STRING AND LIST PROCESSING SAMPLER

L.K. Flanigan
W.E. Riddle

Department of Computer and
Communication Sciences

University of Michigan

Summer 1972

PREFACE

This compilation of notes and short papers is the final result of a seminar, entitled "String and List Processing", which was given during the summer of 1972 in the Department of Computer and Communication Sciences under the guidance of Professors Flanigan and Riddle. The intent of the seminar was to allow students to pursue interests in the general area of string and list processing languages and facilities and to report the results of their work to the seminar. Some twenty students participated in the seminar, which met once a week, for three hours, for presentations and group discussions. Each participant making a presentation also produced an outline for his talk which was made available at the presentation. In addition, those taking the seminar for academic credit were required to hand in a final report which extended their outlines. The outlines and final reports produced are here gathered into one volume. Further, those who reported on list and string processing languages also each produced a sample program which demonstrates the use of that language which they discussed; these sample programs have also been included.

The material presented here has been given minimal editing by Professors Flanigan and Riddle. We did attempt to correct all known logical errors in the content of each paper; we did not correct minor misspellings and puncuation errors which did not harm the understanding of a paper. In some instances the sentence structure is a bit ambiguous; as long as the correct interpretation could be determined, we left the original text unchanged. Note, however, that we did detach the sample programs from each paper and combined them into the final section of the volume; each paper still contains references to the "attached sample program", although it is not attached in the final version.

The following page shows the topics presented in the seminar together with the names of those who presented them. We wish to thank these participants for their work and their presentations, which provided an enjoyable and informative seminar. We also wish to thank the additional seminar participants who, although they did not make presentations, provided a responsive audience and made the seminar even more informative. We were pleased with the seminar and enjoyed participating in it; we hope this printed record of the seminar will carry some of the flavor of the real thing.

L.K. Flanigan
W.E. Riddle

STRING AND LIST PROCESSING SEMINAR

I. Language Externals (5/18, 5/25)

SLIP,IPL/V	Lucier
PL/I	Unger
LISP	Flanigan
SIMSCRIPT	Rinn
POP-2	Brindle

II. Virtual Machine Capabilities (5/31) Riddle

A. Data Types

B. Order Codes

C. Dynamic Storage Management

III. Language Internals (5/31, 6/8, 6/15)

LISP	Hafner
PL/I	Hoffman
SNOBOL	Lift
SIMSCRIPT	Henriksen

IV. Selected Applications (6/22, 6/28)

UMMPS	Hamilton
GRAPHICS	Bauer
STDS	Katz

TABLE OF CONTENTS

W. E. Riddle
5/19/72

An Introduction to the IPL-V Programming Language

SYMBOLS:

A. GLOBAL SYMBOLS

- unique throughout program
- a letter followed by (at most) 4 digits, e.g. A14
- two types

  1. pre-defined

     HXXXX accumulators and special registers
     e.g. H0 -"communication cell"- push-down-stack general-purpose register
     H1 -instruction address register, also push-down
     H2 -head of available space list (user never has to use)
     H5 -condition code register, value either + or -

     WXXXX working, temporary storage

     JXXXX names of basic routines (more on this below)

  2. user-defined - global symbols, other than those starting with H,W, or J; used as identifiers

B. LOCAL SYMBOLS

- have the form 9-XXXX
- used as names of sublists
- scope is local to list where used

C. ADDRESSES

- numbers which are storage addresses
- provided by system (in examples below, blank indicates occurrence of a system provided address)

DATA TERMS: hold a piece of data

P	Q	DATA
0 - decimal integer 1 - decimal flt-pt 2 - alphanumeric 3 - octal	1	value

Q = 1 flags this as data in context where data is being expected.

CELLS: have a name and hold a value

A. format

P	Q	SYMB	LINK

SYMB is the item stored in the cell

LINK is a pointer to another cell

Q is "designation prefix" and indicates amount of indirection needed to get value of cell
  Q = 0 value is SYMB itself
  Q = 1 value is SYMB field of cell named by this SYMB field (i.e. one-level indirection)
  Q = 2 two-level indirection
P is "operation prefix", explained below.

