing powers similar to those of languages like IPL-V, LISP, or FLPL.

The primary value of such a system is its data handling power. Through SLIP, one can set up lists of data, scan those lists, alter them, and read or write them via external devices with minimal concern for space allotment, data types, or data structure organization. It is possible, for example, to write general programs which create and manipulate list structures whose shape, size, and contents are completely defined only during execution, by the shape, size, and contents of the data.

SLIP exists as a set of library subroutines which do the actual manipulations. Thus the user simply writes a normal PL-I program in which some statements refer to SLIP functions. The user has access to all the normal facilities of PL-I and, in addition, has the power to create and manipulate data lists.

Weizenbaum orginally wrote SLIP to be imbedded in FORTRAN; since then, it has been adapted to be used with other algebraic languages (e.g., MAD; see Johnson, Rosin, & Leaf, 1967) and has been adapted at least once to PL-I (Johnson, 1968).

*This work was done while the author was a Visiting Research Psychologist at Educational Testing Service.

The present version of SLIP differs from the earlier versions in many respects. The major reason for this is the large difference between FORTRAN and PL-I. The latter language has a much wider variety of data types than the former, and it seemed important to mirror this flexibility in SLIP.

Thus the form of data storage has been changed slightly to fit with PL-I conventions. Many of the functions in SLIP now do slightly different things, in different ways, than they did in the earlier versions. And the names of the functions have sometimes been changed, primarily to improve the mnemonics but also to take advantage of the fact that PL-I allows 7-character names.

SLIP's overall powers and purposes have not been changed, however. The basic features of the language can be briefly summarized under four headings: list creation and manipulation, list scanning, description lists, and list input-output.

List creation and manipulation. Lists may contain any number of data cells and sublists (which have all the flexibility of main lists). Lists may be created (via the function LIST), copied (COPYLST), or erased (ERASLST). New data may be added to a list (NEWTOP, NEWBOT) or they may replace old data (SUBTOP, SUBBOT, REPLACE). Old data may be retrieved (TOP, BOT, DATUM) or removed from lists (POPTOP, POPBOT, REMOVE). The entire contents of a list may be moved to another list (MOVEL, MOVER) or erased (EMTYLST).

List scanning. The functions described above are best for manipulating data at the top or bottom of a list. By means of SLIP's reader facility, however, one can scan through a list or its sublists for data, then remove, replace, or retrieve the data as desired. Readers are pointers which may be moved up or down a list (via the advance functions, ADVSTR, ADVSTL, ADVLNR, and ADVLNL) to find any kind of datum or a particular kind of datum.

A particularly useful function, SEARCH, will scan a list's data for a particular datum.

Description lists. It is often useful to attach to a list a set of descriptors which identify the nature of the list (its data types or its importance to the program). Each list may have a description list which contains such information in Attribute Dimension-Value pairs. Specialized functions exist which conveniently manipulate such descriptive information. For example, NEWVAL stores the value for a particular attribute, ITSVAL retrieves the current value, and NOVAL removes the attribute and its value from the description list.

List input-output. Often the most economical way to create complex lists is to prepare them as input for the running program. Two functions, READLST and PRNTLST, exist which can read and write lists of any complexity. The output from PRNTLST is readable by READLST, so that one can save lists from day to day or use one program to create a complex list for another.

Organization of SLIP Lists

Lists are made up of a main cell, or header, and any number of data cells (see below for a description of the data types which may be stored in cells). The cells are linked together linearly and symmetrically. That is, the header is linked to the first data cell, the first is linked to the second cell, . . . , the next-to-last cell is linked to the last cell, and the last cell is linked to the header. In addition, the header is linked back to the last cell and all other cells are linked back to the preceding cell on the list. Thus the keyword <u>symmetric</u>: from any point on the list, it is possible to use the chain of links to go either forward or backward to any other point on the list. (The terminology for locating cells on a list relative to other cells on the list can be confusing. In normal linear lists, data cells are either <u>above</u> or <u>below</u> each other, and the header cell is the <u>top</u> of the list. These terms are still used in SLIP, but since lists are symmetric, a conventional meaning for "above" and "below" must be agreed upon. Each cell has two links; conventionally, the one called "LNKR" (<u>link right</u>) points <u>forward</u> to the cell <u>below</u> the current cell. The <u>link left</u> ("LNKL") points <u>upward</u> or <u>backward</u> to the preceding cell. Exceptions occur around the header, but it is traditionally regarded as the top of the list, and the cell to the "left" of the header is regarded as the bottom of the list.)

Data cells may contain data such as numbers or bit or character strings. They may also contain pointers to other lists. Thus any list may have any number of sublists, which in turn may have sublists of their own, etc. A list may be its own sublist. Thus it is possible to create extremely complex list structures.

A list may also have a <u>description list</u>. In IPL-V, description lists were usually seen as containing descriptive information about the nature of the main list, information completely separate from its contents. In SLIP, one may use the description lists for the same purposes; there are specialized subroutines designed to conveniently create and scan such lists.

Or one can use a description list for any purpose desired; practically, description lists have all the flexibility of organization and use as other lists, including the ability to have description lists.

SLIP Cells

Figure 1 gives the schematic form for any SLIP cell. Because PL-I allows data of different physical lengths, the actual length of a cell is determined by its datum. Each cell is made up of a two-word "links" portion followed by the datum.

SLIP field	ID	L N K L			MARK	L N K R			datum
Byte	0	1	2	3	4	5	6	7	8 ... 4n-1

Figure 1. Storage representation of a typical SLIP cell.

The first byte of the cell (which always begins on a fullword boundary) contains the ID which identifies the type of datum contained in the cell; currently, ID values of 1 to 13 are meaningful (see below). The rest of the first word (bytes 1-3) is the link left which points to the next higher cell on the same list. The second word begins with a 1-byte MARK portion. It is unused by SLIP, but may be used by the programmer to identify particular cells--for example, during a searching or scanning operation. The MARK is initially set to 0; it may be given any value from 0 to 255. Bytes 5-7 of the second word (LNKR) contain the link to the next lower cell on the list. Both LNKR and LNKL are machine addresses corresponding to PL-I pointer variables.

ID	DATUM Type[a]	
1	fixed binary (15,0)	(default)
2	fixed binary (31,0)	(maximum)
3	float binary (21)	(default)
4	float binary (53)	(maximum)
5	fixed decimal (5,0)	(default)
6	fixed decimal (15,0)	(maximum)
7	float decimal (6)	(default)
8	float decimal (16)	(maximum)
9	character string ($\leq$ 256)	
10	bit string ($\leq$ 2048)	
11	name of sublist (pointer variable)	
12	header of list (8-byte datum) containing a description list name (pointer variable) and a reference counter (binary integer)	
13	reader cell	

Figure 2. Allowable SLIP data types and their corresponding SLIP IDs. While only codes 1-11 are strictly data, codes 12 and 13 are included in this table for completeness.

[a]The parenthesized numbers represent the length of the datum and, if present, the scale factor.

<u>Header cells (ID = 12)</u>. The datum portion of a header cell is made up of two words. The first word contains a pointer to the list's description list, if any. The second word is a binary integer whose value is the list's <u>reference counter</u>. The reference counter tallies the number of times the list has been mentioned on other lists as a sublist or a description list. The tally is extremely important in telling SLIP if it is all right to erase a list or if the list cannot be erased because it is referenced by other valid lists. (See <u>Space Managing</u> below for more details.)

<u>Reader cells (ID = 13)</u>. Reader cells are special application cells used by the reader mechanisms for scanning list structures. They cannot be used

themselves as list items. (See The Reader Facility below for a complete description.)

Data cells (ID = 1 to 11). PL-I has a very wide range of data types--binary and decimal integer and floating point numbers, complex numbers, pointer and offset variables, bit and character strings, and pound sterling values, for example--but a certain inflexibility in mixing data types. The 11 data types actually allowed in SLIP cells represent a compromise intended to give the user the freedom to use as many meaningfully different data types as possible. Some data types are omitted completely, such as offset variables and complex variables; they are the ones which seem to have the least value to SLIP programmers.

Binary fixed point numbers, in version 4 of PL-I, are all stored in full computer words regardless of precision. SLIP allows the default precision (15,0; ID = 1) and the maximum precision (31,0; ID = 2). (Precision figures in PL-I definitions are in terms of the base used. Thus a fixed binary number of default precision allows 15 binary places; this is approximately equivalent to the default precision of fixed decimal numbers, which is 5 decimal digits.)

Fixed decimal numbers in PL-I are stored internally as decimal digits plus a sign. Such numbers may vary in length from 1 byte (precision = 1) to 2 words (precision = 15). SLIP allows the default precision (5,0; ID = 5) and the maximum (15,0; ID = 6).

Floating point numbers, either binary or decimal, are all stored in the 360 as floating hexadecimal numbers of either 1 or 2 words in length. Because PL-I distinguishes between float binary and float decimal, both are allowed separately in SLIP. The default precisions of float binary (21; ID = 3) and float decimal (6; ID = 7) are equivalent; each is 1 word long. Also allowed are the maximum precisions for float binary (53; ID = 4) and float decimal (16; ID = 8); each is 2 words long.

Strings in PL-I consist of two parts: a two-word Dope Vector and the string itself. The Dope Vector contains the address of the string in the first word and the actual and maximum possible lengths of the string in the second word (the two lengths may be different in the case of variable length strings). Character strings are stored with 1 character per byte; bit strings are stored with 8 bits per byte. In a SLIP data cell, a string is stored with the Dope Vector in the first 2 datum words and the string itself in the following words. All strings are stored as if they are of variable length; the maximum length for any string is the smallest number of full words needed to store the actual string.

In data cells, strings may be up to 256 characters in length (ID = 9) or 2048 bits in length (ID = 10).

Sublist names (ID = 11) are actually pointer variables; these are 1 word long. For a pointer variable to be the name of a sublist, its value must be the address of the sublist's header.

SLIP cells are all made up of a round number of computer words, even when the datum does not require the entire space (as for ID = 5 and for many cases with ID = 9 or 10). The minimum number of words in a cell is 3 (ID = 1, 2, 3, 5, 7, and 11); the maximum possible length is 68 words (ID = 9 and 10) although the exact length of string cells is determined by the length of the strings.

The List of Available Space (AVSL)

SLIP cells are created from a list of available space which is set up by the programmer outside his own program area by calling the function INITAS.

Originally, the list contains 4096 bytes of core. It can be increased by 1024-byte increments, whenever the current space is completely used, by automatic calls for more space, up to the limits of the machine storage allocated to the user. After first calling INITAS, the user need concern himself no more with available space. To him, it appears that all the rest of his job partition space is available at once. The job is terminated if a request is made for more space when none is available or can be found. At that time, one of two messages is printed out:

```
**** INITAS COULD NOT ALLOCATE ANY SPACE FOR SLIP LIST STRUCTURE STORAGE.
**** GO.REGION WAS TOO SMALL.  PROGRAM TERMINATED.
```

if the program took up so much of the user's space that INITAS could not even allocate the initial partition, or

```
**** NO MORE SPACE COULD BE OBTAINED AFTER xxxxx BYTES HAD BEEN ALLOCATED
                                                          TO SLIP.
**** yyyyy BYTES WERE STILL FREE BUT IN SEGMENTS TOO SMALL TO BE USED.
**** PROGRAM TERMINATED.
```

if INITAS had been able to successfully find space at least once. xxxxx and yyyyy are numbers; yyyyy can be greater than zero and allocation can still fail if the bytes are in separate pieces each too small to hold the new cell. (If yyyyy = 0, that comment line does not appear.)

AVSL consists of one or more segments of core, in full-word denominations aligned on full-word boundaries, linked together. The first word of each slot contains the address of the first word of the next slot; the second word contains the length of the slot in bytes. (Slots of 4 bytes length have no second word; their short length is indicated by a minus sign in the first word.) Slots are ordered from longest to shortest.

A request for space from AVSL to create a new cell causes the smallest slot which is still large enough to hold the cell to be removed from AVSL. The bytes necessary for the new cell are taken from that slot, and any "leftover" bytes are returned to AVSL, taking their place in order by size. Cells to be erased, i.e., returned to AVSL, are put back in the list as are leftovers from cell allocations. In returning pieces of storage to AVSL, SLIP checks to see whether they are adjacent to slots already on AVSL. If so, they are combined into one long slot and returned to AVSL in that form.

Space Managing

In any list processing system, some provisions must exist for returning unused cells to the list of available space so that no bit of space is permanently lost. In SLIP, as much as possible of this erasing process is handled by the system--both to relieve the user of needless space managing chores and to guard against the possibility of errors.

Removing data cells presents no problem, since a data cell does not control other storage. Removing header cells, however, which happens whenever a list is erased, may cause problems. If, for example, list A is a sublist of list B, one must not erase A while B still exists. Erasing A would not alter the structure of B, which contains a cell pointing to the sublist A, and any subsequent effort to scan B's structure (e.g., using reader facilities) would fail when it tried to enter the "sublist" which no longer exists.

SLIP maneuvers around this difficulty by means of each list's reference counter. When a list (e.g., A) is created, its reference counter is usually 1. If A is made a sublist or the description list of another list, A's reference counter is increased by 1. If an attempt is made to erase A, via the ERASLST function, the reference counter of A is decreased by 1. Only if the new value of the reference counter is 0 or less is A actually erased. In the example, A would not be erased because it is a sublist.

If a list is actually erased, all its cells are returned to AVSL. In addition, an attempt is made to erase all of the list's sublists and its description list: the reference counters of those lists are reduced by 1 and any of the lists whose new reference values are 0 or less are physically erased.

Example 1.

The following short program illustrates the use of some SLIP facilities on a simple problem: dealing 200 blackjack hands from a complete deck.

Notes: (The numbers refer to statement numbers in DEAL)

2. NEWBOT is a generic function which in fact has entries for each allowable SLIP data type; only the entry for character data is needed in this program, so the full declaration for NEWBOT is not used.

3. LIST is also a generic function. Since only its first entry point is needed, only that one is defined.

4. This initializes available space for SLIP.

5. The input cards are set to give DECK 52 sublists, each representing a card; each sublist contains two elements--the first is the character string name of the card, e.g., CLUB.KING, and the second is the point value of the card, e.g., 10. To simplify things, aces are assumed to count 11.

6. N is the number of hands to be dealt.

7. This makes an empty SLIP list which will contain the cards from each deal.

9. Each card, when dealt into HAND, will be marked with an index number between 0 and 255 so that the card can't be dealt twice into the same hand.

11. RANDEL's value is the address of a randomly-selected element (card) on DECK.

12. DATUM sets CARD equal to the datum in the cell addressed by POINTR. Thus CARD is the name of the sublist with the card name and its point value.

13. Don't put the same card twice into the same hand.

15. CARDSTRING is given the character name of the card being dealt.

16. The card is actually "dealt" onto HAND.

19. Iterate until the hand's count is 21 or more.

21. Print the cards dealt in the complete hand and

22. print the total point value.

23. Empty the list in preparation for the next deal.

```
        DEAL: PROC OPTIONS(MAIN);


 STMT LEVEL NEST
   1                  DEAL: PROC OPTIONS(MAIN);
   2      1            DCL (DECK, HAND, POINTR, CARD) PTR, (N, I, K, COUNT, ITSCOUNT)
                          FIXED BIN (31), RANDEL RETURNS (PTR), MARK RETURNS (FIXED BI
                          (31)), NEWBOT GENERIC (NEWBOT9 ENTRY (PTR, CHAR(1) VAR)),
                          CARDSTRING CHAR (48) VAR, TOP RETURNS (CHAR(48) VAR),
                          BOT RETURNS (FIXED BIN (31)), READLST GENERIC (READLS1
                          ENTRY (PTR)), PRNTLST GENERIC (PRNTLS1 ENTRY (PTR));
   3      1            DCL LIST GENERIC (LIST1 ENTRY (PTR));
   4      1            CALL INITAS;
   5      1            CALL READLST (DECK);
   6      1              N = 200;
   7      1             CALL LIST (HAND);
   8      1            DO I = 1 TO N;
   9      1     1        K = MOD(I,256);
  10      1     1        COUNT = 0;
  11      1     1      LOOP: POINTR = RANDEL(DECK);
  12      1     1        CALL DATUM(POINTR,CARD);
  13      1     1        IF MARK(CARD) = K THEN GO TO LOOP;
  15      1     1        CARDSTRING = TOP(CARD);
  16      1     1        CALL NEWBOT (HAND, CARDSTRING);
  17      1     1        ITSCOUNT = BOT(CARD);
  18      1     1        COUNT = COUNT + ITSCOUNT;
  19      1     1        IF COUNT < 21 THEN GO TO LOOP;
  21      1     1        CALL PRNTLST(HAND);
  22      1     1        PUT DATA (COUNT);
  23      1     1        CALL EMTYLST(HAND);
  24      1     1      END;
  25      1            END DEAL;
```

Below is sample output from the program shown above.

```
( DIAMOND.ACE HEART.FOUR CLUB.QUEEN )
COUNT=                25;
( SPADE.EIGHT SPADE.KING DIAMOND.QUEEN )
COUNT=                28;
( HEART.TWO DIAMOND.THREE SPADE.JACK CLUB.NINE )
COUNT=                24;
( CLUB.TEN DIAMOND.JACK HEART.TEN )
COUNT=                30;
( CLUB.FIVE HEART.KING SPADE.TEN )
COUNT=                25;
( HEART.FIVE DIAMOND.QUEEN SPADE.FOUR HEART.JACK )
COUNT=                29;
( CLUB.SEVEN DIAMOND.QUEEN HEART.KING )
COUNT=                27;
( HEART.JACK HEART.KING DIAMOND.FIVE )
COUNT=                25;
( DIAMOND.ACE HEART.SIX CLUB.FOUR )
COUNT=                21;
( SPADE.SIX DIAMOND.THREE SPADE.KING CLUB.TEN )
COUNT=                29;
```

Facilities in SLIP

The following sections describe the types of things which can be done using SLIP. In each section, a general description and its examples are followed by a formal description of the functions introduced in the section. The index at the end of this paper allows one to easily locate the description of any particular function.

In the description for each function, the allowable calling sequences are listed; they include the list of possible variables. The names of the variables are intended to indicate the data type of the variables. In summary form, the list below shows the exact data type for the dummy variables used in the following sections.

LST, LST1, LST2, ORGL, COPY, COP2, HOST, DLST; PNTR, POINT, CELL, LINKR, LINKL; RDR, RDR2. These are all pointer variables. (In general, all main program variables which name lists or readers or otherwise address SLIP cells should be pointer variables.) LST through DLST are list names (i.e., they contain the addresses of list headers). PNTR through LINKL are cell addresses; in some cases, the cells may be headers which would make the variable a list name. RDR and RDR2 have the addresses of reader cells, which makes them reader names.

DATM, DAT1, DAT2, OLDDAT, ATTR, VALU, OLDVAL, NUVAL. These are allowable SLIP data, which may correspond to any of the data types with ID values between 1 and 11 in Figure 2.

DATID, ATID, VALID, LVL, INT, LNGTH, I, J. These are all binary integers. The first three are used as ID values and, in particular, should only have values between 1 and 11.

STRING may be any bit or character string.

FILENAM is the name of a stream input or stream output file.

X, Y, DX, and DY are floating point numbers. X and Y are one word long; DX and DY are two words long.

VAR may be any variable which does not have a Dope Vector (i.e., any nonstring, nonarray variable).

CODE is a binary integer with values between 0 and 9.

Initializing Available Space

In order to use SLIP functions, one must first set up the list of available space so that SLIP cells may be created and erased. This is accomplished by the statement "CALL INITAS;". In the program, this step must be executed only once and before any other SLIP functions are called. It is often easiest to make this the first executable statement in the program. (See Appendix A for a convenient way of insuring this.)

CALL INITAS; or CALL INITAS(INT);

This function makes available to SLIP all the unused machine storage in the programmer's region. It also initializes all new space to 0 except for the link words necessary to chain the segments of AVSL together. INITAS sets in motion the housekeeping aspects of SLIP which will generate new cells when required and erase old cells when no longer needed.

The argument INT, if included, is relevant to SLIP's error detection procedures (see Appendix B). SLIP normally notes errors during execution and continues processing; to keep from running indefinitely, SLIP tallies those errors. By default, the program is terminated after 50 such errors; the argument INT, if included, serves as the new cutoff criterion.

Creating and Modifying Lists

The functions to be described in this section perform the basic functions of creating lists, adding, subtracting, and replacing elements, retrieving data, reshaping lists, and erasing lists.

Example 2.

Certain models of memory taken from psychology utilize short-term memory banks; these may be represented by SLIP list structures. Assume for the following segments of program that MEMORY is a list whose elements are the items currently held in short-term memory (e.g., CVC trigram pairs in a paired associates learning task). The code shown will move new pairs into memory and remove "forgotten" pairs according to three different schemes.

A. Assume an infinite memory in which no old pairs are forgotten.

```
...
CALL NEWTOP(MEMORY, CVCONE || '=' || CVCTWO);
...
```

Or, if one wishes to place the new pair at a random position on the list,

```
...
N = NUMBEL(MEMORY) + 1;
IF IRAND(N) = 0 THEN PLACE = MEMORY;
ELSE PLACE = RANDEL(MEMORY); /*PLACE is a pointer variable */
CALL NEWTOP(PLACE, CVCONE || '=' || CVCTWO);
...
```

B. If memory is of fixed length and the pair to be replaced by the new pair is the oldest one, the following code will add new pairs to the top of MEMORY and, if memory is full, remove the old pairs from the bottom.

```
...
CALL NEWTOP(MEMORY, CVCONE || '=' || CVCTWO);
IF NUMBEL(MEMORY) > N THEN CALL POPBOT(MEMORY);
...
```

C. If the memory is of fixed maximum length, the following code will replace randomly selected old pairs with new pairs when the memory is full.

```
...
IF NUMBEL(MEMORY) < N THEN CALL NEWTOP(MEMORY,CVCONE || '=' || CVCTWO);
ELSE CALL REPLACE(RANDEL(MEMORY),CVCONE || '=' || CVCTWO);
...
```

In examples 2B and 2C it is possible to retrieve the value of the erased pair by using POPBOT or REPLACE as functions; e.g.,

```
OLDPAIR = POPBOT(MEMORY);
```

if OLDPAIR is a character string variable and one uses a slightly unorthodox declaration for POPBOT or REPLACE, e.g.,

```
DCL POPBOT RETURNS (CHAR(20) VAR);
```

CALL LIST(LST); <u>or</u> LST2 = LIST(LST); <u>or</u> LST = LIST(9);

LIST creates an empty list and in each of the statements above assigns to LST the name of the new list (i.e., places the address of its header cell in LST). In the second version, LST2 is also given the name of the list.

If the argument is a pointer variable, the reference counter of the list is set to 1. In that case, the list can be erased only by an explicit call to ERASLST. If the argument is (the decimal constant) 9, the reference counter is set to 0. The list may be automatically erased if, for example, it is the sublist of another list which is erased.

CALL COPYLST(ORGL, COPY); <u>or</u> COP2 = COPYLST(ORGL, COPY);
<u>or</u> COPY = COPYLST(ORGL);

ORGL, COPY, and COP2 are pointer variables.

COPYLST creates an exact copy of the entire list structure named ORGL and assigns the name of the copy to COPY and, if appropriate, COP2. If COPY is already the name of a list, the contents of ORGL are placed below the current contents of COPY.

COPYLST makes copies of all sublists and description lists in the list structure of ORGL; thus the original and its copy are physically distinct. The topography of the original is repeated in the copy; if, for example, the original references the same sublist twice, the copy references a single copy of the sublist twice.

CALL EMTYLST(LST); or LST2 = EMTYLST(LST);

LST is made into an empty list; its data cells are returned to AVSL, including sublists where appropriate. The description list of LST, if there is one, is not erased. LST2 is given the name of the now empty list (i.e., is made equal to LST).

CALL ERASLST(LST); or LVL = ERASLST(LST);

ERASLST first decrements the reference counter of LST by 1. If the counter's new value is greater than 0, indicating that LST is still the sublist or description list of some existing list, nothing further is done.

All levels of LST are removed--top level, its description list, any sublists, their description lists, their sublists, etc.--subject to the same restrictions that apply to the main list: lists with a new reference level of more than 0 are retained.

If called as a function, ERASLST returns a binary integer whose value is the new value of LST's reference counter. If LVL = 0, the list has been erased.

```
CALL NEWTOP(LST, DATM); or PNTR = NEWTOP(LST, DATM);
CALL NEWBOT(LST, DATM); or PNTR = NEWBOT(LST, DATM);
```

NEWTOP (NEWBOT) creates a new SLIP cell containing DATM and places that cell at the top (bottom) of the list named LST. All other cells of the list are unchanged; the effect is of putting a new link in a chain without breaking or rearranging any other links.

DATM may be any of the 11 data types which may be stored on lists; NEWTOP and NEWBOT are generic functions with separate entries for each data type (see Appendix C).

PNTR is a pointer variable which, if used, is given the address of the newly created cell.

The first argument "LST" for either function may point at a cell on a list rather than name the list; in that case, the new cell is placed to the right of the addressed cell (for NEWTOP; to the left for NEWBOT).

```
CALL SUBTOP(LST, DATM [, ID] [, OLDDAT]);
CALL SUBBOT(LST, DATM [, ID] [, OLDDAT]);
CALL REPLACE(PNTR, DATM [, ID] [, OLDDAT]);
```

All three functions remove a data cell from a list and replace it with a new cell containing DATM. (DATM must be an allowable SLIP data type; LST and PNTR are pointer variables; and ID, if present, is a binary integer whose value is the SLIP ID number corresponding to the data type of DATM. If ID is omitted, it is assumed that DATM is the same type as the datum of the erased cell.)

OLDDAT, if present, is given the value of the datum of the erased cell. OLDDAT must be of the appropriate type; no conversion is made.

If LST names a list, SUBTOP replaces tne top data cell on the list and SUBBOT replaces the bottom cell. As in the case of NEWTOP, LST may be the address of any list cell: the only restriction is that the removed cell may not be the list's header.

REPLACE replaces the cell addressed by PNTR; that cell may not be a list header.

Programming note. In programs in which SUBTOP, SUBBOT, or REPLACE is used to replace a single data type, it may be declared and used as a function which returns that datum type. For example, if only character string data are replaced, one can say DCL SUBTOP ENTRY (PTR,,FIXED BIN (31)) RETURNS (CHAR (20)VAR); and then use statements like OLDSTR = SUBTOP (LST, NEWSTR, 9).

```
CALL TOP(LST, DATM);
CALL BOT(LST, DATM);
CALL DATUM(PNTR, DATM);
```

LST is usually the name of a list, although it may be the address of any list cell. PNTR is the address of any cell except a list header. DATM is set to the value of the datum in the cell to the right of that addressed by LST (for TOP) or to the left of that addressed by LST (for BOT), or in the cell addressed by PNTR (for DATUM).

Programming note. In programs in which TOP, BOT, or DATUM is used only for a single data type, it may be declared and used as a function which returns that datum type. Two instances of this are shown in Example 1 above.

```
CALL POPTOP(LST[, DATM]);
CALL POPBOT(LST[, DATM]);
CALL REMOVE(PNTR[, DATM]);
```

These three functions work like TOP, BOT, and DATUM and in addition erase the cell in question. Because one may wish to remove a cell without being given its datum, the DATM argument may be omitted. If DATM is present, it must be of the same data type as the datum in the cell.

When the cell is removed from a list, the remainder of the list is linked around the gap. The cell to be removed may not be a list header.

Programming note. If POPTOP, POPBOT, or REMOVE is being used only to return a single datum type, it may be declared to return that datum type and used in assignment statements, e.g., ZORCH = POPTOP(LST);.

```
CALL MOVER(LST1, LST2);
CALL MOVEL(LST1, LST2);
```

MOVER and MOVEL move the entire contents of LST2 onto LST1. MOVER places the transferred cells at the bottom of any prior contents of LST1; MOVEL puts the moved cells at the top of LST1, above its prior contents.

If LST2 is the address of a cell on a list rather than a list name, then the cells moved are the ones between the addressed cell and the bottom (for MOVER) or the top (for MOVEL) of the list containing the cell.

Input/Output Facilities for Lists

The functions described in the preceding section are useful for making delicate and relatively small-scale modifications to lists. For creating large or complicated lists, it is much more efficient to read the list from a stream input file by means of READLST.

Also discussed in this section are the specialized functions for outputting lists or cells onto stream output files. They make it possible to conveniently display list structures--either in a form which may subsequently be reread by the input function (through PRNTLST) or in a form with more list structure information for debugging (through PUTLIST and PUTDATM).

CALL READLST(LST [,'BREAK=x'] [, $\{\begin{smallmatrix}\text{'B'}\\\text{'T'}\end{smallmatrix}\}$] [, FILENAM]);

READLST is a general function for reading list structures from stream input files. The default file is SYSIN; this may be overridden by using the FILENAM argument to indicate the desired file. READLST creates a new list, assigns the name to LST, and makes its contents the structure given in the input file. If LST already named a valid list, that list is erased.

The input format consists of a left parenthesis to indicate the beginning of the list, followed by the elements of the list, followed by a right parenthesis to end the list. The parantheses and the elements are separated from each other by one or more break characters; thus parentheses may be parts of elements if joined on either or both sides by nonbreak characters. (The default break character is the blank, but may be altered to any other character except (,), or " by the argument 'BREAK=x', where the substitute for x is the new break character.) Sublists are entered in the same format within the delimiting parentheses for the higher list. Description lists are entered as sublists except they are preceded by "DLIST:" which serves as an identifier and is not interpreted as a list element.

For example,

```
( DLIST: ( CONTENTS NOUNS NUMBER SINGULAR ) HORSE COW ( DLIST: (
TYPE DOGS ) DALMATION BEAGLE MUTT )  )
```

would produce a main list containing "HORSE" and "COW" and a sublist with dog names; each list would have a description list.

All allowable SLIP data types may be read in; for each element, the decision of which data type to assign is based on the element itself and whether or not either option 'B' or option 'T' is selected. If option 'T' (for text) is chosen, all elements are assumed to be character string variables (ID = 9). The table below indicates the ID selections for the default condition and for 'B' (for binary).

Input Element Form	ID Assignment		Examples
	default	'B'	
[±]dddddd	1 or 2	5 or 6	123 +57 1011101 -999999
[±]bbbbbbbbB	1 or 2	1 or 2	-1011101B 1B 0B
[±]ddd.ddd	7 or 8	7 or 8	.01 1.0 -.00001 +123.
[±]bbbb.bbbbB	7 or 8	3 or 4	+.01B 11.01B 1010101.B
'bbbbbb'B	10	10	'100'B '1'B '0'B
(all else)	9	9	WORDS '101' .833E+15

Note.--"d" stands for any decimal digit 0-9; "b" stands for either binary digit, 0 or 1.

The choice between adjacent codes 1 and 2, 3 and 4, 5 and 6, and 7 and 8 depends on the length of the number. If a fixed decimal number has 5 digits or less, or a float decimal number has 6 significant digits or less, or a fixed binary number has 15 bits or less, or a float binary number has 21 significant bits or less, it is given the ID of the shorter precision. If numbers are longer than 15 or 16 digits or 31 or 53 bits (fixed and floating point, respectively) the most significant extra places are lost. Bit strings (ID = 10) may contain up to 2048 bits. Character strings may have up to 256 characters.

The data representations correspond to the form for constants written in PL-I programs with one exception: character strings are stored _in toto_; delimiting single quotes (') are not necessary and, if present, are stored as part of the string. E-format numbers are not allowed; they would be interpreted as character strings.

The binary option ('B') allows one to read fixed or float decimal or binary. Because it is awkward to write binary numbers and because the 360 performs fixed decimal arithmetic slowly, the default option assumes that any number should be stored as either fixed binary or float decimal. (These are the default PL-I data types also.)

It is sometimes handy to create list structures in which a single list is a sublist at two or more points--or is its own sublist. To read such lists, one can indicate each repeated list by assigning it a distinct number the first time the list is read and simply repeating the number whenever the list is to be rereferenced. The numbers need not form any sequence.

For example,

```
("83" THIS LIST REPEATS  ( THIS ONE DOESN'T ) ("83"
) DLIST: ("83"  )    )
```

The main list is its own sublist and its own description list. Note that unrepeated lists need no index number. Note also that there is no space separating the left parenthesis from the index, but there is one between the index and the right parenthesis.

In record-oriented I/O, it has been customary for lists to begin in the first column of a card or record. No such restriction is relevant to stream I/O. In the current implementation of READLST, the input file is read in 80-character segments. A new segment is read at the start of each call to READLST. Therefore one must begin each list beyond the 80-character block which contained the end of the previous list or after the end of previous data read by a GET statement. To be safe, <u>allow at least 80 characters before the start of any list</u>. (Or, if one is really using card input with only lists in the input stream, it is more convenient and equally reliable to start each list on a new card.)

<u>Notes</u>. The characters between the end of one list and the start of another need not be break characters; they may be anything except left parentheses, such as phrases identifying the contents of the coming list.

READLST will scan only three 80-character segments for the start of a list. If it has not found a left parenthesis in those 240 characters, READLST will print an error message and return control without creating a new list.

CALL PRNTLST(LST [, 'BREAK=x'] [, 'B'] [, FILENAM]);

PRNTLST prints the contents of the list structure named LST on an output file--either SYSPRINT or, if the FILENAM argument is present, FILENAM.

The output format is identical to that required by READLST: lists are delimited by parentheses; description lists are printed as sublists with the label DLIST: in front of the left parenthesis, etc. The default break character is the blank; it may be changed to any desired value except (,), or " by using the 'BREAK=x' argument.

If the argument 'B' is omitted, i.e., the default is chosen, numbers are printed as either fixed or float decimal values. If 'B' is chosen, data with ID = 5-8 are printed as decimal numbers and data with ID = 1-4 are printed as fixed or float binary numbers. Character and bit string data are printed as indicated for READLST under either option. Since binary numbers are difficult to interpret visually, one should prefer the default printing style unless he wants to preserve the distinction between binary and decimal machine representation.

Within the list structure, if a list is referenced more than once, it is given an index number for the first appearance of the list; the index number alone appears for each additional reference to the list. The number is, in fact, the name of the list--the address of its header--printed in hexadecimal.

CALL PUTLIST(LST);
CALL PUTDATM(PNTR);

PUTLIST and PUTDATM are intended more as debugging aids than as convenient list output devices. PUTDATM prints the contents of the single cell addressed by PNTR, which may be the name of a header cell as well as of any data cell. PUTLIST prints the list named by LST, one cell per printed line, including sublists but excluding description lists.

Printout for each cell consists of its ID, LNKL, LNKR, and the datum; for a header cell, this includes two items: the name of the description list and the reference counter.

All items which are addresses--LNKL, LNKR, and sublist and description list names--are printed as hexadecimal constants. Other data are printed in forms consistent with the data types.

The Reader Facility

In order to take full advantage of the power of list processing, the user must have some general means of scanning lists whose exact structure could not be known during the programming. Readers provide such a mechanism. A reader is essentially a pointer which can be moved through a list structure to find certain data or data types, can be backed up or moved forward, and can perform these activities in a list structure of any degree of complexity.

Readers are specialized SLIP cells made up of 4 words organized as shown below:

Reader field	ID	LINK	LVLCNT	LOFRDR	CELLPNT
Bytes	0	1-3	4-7	8-11	12-15

The ID for such cells is 13. In place of the usual LNKL is a LINK address which points to the next reader cell on the stack, if any (a new reader cell is generated for each level descended into the list's sublist structure). In place of MARK and LNKR is the integer LVLCNT, which is the number of levels into the list structure the reader has descended as well as the number of additional cells in the reader stack. LOFRDR is the name of the list (or sublist) currently being scanned, and CELLPNT is the address of the actual cell being pointed to.

Example 3.

This example shows two slightly different procedures for counting the sublists in a list structure. The first counts only the sublists on the main list. LST is the name of the list, RDR a pointer variable which will be the name of the reader, and FLAG is a bit string 1 bit long.

```
     ...
     RDR = RDROF(LST);               A reader is created for LST and RDR is
                                     given the address of the reader cell.
                                     Initially the reader points to LST's
                                     header.
     N = 0;
     I = 11;
LOOP:  CALL ADVLNR(RDR,FLAG,I);      The reader is advanced to the first cell
                                     on the main list which contains a sublist
                                     name--i.e., has an ID of 11. FLAG = '1'B
                                     if a sublist name was found; '0'B if none
                                     was found. ADVLNR (advance linearly to
                                     the right) moves the reader down the list
                                     but does not move it into sublists.
     IF ~FLAG THEN GO TO NEXT;
     N = N + 1;
     GO TO LOOP;
NEXT:  CALL ERASRDR(RDR);            At this point N = the number of sublists
     ...                             in the main level. ERASRDR erases the
                                     reader cells since there is no further
                                     need for the reader.
```

The second version counts all of the sublist name cells in the entire list structure. CELL is another pointer variable; MARK and SETMARK are functions for testing and changing the MARK portion of SLIP cells.

```
      ...
      RDR = RDROF(LST);
      N = 0;
      I = 1;
 LOOP:  CALL ADVSTR(RDR,I,FLAG);     ADVSTR (advance structurally to the right)
                                     will move the reader through LST's entire
                                     structure, entering sublists as they are
                                     encountered.

      IF ~FLAG THEN GO TO NEXT;      Quit when the scan has gone through the
                                     entire structure, as before.

      CELL = CELLPNT(RDR);           CELL is given the address of the cell
                                     actually pointed to by the reader.

      IF MARK(CELL) = 0 THEN DO;     As each sublist cell is counted, it is
        N = N + 1;                   marked so that it won't be counted twice--
        CALL SETMARK(CELL,I);        as it would, for example, if the same
      END;                           sublist was referenced twice and it had
      GO TO LOOP;                    one or more sublist cells.

 NEXT:  IF N > 0 THEN DO;            Erase any marks that were made during
 RELOOP:  CALL ADVSTR(RDR,I,FLAG);   the counting process.
          IF FLAG THEN DO;
            CALL SETMARK(CELLPNT(RDR)
                              ,0);
            GO TO RELOOP;
          END;
      END;
      CALL ERASRDR(RDR);
      ...
```

RDR = RDROF(LST);

If LST is the name of a list, then RDROF creates a reader cell for that list and assigns the address of the cell to RDR. The reader initially points to the header of LST--i.e., its LOFRDR and CELLPNT sections both contain the address of the list.

RDR2 = COPYRDR(RDR1);

If RDR1 is the name of a reader, a copy of the entire reader stack is made and RDR2 is made the name of the copy. This function is useful in cases in which one may want to make several different scanning passes at a list after reaching some point in the list. The copy insures that one can always come back to the same point for the next scan.

```
CALL ERASRDR(RDR); or LVL = ERASRDR(RDR);
```

ERASRDR erases the reader named by RDR; all the cells in the reader stack are returned to AVSL. If LVL is present, it is set to the LVLCNT of the reader--i.e., how many lists deep into the list structure it had descended.

```
CALL ADVSTR(RDR [,DATID] [,FLAG]); or PNTR = ADVSTR(RDR [,DATID] [,FLAG]);
CALL ADVSTL(RDR [,DATID] [,FLAG]); or PNTR = ADVSTL(RDR [,DATID] [,FLAG]);
CALL ADVLNR(RDR [,DATID] [,FLAG]); or PNTR = ADVLNR(RDR [,DATID] [,FLAG]);
CALL ADVLNL(RDR [,DATID] [,FLAG]); or PNTR = ADVLNL(RDR [,DATID] [,FLAG]);
```

These functions perform the actual scanning operations. They may move the reader to the right (downward) via ADVSTR or ADVLNR, or to the left (upward) via ADVSTL or ADVLNL. If it encounters sublists on the list being scanned, the reader may move structurally into the sublist via ADVSTR or ADVSTL or it may continue linearly along in the same list via ADVLNR or ADVLNL.

If called in assignment statements, the reader advance subroutines give PNTR the address of the cell advanced to, unless the advance fails to find a proper datum cell, in which case PNTR is set to NULL.

In the full argument list, RDR is a pointer variable naming the reader, FLAG is a bit string of length 1 whose value is set to '1'B if an acceptable datum is found and '0'B if none is found, and DATID is a binary integer with values between 1 and 11 to indicate the type of data cell to be found. FLAG and/or DATID may be omitted from the calling sequence. If DATID is omitted, any data cell may satisfy the scan.

If RDR is advanced by ADVLNR or ADVLNL, the following actions take place: (a) CELLPNT is made equal to the address of the next cell to the right (left) of the one it was pointing to. (b) If the new cell is the header of the list (i.e., LOFRDR = CELLPNT), FLAG (if present) is set to '0'B, and the function returns. (c) Or, if the ID of the new cell = DATID or if DATID was omitted, FLAG (if present) is set to '1'B; PNTR (if present) is given the cell's address, and the function returns. (d) Otherwise, an acceptable cell hasn't been found, but the list has not been completely scanned, so step (a) is repeated.

If RDR is advanced via ADVSTR or ADVSTL, the same basic scanning process occurs except for modifications which allow RDR to scan the entire list structure rather than just the top level. (e) When the reader is advanced to a cell containing a sublist name (and that isn't the desired datum type), a new reader cell is created for the sublist (pointing initially to the header of the sublist). The scan then continues in that sublist, searching <u>its</u> cells for the type of datum desired. (This process may repeat endlessly, allowing sublists of sublists of sublists, etc., to be scanned.) (f) If the scan fails in a sublist (i.e., lands back on the header without finding a proper target), the reader cell which was created for that sublist is erased and the scan continues, in the list one level higher, with the cell one to the right (left) of the one which had named the sublist.

No matter how simple or complex the past scanning of a reader, it may be advanced further by any of the four ADV___ functions. In particular, there is no problem with advancing a reader via ADVLN_ which had been left pointing in a sublist of the main list by a previous ADVST_ call. (In such cases, the reader acts like a linear reader of the given sublist.)

Note. ADVSTR and ADVSTL have built-in protection against list structures which loop back on themselves. By SLIP standards, for example, it is fine to construct a structure of lists A and B such that each is a sublist of the other. In such a structure, an unsophisticated reader could enter sublists indefinitely without ever running out of sublists, since it would be tracing a loop. Therefore, when ADVSTR or ADVSTL is asked to enter a sublist, it checks the reader's stack to make sure the sublist is not one which, at a higher level, RDR is already scanning. If this is true, RDR is advanced linearly to the next cell beyond the one with the sublist name, just as is normally done by ADVLNR and ADVLNL.

CALL INITRDR(RDR); or RDR2 = INITRDR(RDR);

INITRDR initializes the reader within the list currently being scanned--that is, the reader is set pointing at the header of whatever list or sublist is currently being scanned. (If the reader was already pointing to the header, no change occurs.) RDR2, if present, is given the name of the reader.

CALL LVLRVT(RDR); or RDR2 = LVLRVT(RDR);
CALL LVLRVT1(RDR); or RDR2 = LVLRVT1(RDR);

These functions move a reader back up a list structure. The reader may back up one level (with LVLRVT1) or as many levels as necessary to reach the top level (with LVLRVT). (Each step of backing up is accomplished by popping the top cell off the reader, leaving the reader pointing to the cell from which it had entered the lower sublist.) The reader is left pointing to the cell of the proper level from which the reader had descended. If the reader was somewhere in the top level when either function was called, there would be no action taken. RDR2, if present, is set to the name of the reader.

Programming note. These functions, along with INITRDR, are useful for reinitializing a reader without having to erase it and then create a new one. The statement CALL INITRDR(LVLRVT(RDR)); first brings the reader back into the main list and then makes it point to the header; this is exactly as the reader was when first formed.

CALL REED(RDR,DATM);

REED sets DATM equal to the datum of the cell currently pointed to by RDR. It assumes the data types match. (If the reader points to a list header, no assignment is made.)

Programming note. If REED is only used to assign a single data type, it may be declared as a function which returns that type and then used in statements like DATM = REED(RDR);.

INT = LVLCNT(RDR);

INT is given the LVLCNT value for the reader named RDR. If the reader is in the top level of the list, INT = 0; if the reader is in a sublist, INT = 1; if the reader is in a sublist of a sublist, INT = 2; etc.

LST1 = LOFRDR(RDR);

LST1, a pointer variable, is given the name of the list currently being scanned by RDR. This may be the list for which RDR was originally created, if LVLCNT = 0, or it may be the name of any of the list's sublists.

CELL = CELLPNT(RDR);

CELL is given the address of the cell currently being pointed to by the reader.

CALL SEARCH($\left\{ {\text{LST} \atop \text{RDR}} \right\}$, DATM, DATID [, PTR]);
<u>or</u> FLAG = SEARCH($\left\{ {\text{LST} \atop \text{RDR}} \right\}$, DATM, DATID [, PTR]);

SEARCH scans the list LST for an occurrence of DATM, which is a datum of the type with ID = DATID. If the search is successful, the argument PTR (if present) is given the address of the cell in which the datum was found and FLAG (if present) is set to '1'B. If the search fails, PTR is set to NULL and FLAG = '0'B.

The search always proceeds by means of a reader advanced by ADVSTR; thus if DATM is on LST or any of its sublists the search succeeds.

If the first argument is LST, the name of the list, then the search begins at the top of the list. If one wanted to search for different occurrences of the same datum, for example, an alternative exists. One can create a reader for the list (e.g., RDR = RDROF(LST);) and then use the reader as the first argument in the calling sequence. Then SEARCH scans LST from the cell at which the reader was pointing. The following code, for example, will count the occurrences of THE on LST:

```
     ...
     N = 0;
     RDR = RDROF(LST);
LOOP: FLAG = SEARCH(RDR, 'THE', 9);
     IF FLAG THEN DO;
       N = N + 1;
       GO TO LOOP;
     END;
     CALL ERASRDR(RDR);
     ...
```

<u>Description List Functions</u>

Any list may have a description list, and that description list may have any structure desired. Classically, however, description lists have been thought of as lists of <u>attribute dimensions</u> plus <u>values</u> for those dimensions which define or describe the main list. For example, if a list is NEWCAR, it might have these attributes and values: MAKE, STUTZ-BEARCAT; PRICE, 8350; YEAR, 1970; STYLE, CONVERTIBLE; COLOR, BURGUNDY; etc.

This type of description list is organized as a linear list with adjacent cells representing attribute-value pairs. The top cell of the list is the name of the first attribute dimension (e.g., MAKE), the cell below that is the list's value on that dimension (e.g., STUTZ-BEARCAT), etc. The functions described below are used to add, remove, and locate attributes and values on such a list.

Example 4.

This example shows the use of description lists in the context of another card problem. The procedure PLAYHI is intended (for bridge, say) to play a card on a trick to which other cards have already been played. From HAND, PLAYHI must select a card from the right suit and make it either the highest card in the hand's suit, if it will beat the cards already played, or the lowest card in the suit.

Let us (and PLAYHI) assume that HAND contains 4 sublists, one per suit, each having whatever cards of that suit the hand contains arranged by rank, with the highest card at the top. Let each sublist also have a description list with the following attribute pairs: NAME, (SPADES or HEARTS or DIAMONDS or CLUBS); NUMBER.OF.CARDS, (#); and RANKS (sublist with integers from 1 (deuce) to 13 (ace) for each of the cards the suit has).

```
PLAYHI: PROC (HAND, SUITNAM, RANKHI) CHAR (20) VAR;
     DCL (RDR, HAND, PSUIT, PNTR) PTR, RDROF RETURNS (PTR),
       (NAME, SUITNAM) CHAR (8) VAR, RETN CHAR (20) VAR,
       RANKHI FIXED BIN, NEWVAL ENTRY
       (PTR,,FIXED BIN,,FIXED BIN), ITSVAL ENTRY (PTR,,FIXED BIN,),
       TOP RETURNS (FIXED BIN), (POPTOP, POPBOT) RETURNS (CHAR (20)
       VAR), ADVLNR GENERIC (ADVLNR4 ENTRY (PTR) RETURNS (PTR)),
       CONTS RETURNS (PTR), SUIT PTR BASED (PSUIT);

     RDR = RDROF(HAND);             PSUIT is set pointing to the datum of
LOOP:PSUIT = ADVLNR(RDR);           the new cell; thus SUIT becomes the
     PSUIT = CONTS(PSUIT);          name of the next suit sublist.

     CALL ITSVAL(SUIT, 'NAME',9, NAME);     NAME is given  the SUIT's
                                            value on the attribute NAME.

     IF NAME~ = SUITNAM THEN GO     Scan until correct suit is found.
                         TO LOOP;
     CALL ITSVAL(SUIT,'NUMBER.OF.   N gets the integer number of cards in
              CARDS',9,N);          the suit.
     IF N = 0 THEN DO;
       RETN = '';                   If no cards, return the null string
       GO TO SCOOT;                 to tell the main program so.
     END;
     CALL NEWVAL(SUIT,              Decrement the number of cards value to
       'NUMBER.OF.CARDS',9,N-1,1);    compensate for the card removed below.

     CALL ITSVAL(SUIT, 'RANKS',9,   PNTR is made the name of the sublist
                         PNTR);     with the card ranks.
     N = TOP(PNTR);
     IF N > RANKHI THEN DO;         Play a card to try to win the trick.
       CALL POPTOP(PNTR);
       RETN = POPTOP(SUIT);
     END;
     ELSE DO;                       Play the lowest card in the suit.
       CALL POPBOT(PNTR);
       RETN = POPBOT(SUIT);
     END;
SCOOT: RETURN (RETN);
     END PLAYHI;
```

CALL MAKDLST(HOST, DLST); <u>or</u> LST = MAKDLST(HOST, DLST);

MAKDLST takes two already-created lists and makes the second the description list of the first. If HOST already had a description list, it is erased.

If used in an assignment statement, MAKDLST returns the name of the host list.

CALL NODLST(HOST); or LST = NODLST(HOST);

If the list named by HOST has a description list, it is erased and removed from HOST. LST, if present, is made the name of the host list.

Both MAKDLST and NODLST may be used on description lists of any form.

CALL NEWVAL(HOST, ATTR, ATID, NUVAL, VALID [, OLDVAL]);

On the description list of HOST, the old value of ATTR is removed and replaced by NUVAL. ATID is the SLIP ID of the attribute and VALID is the ID of the new value; both must be present. If OLDVAL is included in the argument list, it is given the old value for the attribute. OLDVAL must be of the same data type as the old value.

If there was no old value, and in fact was no ATTR on the attribute list, ATTR and NUVAL are added as the bottom pair of cells on the list.

If there was no attribute list for HOST, one is created with ATTR and NUVAL on it.

Programming note. If NEWVAL is to return old values of only one data type, the user can declare the function to return that type and then use assignment statements like this:

```
OLDVAL = NEWVAL(HOST, ATTR, ATID, NUVAL, VALID);
```

The same procedure may be applied to ITSVAL and NOVAL, described next, if appropriate.

CALL ITSVAL(HOST, ATTR, ATID, VALU);

ITSVAL searches the attribute list of HOST for the attribute ATTR, whose SLIP ID is given by ATID. If it is found, VALU is given the value for ATTR. VALU must be of the same data type as the attribute's value.

CALL NOVAL(HOST, ATTR, ATID [, VALU]);

NOVAL removes from the description list of HOST the cells with the attribute ATTR (which has ID = ATID) and with the attribute's value. If VALU is included in the argument list, it is given the value of the erased attribute.

DLST = NAMDLST(HOST);

If HOST has a description list, then DLST is given the name of that description list.

Boolean Tests

The functions described below are ones giving True-False answers about certain properties of specific lists. SLIP routines use these functions (particularly LSTNAME) to test the arguments they are given so that they do not blithely manipulate random storage locations. They may be useful in user programs as switches to leave or enter a section of the program, or as tests to guard against improper manipulations. Many SLIP functions return nothing if given invalid arguments and, at least in some parts of the user program, it is desirable to check the output of such functions before continuing.

FLAG = LSTNAME(LST);

If LST is the name of a valid list, FLAG is set to '1'B; otherwise, '0'B. LSTNAME checks that the addressed cell has ID = 12 and that the top and bottom cells of the list in fact point back to the header.

FLAG = CELLNAM(PNTR);

If PNTR points to a valid cell which is part of a list, FLAG ='1'B; otherwise, FLAG = '0'B. CELLNAM checks that the cells to the right and left of the one named by PNTR point back to it and that the ID is between 1 and 11.

FLAG = RDRNAME(RDR);

If RDR points to a validly-formed reader cell, FLAG = '1'B; otherwise, FLAG = '0'B. The function tests the ID of the cell and whether its LOFRDR points to a list and its CELLPNT points to a valid cell.

FLAG = LSTEMTY(LST);

If LST names an empty list consisting of the header and no data cells, FLAG = '1'B.

FLAG = LSTSEQL(LST1, LST2);

FLAG = '1'B if LST1 equals LST2, '0'B otherwise. LST1 "equals" LST2 if they both point to the same list or if (a) both list structures have equal data cells (ID = 1-10) at all points and (b) both cite the same or equal sublists at the same points. The list topographies must also match: if LST1 cites the same sublist twice, LST2 must also cite its version of the sublist twice. Description lists are not checked.

Random Number Functions

INT = NUMBEL(LST);

NUMBEL returns the integer number of data cells on the main list named LST. It does not count cells on sublists, nor does it count LST's header.

PNTR = RANDEL(LST);

RANDEL returns the address of a randomly-selected data cell from the main list LST. It will not select a cell from a sublist, although it may select a datum cell in LST which names a sublist.

The following functions generate random numbers according to several possible formats. Their use is not necessarily related to SLIP, and they may in fact be used in any PL-I program. The functions may also be called from FORTRAN programs or Assembly Language programs.

All the generating functions use a technique originally proposed by Tausworthe (1965) and described by Whittlesey (1968). From a starting 64-bit random number, the next number in the pseudorandom sequence is generated by ORing operations performed by the following machine code (RANDUB is the 64-bit number):

```
LM 2,3,RANDUB
LR 4,2
LR 5,3
SLDL 4,1
XR 3,4
XR 2,3
LPR 2,2
STM 2,3,RANDUB
```

This procedure has the advantages of high speed (for a subroutine call), minimum storage requirements, uniformity of distribution, and freedom from systematic sequential dependencies.

CALL SETRND(DX);

DX is a two-word long configuration by which the user provides the starting value for the generating sequence. If DX is greater than 0, it is used as the initial value; if it is 0 or negative, the 64-bit generator number is created from the current value of the computer's clock. By this latter option, a program can be written to automatically generate a new starting point every time it is run.

The generator number is initially set to a usable value, so it is not necessary to use SETRND prior to using any of the following functions. (Suitable starting numbers ought to have almost equal numbers of 0 and 1 bits.)

```
I = IRAND(J);
X = RAND(Y);
DX = DRAND(DY);
DX = DRANDM;
```

IRAND returns a binary integer between 0 and J-1, inclusively. J must be a positive binary integer.

RAND returns a single word floating point number between 0 and Y--which must be a positive floating point number.

DRAND and DRANDM return double word floating point numbers. DRANDM returns a value between 0 and 1, while DRAND returns a value between 0 and DY, a double word floating point number. (Note. To conform to FORTRAN's calling conventions, DRANDM must be invoked by a statement such as CALL DRANDM(DX).)

```
DZ = SAVRND;
```

SAVRND returns the double word number used in generating the pseudo-random numbers in the functions above. By using SAVRND at the end of one day's random number generation, it is possible to initiate the generator with that value on the next run, via SETRND, thus continuing with one long sequence of numbers. (In FORTRAN, SAVRND must be invoked via CALL SAVRND(DZ).)

Other Functions

The rest of the functions described below are ones which deal with cells and data storage in more detail. With the exception of the list marking functions, the user should have few occasions to use them.

Cell information retrieval functions.

```
INT = ID(PNTR);
INT = MARK(PNTR);
POINT = LNKR(PNTR);
POINT = LNKL(PNTR);
```

If PNTR contains the address of any SLIP cell, these four functions return the value from their particular portion of the cell's links word.

```
POINT = CONTS(PNTR);
```

If PNTR contains the address of a SLIP cell, CONTS returns the address of the cell's datum.

```
POINT = LINKS(PNTR);
```

If PNTR has the address of the datum of a cell, LINKS returns the address of the cell. (LINKS and CONTS are opposites; CONTS returns PNTR + 8 and LINKS returns PNTR - 8.)

```
INT = REFCNT(LST);
```

REFCNT returns the reference counter value from the header of the list named LST.

Functions for altering cell links field information.

```
CALL SETID(PNTR, INT);
CALL SETMARK(PNTR, INT);
```

SETID and SETMARK insert the value (modulo 256) of the binary integer INT into the ID and MARK portions of the cell addressed by PNTR. Caution: If one uses ID values other than 1-13 for list cells, most SLIP functions will misinterpret or simply fail on such cells.

```
CALL SETLNKR(PNTR, POINT);
CALL SETLNKL(PNTR, POINT);
```

SETLNKR (SETLNKL) inserts the address contained in POINT into the LNKR (LNKL) field of the cell addressed by PNTR. These functions should also be used cautiously, since they directly alter the structure of the list containing the cell.

```
CALL SETIND(PNTR, IDVAL, LINKL, LINKR);
CALL SETDIR(CELL, IDVAL, LINKL, LINKR);
```

These functions may be used to set the ID and/or the LNKL and/or the LNKR fields of a SLIP cell. If one or more of the fields are not to be changed, the corresponding argument should be the binary integer -1 or the pointer variable NULL. SETIND changes the cell addressed by PNTR; SETDIR changes CELL itself.

```
CALL MRKLSTS(LST, INT);
```

If LST names a list structure, MRKLSTS sets the MARK portion of the headers of LST and all its sublists to the value of INT (mod 256). This is often useful to reset the MARKs after a scanning or searching operation has used header MARKs to remember which lists had been scanned. MRKLSTS ignores description lists.

Miscellaneous functions.

```
INT = STRLNTH(STRING);
```

If STRING is the name of any PL-I string variable, STRLNTH returns its current length as read from the Dope Vector. This function is usefully different from the PL-I built-in function LENGTH only in the case of strings defined as fixed length which have been treated by SLIP as variable length.

```
PNTR = NEWCELL(DATM);
```

If DATM is any allowable SLIP datum, NEWCELL creates a new cell containing DATM and returns the address of the cell. The ID of the cell is set to the proper value between 1 and 11; the MARK, LNKL, and LNKR fields are 0.

PNTR = MAKCELL(DATM, DATID);

This function actually does the work of obtaining new cells from AVSL. DATM is stored in the cell and DATID is stored as the cell's ID. PNTR is given the address of the new cell.

CALL RCELL(PNTR);

RCELL actually returns erased cells to AVSL. PNTR addresses the cell to be returned, which can have an ID only between 1 and 11. RCELL simply returns the cell; in particular, it does not link other list cells around the removed cell.

CALL INSERTR(PNTR, POINT);
CALL INSERTL(PNTR, POINT);

The cell addressed by POINT, which must not be part of a list, is inserted to the right (left) of the list cell addressed by PNTR.

FLAG = COMPARE(DAT1, DAT2, DATID);

COMPARE tests the equality of two SLIP data, both of ID type DATID, and returns '1'B if they are equal or '0'B if they are not. DAT1 and DAT2 must be the data themselves, not pointers to the data. Two strings of unequal length are considered equal if they are equal for the common length and if the longer is blank (for character strings) or 0 (for bit strings) for the extra length; this is in accord with PL-I conventions.

CALL DSADUMP;
CALL LDUMP(VAR, LNGTH, CODE);

Both these functions are intended primarily as debugging functions. They use the FORTRAN library function PDUMP; thus, to use them, the user must include the FORTRAN library (FORTLIB) in the list of libraries available to the linkage editor and must define the standard FORTRAN output file in the execution step (e.g., //GO.FTØ6FØØ1 DD SYSOUT=A).

DSADUMP dumps the dynamic storage area of the program or subroutine in which the calling statement is placed. This is often useful because it gives the status of the 16 index registers at the time DSADUMP was called and the values of all the dynamic-allocation variables in the program or subroutine at the time of the call. (These are all unlabeled; the user must have some knowledge of PL-I program organization and a program object code listing to make sense of the dump.) The dump is in hexadecimal.

LDUMP dumps the section of core storage beginning with the variable VAR and continuing for approximately LNGTH bytes. LNGTH may be negative, in which case VAR is the upper end of the dumped core. CODE is a binary integer indicating the form of the dump, according to the following table (the parenthesized values are useless in a PL-I dump):

CODE	Interpretation of data for output
0	Hexadecimal.
(1)	Boolean, in byte chunks (T if at least one 1 in the byte, F if all 0).
(2)	Boolean, in word chunks.
(3)	Halfword binary integer.
4	Fullword binary integer.
5	Fullword floating point.
6	Double word floating point.
(7)	Fullword floating point, complex.
(8)	Double word floating point, complex.
9	Characters, 1 per byte.

The three arguments may be repeated any number of times, so that a single call to LDUMP can produce dumps of different segments of core, different lengths, or different output formats.

Neither function terminates execution of the main program; thus they may be used to print key areas of storage several times in a single run.

Index of SLIP Functions

Name and Argument List	Purpose	Page
ADVSTR(RDR [, DATID][, FLAG]) ADVSTL ADVLNR ADVLNL	advance reader structurally (linearly) to the right (left)	29
BOT(LST, DATM)	what is the list's bottom datum?	20
CELLNAM(PNTR)	is the argument a cell's name?	38
CELLPNT(RDR)	to what cell does a reader point?	32
COMPARE(DAT1, DAT2, DATID)	compare two data	43
CONTS(PNTR)	what is the address of a cell's datum?	41
COPYLST(ORGL, COPY)	create a copy of a list structure	17
COPYRDR(RDR1)	create a copy of a reader	28
DATUM(PNTR, DATM)	what is a cell's datum?	20
DRAND(DY)	random number (long floating point)	40
DRANDM	unit random number (long floating point)	40
DSADUMP	dump the program's dynamic storage area	43
EMTYLST(LST)	empty a list	18
ERASLST(LST)	erase a list	18
ERASRDR(RDR)	erase a reader	29
ID(PNTR)	what is a cell's ID value?	41
INITAS([INT])	initialize the available space list	15
INITRDR(RDR)	initialize a reader within its level	32
INSERTR(PNTR, POINT) INSERTL	insert a cell to the right (left) in a list	43

Name and Argument List	Purpose	Page
IRAND(J)	random number (long binary integer)	40
ITSVAL(HOST, ATTR, ATID, VALU)	what is the attribute's value?	36
LDUMP(VAR, LNGTH, CODE)	dump core segment by length	45
LINKS(PNTR)	what is the address of the cell datum's links portion?	41
LIST(LST)	create (an empty) list	17
LNKL(PNTR) LNKR	what is a cell's LNKL (LNKR) value?	41
LOFRDR(RDR)	what is the list of a reader?	32
LSTEMTY(LST)	is a list empty?	38
LSTNAME(LST)	is the argument a list name?	37
LSTSEQL(LST1, LST2)	are two list structures equal?	38
LVLCNT(RDR)	what is the reader's level?	32
LVLRVT(RDR)	revert to the top level	31
LVLRVT1(RDR)	revert one level back	31
MAKCELL(DATM, DATID)	create a new cell	43
MAKDLST(HOST, DLST)	make a description list	35
MARK(PNTR)	what is a cell's MARK value?	41
MOVER(LST1, LST2) MOVEL	move a segment of a list to the right (left) in another list	20
MRKLSTS(LST, INT)	set the MARKs of the headers in a list structure	42
NAMDLST(HOST)	what is the name of the description list?	37
NEWCELL(DATM)	create a new cell	42
NEWTOP(LST, DATM) NEWBOT	put a new datum at the top (bottom) of a list	19

Name and Argument List	Purpose	Page
NEWVAL(HOST, ATTR, ATID, NUVAL, VALID [, OLDVAL])	give the attribute a new value	36
NODLST(HOST)	remove and erase the description list	36
NOVAL(HOST, ATTR, ATID [, VALU])	remove the attribute and its value	37
NUMBEL(LST)	how many elements are on a list?	38
POPTOP(LST [, DATM]) POPBOT	pop the top (bottom) cell from a list	20
PRNTLST(LST [, 'BREAK=x'][,'B'][, FILENAM])	print a list structure	24
PUTDATM(PNTR)	print a cell and its datum	25
PUTLIST(LST)	print a list structure in debugging format	25
RAND(Y)	random number (floating point)	40
RANDEL(LST)	select a random list element	38
RCELL(PNTR)	return a cell to AVSL	43
RDRNAME(RDR)	is the argument a reader's name?	38
RDROF(LST)	make a reader of a list	28
READLST(LST [, 'BREAK=x'][, {'B' <u>or</u> 'T'}][, FILENAM])	read a list structure from stream input	21
REED(RDR, DATM)	what is the datum to which the reader points?	32
REFCNT(LST)	what is a list's reference counter?	41
REMOVE(PNTR [, DATM])	remove a cell from a list	20
REPLACE(PNTR, DATM [, DATID][, OLDDAT])	replace a datum cell	19
SAVRND	save random number generator value	40
SEARCH({LST <u>or</u> RDR}, DATM, DATID [, PNTR])	search a list structure for a datum	33

Name and Argument List	Purpose	Page

SETDIR(CELL, IDVAL, LINKL, LINKR)
SETIND(PNTR, IDVAL, LINKL, LINKR)
	set the ID, LNKL, and LNKR of the argument (or the cell addressed by the argument)	42

SETID(PNTR, INT)
SETLNKL(PNTR, POINT)
SETMARK(PNTR, INT)
SETLNKR(PNTR, POINT)
	set the ID (LNKL, MARK, or LNKR) portion of the addressed cell	41

SETRND(DX)	set random number generator value	39
STRLNTH(STRING)	what is a string's length?	42
SUBTOP(LST, DATM [. DATID][, OLDDAT]) SUBBOT	substitute the top (bottom) data cell on a list	19
TOP(LST, DATM)	what is the top datum of a list?	20

References

Johnson, E. S. SLIP--A symmetric list processor. Research Memorandum No. 31. Chapel Hill, N.C.: L. L. Thurstone Psychometric Laboratory, University of North Carolina, 1968.

Johnson, E. S., Rosin, R. F., & Leaf, W. A. SLIP--A symmetric list processor. Memorandum No. 62. New Haven, Conn.: Yale University Computer Center, 1967.

Tausworthe, R. C. Random numbers generated by linear recurrence modulo two. Mathematics of Computation, 1965, 19, 201-209.

Weizenbaum, J. Symmetric list processor. Communications of the Association for Computing Machinery, 1963, 6, 524-536.

Whittlesey, J. R. B. A comparison of the correlational behavior of the random number generators for the IBM 360. Communications of the Association for Computing Machinery, 1968, 11, 641-644.

Appendix A

Running a SLIP Program

Because SLIP is a set of library subroutines which are not stored on the standard system library, a few changes in job control (JCL) cards must be made.

1. In the link editor stage, the library must be revised from the usual SYS1.PL1LIB to the following:

```
//LKED.SYSLIB DD DSNAME=SYS1.PL1LIB,DISP=SHR
//             DD DSNAME=OCS.SLIPLIB,DISP=SHR
```

If either LDUMP or DSADUMP is being used, the FORTRAN library must be concatenated by a third card:

```
//             DD DSNAME=SYS1.FORTLIB,DISP=SHR
```

2. In the execution stage, one extra output file should be declared, not only for SLIP but for any PL-I program in which an abnormal job termination might occur:

```
//GO.PL1DUMP DD SYSOUT=A
```

And, if LDUMP or DSADUMP is being used,

```
//GO.FTØ6FØØ1 DD SYSOUT=A
```

One of the most time-consuming chores in writing SLIP programs is defining, via a DECLARE statement, all the SLIP functions one plans to use. In order to avoid as much of this effort as possible, an external library has been established with three different sets of function declarations: LISTS, READERS, and DLISTS. The first, and most frequently used, defines the basic list functions. This segment also contains a "CALL INITAS;" statement, so that one need not write a separate call while using the first declaration.

The second declaration defines all reader-connected functions, and the third defines the description list functions. The contents of these segments, along with standard definitions for the SLIP functions not covered by the library, are listed below.

To incorporate any or all of these definition segments into the user's program, one calls upon one of PL-I's preprocessor statements--the %INCLUDE statement.

At ETS, these segments exist on ETSLIB, in the partitioned data set OCS.SLIPTXT. One must follow these steps in order to have access to them.

1. Include the following data definition card in the PL-I step:

```
//PL1L.SYSLIB DD DSNAME=OCS.SLIPTXT,DISP=SHR
```

2. In the option list for the PL-I job step, include the parameter which will cause the preprocessor to be invoked (MACRO) along with whatever other parameters are desired (e.g., LOAD,ATR,XREF). One may wish SOURCE2, which will print the program as it appears after the preprocessor pass, and also MACDCK, which will cause the same thing to be punched on cards.

3. In the program, preferably before any executable statements, request the inclusion of the segments:

```
    %INCLUDE LISTS, READERS, DLISTS;
```

using all three segments or any two or one.

Note. If one wishes to redefine one of the functions included in one of the library DECLARE segments (as, for example, to use TOP or BOT in assignment statements--see pp. 18-19), it is not necessary to avoid using the segment. Simply define the function as is appropriate to the program in a DECLARE statement which precedes the %INCLUDE statement: PL-I will accept the first definition and ignore the one which comes later, although the compiler will print a warning against multiple function definitions.

There is some cost to the user when he uses these preprocessor segments. First, the preprocessor stage takes some computer time. Second, and more important, all of the functions defined are loaded from SLIPLIB into the computer--whether the program will use them or not. In cases with programs to be run once or twice, these costs may be tolerable. For longer runs, this procedure seems to be the best compromise:

1. In the first run, use the MACDCK option along with MACRO; this will cause to be punched a deck of the source program after the preprocessor run--i.e., including the requested SLIPTXT segments.

2. For future runs, omit the MACRO and MACDCK options and run without the preprocessor stage (unless the program requires it for other reasons).

3. Remove from the SLIPTXT cards the definition cards for the functions either not used in the program or redefined in earlier declarations.

The LISTS Text Segment

```
DECLARE
  [illegible] ENTRY (PTR,),
  CELLNAM ENTRY (PTR) RETURNS (BIT(1)),
  COMPARE ENTRY (,, FIXED BIN (31)) RETURNS (BIT(1)),
  CONTS ENTRY (PTR) RETURNS (PTR),
  COPYLST ENTRY (PTR, PTR) RETURNS (PTR),
  DATUM ENTRY (PTR,),
  DATYLST ENTRY (PTR) RETURNS (PTR),
  [illegible]SLST ENTRY (PTR) RETURNS (FIXED BIN (31)),
  ID ENTRY (PTR) RETURNS (FIXED BIN (31)),
  INITIAS ENTRY (FIXED BIN (31)),
  LINKS ENTRY (PTR) RETURNS (PTR),
  LIST GENERIC
    (LIST1 ENTRY (PTR) RETURNS (PTR),
     LIST2 ENTRY (FIXED BIN (31)) RETURNS (PTR),
     LIST3 ENTRY (FIXED DEC (1)) RETURNS (PTR)),
  LNKL ENTRY (PTR) RETURNS (PTR),
  LFKI ENTRY (PTR) RETURNS (PTR),
  LSTEMTY ENTRY (PTR) RETURNS (BIT(1)),
  LSTNAME ENTRY (PTR) RETURNS (BIT(1)),
  LSTEQL ENTRY (PTR, PTR) RETURNS (BIT(1)),
  MARK ENTRY (PTR) RETURNS (FIXED BIN (31)),
  NEWBOT GENERIC
    (NEWBOT1 ENTRY (PTR, FIXED BIN (15)) RETURNS (PTR),
     NEWBOT2 ENTRY (PTR, FIXED BIN (31)) RETURNS (PTR),
     NEWBOT3 ENTRY (PTR, FLOAT BIN (21)) RETURNS (PTR),
     NEWBOT4 ENTRY (PTR, FLOAT BIN (53)) RETURNS (PTR),
     NEWBOT5 ENTRY (PTR, FIXED DEC ( 5)) RETURNS (PTR),
     NEWBOT6 ENTRY (PTR, FIXED DEC (15)) RETURNS (PTR),
     NEWBOT7 ENTRY (PTR, FLOAT DEC ( 6)) RETURNS (PTR),
     NEWBOT8 ENTRY (PTR, FLOAT DEC (16)) RETURNS (PTR),
     NEWBOT9 ENTRY (PTR, CHAR (256)) RETURNS (PTR),
     NEWBT9A ENTRY (PTR, CHAR (256) VAR) RETURNS (PTR),
     NEWBT10 ENTRY (PTR, BIT (2048)) RETURNS (PTR),
     NUBT10A ENTRY (PTR, BIT (2048) VAR) RETURNS (PTR),
     NEWBT11 ENTRY (PTR, PTR) RETURNS (PTR)),
  NEWTOP GENERIC
    (NEWTOP1 ENTRY (PTR, FIXED BIN (15)) RETURNS (PTR),
     NEWTOP2 ENTRY (PTR, FIXED BIN (31)) RETURNS (PTR),
     NEWTOP3 ENTRY (PTR, FLOAT BIN (21)) RETURNS (PTR),
     NEWTOP4 ENTRY (PTR, FLOAT BIN (53)) RETURNS (PTR),
     NEWTOP5 ENTRY (PTR, FIXED DEC (5) ) RETURNS (PTR),
     NEWTOP6 ENTRY (PTR, FIXED DEC (15)) RETURNS (PTR),
     NEWTOP7 ENTRY (PTR, FLOAT DEC ( 6)) RETURNS (PTR),
     NEWTOP8 ENTRY (PTR, FLOAT DEC (16)) RETURNS (PTR),
     NEWTOP9 ENTRY (PTR, CHAR (256)) RETURNS (PTR),
     NEWTP9A ENTRY (PTR, CHAR (256) VAR) RETURNS (PTR),
     NEWTP10 ENTRY (PTR, BIT (2048)) RETURNS (PTR),
     NUTP10A ENTRY (PTR, BIT (2048) VAR) RETURNS (PTR),
     NEWTP11 ENTRY (PTR, PTR) RETURNS (PTR)),
  POPBOT ENTRY (PTR,),
  POPTOP ENTRY (PTR,),
  REFCNT ENTRY (PTR) RETURNS (FIXED BIN (31)),
  REMOVE ENTRY (PTR,),
        REPLACE ENTRY (PTR,,FIXED BIN (31),),
        SETMARK ENTRY (PTR, FIXED BIN (31)),
        SUBTOT ENTRY (PTR,, FIXED BIN (31),),
        SUBTOP ENTRY (PTR,, FIXED BIN (31),),
        TOP ENTRY (PTR,);
      CALL INITAS;
```

The READERS Text Segment

```
    DECLARE
      ADVSTR GENERIC
        (ADVSTR1 ENTRY (PTR, FIXED BIN (31),BIT(1)) RETURNS (PTR),
         ADVSTR2 ENTRY (PTR, BIT (1)) RETURNS (PTR),
         ADVSTR3 ENTRY (PTR, FIXED BIN (31)) RETURNS (PTR),
         ADVSTR4 ENTRY (PTR) RETURNS (PTR)),
      ADVSTL GENERIC
        (ADVSTL1 ENTRY (PTR, FIXED BIN (31), BIT(1)) RETURNS (PTR),
        ADVSTL2 ENTRY (PTR, BIT (1)) RETURNS (PTR),
        ADVSTL3 ENTRY (PTR, FIXED BIN (31)) RETURNS (PTR),
        ADVSTL4 ENTRY (PTR) RETURNS (PTR)),
      ADVLNR GENERIC
        (ADVLNR1 ENTRY (PTR, FIXED BIN(31), BIT(1)) RETURNS (PTR),
         ADVLNR2 ENTRY (PTR, BIT(1)) RETURNS (PTR),
         ADVLNR3 ENTRY (PTR, FIXED BIN (31)) RETURNS (PTR),
         ADVLNR4 ENTRY (PTR) RETURNS (PTR)),
      ADVLNL GENERIC
        (ADVLNL1 ENTRY (PTR, FIXED BIN (31), BIT (1)) RETURNS (PTR),
         ADVLNL2 ENTRY (PTR, BIT (1)) RETURNS (PTR),
         ADVLNL3 ENTRY (PTR, FIXED BIN (31)) RETURNS (PTR),
         ADVLNL4 ENTRY (PTR) RETURNS (PTR)),
      CELLPNT ENTRY (PTR) RETURNS (PTR),
      COPYRDR ENTRY (PTR) RETURNS (PTR),
      ERASRDR ENTRY (PTR) RETURNS (FIXED BIN (31)),
      FIRMRDR ENTRY (PTR) RETURNS (PTR),
      LOFRDR ENTRY (PTR) RETURNS (PTR),
      LVLCNT ENTRY (PTR) RETURNS (FIXED BIN (31)),
      LVLEVT ENTRY (PTR) RETURNS (PTR),
      LVLEVT1 ENTRY (PTR) RETURNS (PTR),
      RDRNAME ENTRY (PTR) RETURNS (BIT(1)),
      SIZOF ENTRY (PTR) RETURNS (PTR),
      RFID ENTRY (PTR,);
```

The DLISTS Text Segment

```
DECLARE
    ITSVAL ENTRY (PTR,, FIXED BIN (31),),
    MRKDLST ENTRY (PTR, BIT) RETURNS (PTR),
    NAMDLST ENTRY (PTR) RETURNS (PTR),
    NEWVAL ENTRY (PTR,, FIXED BIN (31),, FIXED BIN (31),),
    NODLST ENTRY (PTR) RETURNS (PTR),
    NOVAL ENTRY (PTR,, FIXED BIN (31),);
```

The following declarations are not included on any SLIPTXT segment because they represent functions which are used rarely--too infrequently to justify keeping them present, occupying computer space, for every program run. The declarations below are the ones which ought to be used if one does wish to call on the functions:

```
DECLARE     DRAND ENTRY (FLOAT BIN (53)) RETURNS (FLOAT BIN (53));
DECLARE     DRANDM RETURNS (FLOAT BIN (53));
DECLARE     DSADUMP ENTRY;
DECLARE     INSERTR ENTRY (PTR, PTR);
DECLARE     INSERTL ENTRY (PTR, PTR);
DECLARE     IRAND ENTRY (FIXED BIN (31)) RETURNS (FIXED BIN (31));
DECLARE     LDUMP ENTRY (, FIXED BIN (31), FIXED BIN (31));
DECLARE     MAKCELL (, FIXED BIN (31)) RETURNS (PTR);
DECLARE     MOVER ENTRY (PTR, PTR);
DECLARE     MOVEL ENTRY (PTR, PTR);
DECLARE     MRKLSTS (PTR, FIXED BIN (31));
DECLARE     NEWCELL GENERIC
     (NEWCEL1 ENTRY (FIXED BIN (15)) RETURNS (PTR),
      NEWCEL2 ENTRY (FIXED BIN (31)) RETURNS (PTR),
      NEWCEL3 ENTRY (FLOAT BIN (21)) RETURNS (PTR),
      NEWCEL4 ENTRY (FLOAT BIN (53)) RETURNS (PTR),
      NEWCEL5 ENTRY (FIXED DEC ( 5)) RETURNS (PTR),
      NEWCEL6 ENTRY (FIXED DEC (15)) RETURNS (PTR),
      NEWCEL7 ENTRY (FLOAT DEC ( 6)) RETURNS (PTR),
      NEWCEL8 ENTRY (FLOAT DEC (16)) RETURNS (PTR),
      NEWCEL9 ENTRY (CHAR (256)) RETURNS (PTR),
      NEWCL9A ENTRY (CHAR (256) VAR) RETURNS (PTR),
      NEWCL10 ENTRY (BIT (2048)) RETURNS (PTR),
      NUCL10A ENTRY (BIT (2048) VAR) RETURNS (PTR),
      NEWCL11 ENTRY (PTR) RETURNS (PTR));
```

```
DECLARE     NUMBEL ENTRY (PTR) RETURNS (FIXED BIN (31));
DECLARE     PRNTLST  GENERIC
    (PRNTLS1 ENTRY (PTR),
     PRNTLS2 ENTRY (PTR, CHAR(1)),
     PRNTLS3 ENTRY (PTR, CHAR(1), CHAR(1)),
     PRNTLS4 ENTRY (PTR, FILE),
     PRNTLS5 ENTRY (PTR, CHAR (1), FILE),
     PRNTLS6 ENTRY (PTR, CHAR (1), CHAR (1), FILE));
DECLARE     PUTDATM ENTRY (PTR);
DECLARE     PUTLIST ENTRY (PTR);
DECLARE     RAND ENTRY (FLOAT BIN (21)) RETURNS (FLOAT BIN (21));
DECLARE     RANDEL ENTRY (PTR) RETURNS (PTR);
DECLARE     RCELL ENTRY (PTR);
DECLARE     READLST GENERIC
    (READLS1 ENTRY (PTR),
     READLS2 ENTRY (PTR, CHAR (1)),
     READLS3 ENTRY (PTR, CHAR (1), CHAR (1)),
     READLS4 ENTRY (PTR, FILE),
     READLS5 ENTRY (PTR, CHAR(1), FILE),
     READLS6 ENTRY (PTR, CHAR (1), CHAR (1), FILE));
DECLARE     SAVRND RETURNS (FLOAT BIN (53));
DECLARE     SEARCH ENTRY (PTR,, FIXED BIN (31), PTR) RETURNS (BIT (1));
DECLARE     SETDIR ENTRY (, FIXED BIN (31), PTR, PTR);
DECLARE     SETIND ENTRY (PTR, FIXED BIN (31), PTR, PTR);
DECLARE     SETID ENTRY (PTR, FIXED BIN (31));
DECLARE     SETLNKL ENTRY (PTR, PTR);
DECLARE     SETLNKR ENTRY (PTR, PTR);
DECLARE     SETRND ENTRY ;
DECLARE     STRLNTH RETURNS (FIXED BIN (31));
```

Appendix B

SLIP Error Facilities

Although SLIP is a subroutine language operating within PL-I, the PL-I diagnostics are generally unhelpful for locating an error in list processing. There are two reasons for this: first, SLIP performs operations which are outside the realm of normal PL-I capabilities; and second, SLIP subroutines take some liberties and violate some assumptions of PL-I programming in being able to accomplish their ends.

The major violation of PL-I standards is in allowing argument lists of variable length for the subroutine calls. While taboo in PL-I, variable length argument lists are common to most other languages and have been artificially preserved in SLIP. This allows the user much more flexibility than fixed-length argument lists but makes most of PL-I's error messages on argument lists inapplicable.

An advantage to PL-I is that its error messages have several levels of severity. The philosophy seems to be to inform the user if he has made a mistake but then allow his program to run as far as possible anyway. This works ideally with SLIP, in which these "mistakes" are made quite frequently and purposefully.

SLIP contains some brief error detec[illegible]n facilities of its own. SLIP attempts to keep the user's program running as long as possible and to allow the user as much freedom as possible. There is only one condition for which SLIP will terminate a program: lack of space, which makes it impossible to create new cells. In all other cases, SLIP subroutines check to make sure that its list manipulations are actually being done to lists; if not, a warning is printed and execution continues without the asked-for manipulation. These checks are intended to provide the user with some indication that his program may be misperforming and to prevent it from doing something so wrong that the PL-I monitor will detect an error and halt execution.

If the user's program is going very wrong, then repeatedly calls to SLIP functions will have invalid arguments and a great amount of computer time might be wasted. To counteract this, SLIP counts the calls to its error routines. If the calls exceed a certain number (by default 50; it may be set to any desired value, though, by including the value as an argument to INITAS; see p. 15 above), execution is stopped, the following error message is printed, and the standard PL-I error condition is raised:

```
**** SLIP ERROR TALLY HAS REACHED nnnn.  PROGRAM EXECUTION HAS BEEN
          TERMINATED AT THIS POINT.
```

INITAS has two error messages to indicate that no more space could be found for list storage and the program is therefore halted; these were listed on p. 9.

Several functions require one or more list names as arguments. Because they must perform manipulations based on the list names, they test this and in general do nothing if they have been given nonnames. The following error message is printed in such cases:

```
**** fname WAS CALLED WITHOUT A LIST NAME (OR NAMES) IN ITS ARGUMENT LIST.
**** THE FUNCTION EXITED WITHOUT DOING ANYTHING.
```

The name of the function is inserted in the first line. The functions which do this include COPYLST, EMTYLST, ERASLST, ITSVAL, LSTSEQL, MAKDLST, MOVER, MOVEL, MRKLSTS, NAMDLST, NEWVAL, NODLST, NOVAL, NUMBEL, PRNTLST, PUTLIST, RANDEL, and RDROF.

Other functions have slightly different argument requirements, or detect other types of errors. Their main comment lines, which appear with a reminder that the function did nothing, are listed below:

ADVSTR ADVSTL ADVLNR ADVLNL COPYRDR ERASRDR	REQUIRES THE NAME OF THE READER OF A VALID LIST IN ITS ARGUMENT LIST.
BOT DATUM TOP	WAS ASKED TO RETURN THE DATUM OF SOMETHING WHICH WAS NOT A SLIP DATA CELL.
ITSVAL NOVAL	WAS GIVEN THE NAME OF A LIST WITH NO DESCRIPTION LIST.
NEWBOT NEWTOP	REQUIRES A LIST NAME OR LIST CELL ADDRESS AS ITS FIRST ARGUMENT.

IF THE DATUM TO BE ADDED TO THE LIST IS A POINTER VARIABLE, {NEWTOP / NEWBOT / NEWCELL} REQUIRES IT TO BE A SUBLIST NAME.

POPBOT POPTOP REMOVE	WAS ASKED TO ERASE SOMETHING WHICH WAS NOT A SLIP DATA CELL.

PUTDATM REQUIRES THE ADDRESS OF A SLIP CELL AS ITS ARGUMENT.

READLST COULD NOT FIND THE START OF A LIST WITHIN 240 CHARACTERS.
READLST FOUND SOME CHARACTERS WHICH COULD NOT BEGIN A LIST.

SEARCH REQUIRES THAT ITS FIRST ARGUMENT NAME A LIST OR A READER.

SUBBOT SUBTOP REPLACE	WAS ASKED TO SUBSTITUTE A NEW DATUM IN PLACE OF SOMETHING WHICH WAS NOT A SLIP DATA CELL.

Appendix C

Subtle Features of PL-I(F)

For the average user of PL-I and SLIP, it is adequate to know the information presented to this point. For the user who may wish to attempt things slightly outside the normal applications of SLIP, and for the user who reads and is concerned with the PL-I compilation error messages which commonly accompany SLIP, the following section is intended.

Function and Subroutine Conventions

PL-I almost follows standard OS conventions in the way it sets up and manipulates function calls and argument lists. By convention, the argument list of a function is represented by a string of word-long addresses which, in order, point to the actual first, second, etc. arguments to the function. This address string is in turn addressed by the value in General Register 1 (out of 16, numbered 0 through 15). Because OS allows variable-length lists of arguments, there is normally a flag set in the last address in the list to indicate that it is the address of the last argument; this flag is simply a minus sign--i.e., the first bit in the last address is 1.

PL-I sets up its argument list in the same way. Although PL-I does not "allow" variable-length argument lists, and thus has no need for flagging the last argument address, PL-I does flag the address in order to be consistent with OS conventions.

There are two differences, however. If a FORTRAN function, for example, is to return a value, the function leaves the value in a conventional spot and the calling program retrieves it. Since PL-I may return anything which is a valid data type, PL-I can't reserve a conventional spot sufficiently flexible to accept any return value. Therefore, PL-I artificially takes the recipient of the function's value and makes it the final argument to the call. This is why, for example, the normal call for TOP (e.g., CALL TOP(LST, DATM);) can be redefined so that DATM = TOP(LST); is acceptable. Both statements generate a call to TOP which appears to have two arguments.

The second difference appears to be a compiler error, but one requiring enough coincidences that it is unlikely to bother the user. Because of SLIP's non-PL-I treatment of arguments, however, program errors due to this are extremely hard to detect and the problem deserves description.

Under certain circumstances, PL-I will flag an address which is not the final argument address; this makes it appear that the argument list is shorter than it actually is. This may happen if the function call comes from within a subroutine and the final parameter to the subroutine is a nonfinal argument in the function call. For example:

```
DUMMY:   PROG (VAR, VARY);
         . . .
         CALL VERYDUM(VARY, X, Y, Z);
         . . .
         END DUMMY;
```

In coding the call to VERYDUM, PL-I uses the address of VARY as taken from the argument list to DUMMY. Since VARY was the last argument to DUMMY, its address was flagged; PL-I does not remove the flag before putting the address into VERYDUM's argument list.

If VERYDUM is a function whose operation depends on the number of arguments it thinks it receives, it will operate not as intended since it will find it has only one argument.

Many SLIP functions count their arguments and respond differently for different numbers of arguments. They have been programmed around this compiler quirk for their own calls to other SLIP functions, but the user can run into this problem if he writes subroutines which call SLIP functions.

SLIP uses variable length argument lists in two ways: first, by allowing certain arguments to be dropped from the list if they are unwanted (e.g., CALL PRNTLST(LST, 'BREAK=.', OWNFILE); vs. CALL PRNTLST(LST);); and second, by letting functions return values or not (e.g., CALL ERASLST (LST); vs. ICOUNT = ERASLST(LST);). PL-I does not allow such things, and it objects during compilation with different degrees of severity:

1. CALL ERASLST(LST); and ICOUNT = ERASLST(LST); PL-I codes both correctly and issues no warning; PL-I seems completely blind to this kind of violation. PL-I will object, however, if the function is to return a data type not compatible with the recipient: RANDEL normally returns a pointer value; the incompatible INT = RANDEL(LST); would be rejected, although it would be fine if RANDEL was defined as returning a fixed binary number.

2. DCL COPYLST ENTRY(PTR, PTR) RETURNS (PTR); and COPY = COPYLST (ORGL); The call will be coded as written, which is acceptable to COPYLST, and the following severe error message will appear:

```
IEM0787I  INCORRECT NUMBER OF ARGUMENTS FOR FUNCTION zzzz IN STATEMENT
                                                     NUMBER xxx
```

3. COPY = COPYLST(ORGL, COP); and COPY = COPYLST(ORGL); Both statements will be coded as written but this warning will be printed:

```
IEM0791I  NUMBER OF ARGUMENTS FOR zzzz STATEMENT NUMBER xxx INCONSISTENT
                     WITH NUMBER USED ELSEWHERE
```

5. Incorrect matching of argument attributes with parameter attribute definitions. For normal functions, this combination will result in a data conversion if possible or, if not, statement deletion: DCL SETID ENTRY (PTR, FIXED BIN (31)); and CALL SETID(LST, X); If X is a float decimal, for example, conversion wil. take place. For generic functions, an attribute mismatch will terminate compilation.

6. Generic functions. From Appendix A, it is apparent that SLIP uses generic functions extensively. Generic functions allow the user to specify only the generic name in his program and have the compiler automatically select the proper entry. Such functions allow greater flexibility and power in the writing of the subroutines. In functions like NEWTOP (pp. 19 and 54), for example, the entry point tells NEWTOP the type of datum to be put in a SLIP cell.

Generic functions are also extremely unforgiving. One must specify completely all the arguments' attributes and the return attributes, if any (although even generic functions don't care if the executed statement requires a returned value or not). And if attributes in the declaration don't match the attribute in the calling statement, compilation is terminated rather than data conversion attempted.

For numeric data, precisions must match perfectly: if NEWTOP1 requires a second argument with FIXED BIN (15) attributes, an argument with FIXED BIN (14) attributes will cause termination, even though both types of variables have the same physical representation.

For string data, the length of the string is not considered in the matching. Although NEWTOP9 is declared with a second argument whose attributes are CHAR (256), a fixed length character string of any length will satisfy the requirements. No string of varying length will, however; thus the parallel entry NEWTP9A.

Based Variables and Dope Vectors

Based variables consist of a (potentially existing) variable and a pointer which addresses the location of the variable. Based variables exist only when the pointer has been given a valid address, either through an ALLOCATE statement or by an assignment statement or subroutine call which gives the pointer some value. For SLIP, based variables tend to be quite useful as a means of gaining access to list information. For example. if the declaration DCL DADDR PTR BASED (DPOINT); exists and CALL LIST (DPOINT); is executed, DADDR is the LNKL address of the header of the list (plus the ID, which does not interfere with the address potential of DADDR).

Based variables which are scalars without Dope Vectors are uncomplicated; if the base pointer variable contains an address, then the variable is seen as being at that location.

Based variables which have Dope Vectors, such as strings and arrays, are more complicated. This is primarily because the base pointer, which addresses the variable, and the Dope Vector pointer, which addresses the variable, usually do not hold the same value.

For arrays, the Dope Vector consists of the virtual address of the array element whose subscripts are all 0 and multipliers for determining the location of any element given its subscripts. This virtual address is usually outside the physical limits of the array. For a based array, the base pointer contains the address of the first actual storage location for the array, regardless of its subscripts. Whenever the array is referenced a dummy Dope Vector is created with its address derived from the pointer's value to reference the 0-subscript element.

Based string variables may not have the VARYING attribute because of the way PL-I deals with based string Dope Vectors. There is a single dummy Dope Vector, even though the string may have been ALLOCATEd into any number of locations. In normal string manipulations, PL-I needs the address of the Dope Vector, which indicates where and how long the string is. In based string manipulations, the base pointer contains the address of the start of the string itself, rather than the address of the Dope Vector. For each manipulation, PL-I creates a Dope Vector from the base pointer and the dummy Dope Vector, which contains only length information.

Because of this, it is somewhat difficult to retrieve a string datum from a list cell by means of based variables. If VSTR is a variable-length string, CELLAD is the address of a string-holding cell, PNTR is a pointer variable, and DADDR is a pointer and DSTR is a based string both with DPOINT as their base pointer, the following code would succeed:

```
PNTR = CONTS(CELLAD);
DPOINT = CONTS(PNTR);
VSTR = SUBSTR(DSTR, STRLNTH(PNTR- > DADDR));
```

On the other hand, this SLIP code would succeed also:

```
CALL DATUM(CELLAD, VSTR);
```

Appendix D

A SNOBOL-Like Function in PL-I

FLAG = SNOSCAN(variable number of character string arguments)

SNOSCAN does not strictly belong in a discussion of SLIP functions, but is included here because it was developed in conjunction with the SLIP effort and because it represents another facet of the attempt to make PL-I an even more general language.

SNOSCAN is intended to perform the major string manipulating functions of SNOBOL, and its syntax and abilities are adapted directly from SNOBOL to PL-I. Arguments to SNOSCAN are all character strings, and should generally be of varying length; they fall into four general categories:

1. The first argument is the string to be scanned (STR).

2. The second arguments are pattern strings or variable strings to be matched in STR. These arguments are optional and may be omitted.

3. The third argument is the string (of length 3) '"="'. It is optional also; if it is omitted, arguments of the fourth type are also omitted.

4. The fourth arguments are strings to be substituted in STR for the substring of STR successfully matched by the Type 2 arguments.

Briefly, SNOSCAN allows a string STR to be scanned for particular patterns; the part of STR which is matched may be left alone, may be deleted, or may be replaced by another set of patterns. SNOSCAN returns '1'B if all parts of the scan are successful; it returns '0'B if any part fails, although depending on the part that failed some or all of the string arguments may be successfully modified.

I. Pattern matching. The first part of SNOSCAN tries to find a substring of STR which matches the concatenation of the Type 2 arguments. (If there are no Type 2 arguments, the match is deemed successful with the matched substring all of STR.) Type 2 arguments may be either patterns or variables:

A. Patterns are nonnull strings of characters to be matched in STR. Patterns may have optionally an explicit Position, which says that the match can be successful only at a given distance from the start of STR. (This corresponds to ANCHOR mode in SNOBOL.) The Position, if explicit, is indicated by "xxx" at the start of the pattern; xxx is an unsigned string of decimal characters; there may be any number of characters in xxx. "1" corresponds to the first character in STR. (The match attempted begins with the first character after the Position indicator. If the user wishes to match a pattern which begins with the double quote ", he should repeat it: "". The first one is ignored in the scan.)

Sample patterns:

```
'ABCDEF'
' DYNAMITE '
'"53" DYNAMITE '
'""23 SKIDDOO"'
'""RATS," CRIED CHARLIE BROWN.'
```

The second and third patterns are identical except that the third requires a match starting at the 53rd character of STR. Note in the fourth and fifth patterns that only an initial double quote must be doubled; later ones are interpreted properly.

B. Variables are null strings which will be given nonnull length and values if the scan is successful. Variables are useful for three main purposes: assigning a value to a variable, skipping an unimportant portion of STR between two substrings which must be matched exactly, and looking for repeating patterns in STR via the <u>backreferencing</u> feature of SNOSCAN.

Variables may have optionally a Position, as for patterns, and a <u>Length</u>; thus a variable may not really be a null string: the important point is that it is null after any Position or Length specifications. Position is indicated as it was for patterns; Length is shown in the same way. Position always precedes Length; if one wants to indicate only Length, the Position number may be replaced by a single nonnumeric character (not ").

Sample variables:

```
''
'"57"'
'"57""13"'
'" ""13"'
'"X"'10352"'
```

The second example specifies a position in STR at which it must begin; the third example also specifies that the variable must be exactly 13 characters long. The fourth example specifies Length but not Position, as does the fifth. The fifth also points out the fact that strings may be up to the maximum allowed by PL-I.

Several simple examples will illustrate the kinds of pattern matching SNOSCAN can do:

FLAG = SNOSCAN(STR, A); (A is a pattern.) The string STR is searched for a substring (which may be all of STR) equal to A. If A has an explicit Position indicator, the match will be attempted only at the proper offset in STR; otherwise the match will be tried at all positions, starting at the left end of STR.

FLAG = SNOSCAN(STR, A, B); (A, B are patterns.) A match for A is looked for as described above. If successful, the next substring of STR is compared to B.

FLAG = SNOSCAN(STR, X); (X is a variable.) X is set equal to STR. If X has Position or Length restrictions, it is set equal to the specified substring.

FLAG = SNOSCAN(STR, X, X); (X is a variable.) SNOSCAN tries to find a repeating substring in STR, starting with the beginning of STR (unless X has an explicit Position, in which case X could only end up null), subject to any Length indicator in X. If successful, X is set equal to that substring; if not, X is made null and SNOSCAN still succeeds.

FLAG = SNOSCAN(STR, X, A); (A is a pattern; X is a variable.) STR is searched for A; if successful, X is made equal to the portion of STR from the beginning up to the beginning of A. (SNOSCAN fails if there is any contradiction of explicit Length or Position with what is actually found.)

FLAG = SNOSCAN(STR, A, X); (A is a pattern; X is a variable.) As in the previous example, except X is made equal to the substring of STR after the end of A.

FLAG = SNOSCAN(STR, A, X, B); (A, B are patterns; X is a variable.) STR is searched for A and the portion of STR after A is searched for B; if successful, X is made equal to the substring of STR between A and B.

FLAG = SNOSCAN(STR, A, X, B, X); (A, B are patterns; X is a variable.) STR is searched for A, the portion of STR after A is searched for B, and the portion of STR after B is searched (in ANCHOR mode) for the substring of STR between A and B. If all is successful, X is set to the value of that intermediate substring. This demonstrates the backreferencing feature. There may be any number of references to the same variable in a single call, and there may be any number of variables which are back-referenced in a single call.

II. Substitution. In all the examples considered up to now, SNOSCAN has only been asked to see if STR contains a particular pattern--although the specification of that pattern can become quite complex. By using the Type 3 argument "=", with or without Type 2 or Type 4 arguments, it is possible to change the value of STR.

Type 2 arguments serve to mark out the portion of STR to be modified. If no Type 2 arguments are present, all of STR is to be changed.

Type 4 arguments give the value of the character segment which will be inserted in place of the part of STR to be changed. If no Type 4 arguments are present, the part of STR to be changed is simply deleted.

It is possible to reuse some of the Type 2 arguments as Type 4 arguments in the same call. Variables which had to be matched are given their final value at the end of the scanning portion of SNOSCAN, so that they may be used to replace part of STR.

E.g., FLAG = SNOSCAN(STR, A, B, '"="', B, A); (A, B are patterns.) This, if successful, simply interchanges two portions of STR.

FLAG = SNOSCAN(STR, X,'"="', X); (X is a variable.) This sets X equal to STR (assuming no explicit Position of Length indicators) and replaces STR by itself.

SNOSCAN is intended to modify strings; therefore a few words of caution are in order.

1. One is best off using variable length strings if they are going to be changed by SNOSCAN. PL-I does not assume it knows the length of such strings, as it does with fixed length ones, and looks up the length each time. Also, SNOSCAN resets the length of strings it modifies; this could be hazardous if the string is really fixed length--SNOSCAN has no way of telling them apart at execution time.

2. Make sure the strings are of adequate length. With one exception, SNOSCAN fails as soon as it tries to substitute a character string that is too long for the maximum length of its destination string. The exception is in modifying STR itself. If SNOSCAN has succeeded up to the point of changing STR (the last thing it does), it will modify STR up to its maximum length and throw away the rest of the characters it was supposed to put in STR. Instead of returning '1'B, SNOSCAN will return '0'B to indicate that something went less than perfectly.