B. <u>Any</u> cell is a push-down stack.

C. cell can hold an instruction

P	Q	SYMB	LINK
operation code	indirection indicator	symbol	link address

notation: let S be the symbol or value obtained from SYMB by indirect addressing.

operation codes:

0-EXECUTE S	-take next instruction from location S (push S onto H1)
1-INPUT S	-push S onto H0
2-OUTPUT S	-pop a symbol from H0 and put it into SYMB field of S
3-RESTORE S	-pop the pushdown stack S
4-PRESERVE S	-push down S (i.e. make it possible to store a symbol in S without destroying the present contents)
5-LOAD S	-replace current symbol at top of H0 by S (don't push)
6-STORE S	-replace SYMB field of S by symbol at top of H0 (don't pop)
7-BRANCH S	-take next instruction at S if H5 is -; otherwise, next instruction is at LINK.

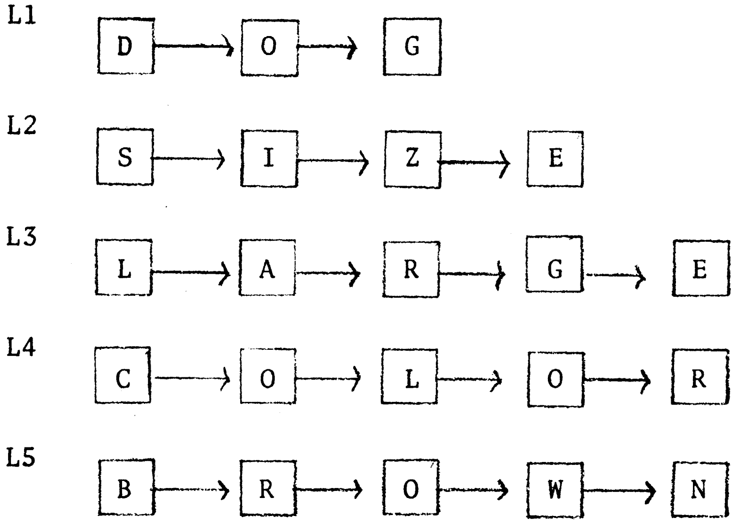
D. cell can be an element of a list.

- last element on list has LINK value zero
- an element on the list may be interpreted as an instruction; or it may merely be a symbol, naming some data item, instruction, or list.

- the first element of a list is a special header cell
  - LINK points to cell holding first item on list; zero indicates empty list.
  - SYMB is usually zero; when non-zero it names a list which is interpreted as holding the values and names of various attributes of the list.

EXAMPLE LISTS

I.



```
L1          0
            9-1
            9-2
            9-3  0
9-1   21    D
9-2   21    O
9-2   21    G
```

The blank LINK fields actually hold addresses of next sequential element. Another way to write the list, using explicit links, would be:

```
L1     0      9-5
9-4    9-2    9-6
9-5    9-1    9-4
9-6    9-3     0
        :
```

The other lists would be similar.

II. The list

```
L6     9-4
       9-1
       9-2
       9-3  0
9-4    0
       L2
       L3
       L4
       L5   0
       9-1    21    D
       9-2    21    D
       9-2    21    G
```

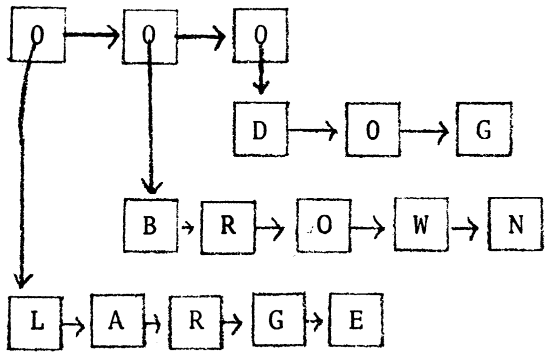
is the same as L1 except the attributes (SIZE, LARGE) and (COLOR, BROWN) have been associated with it.

III. The list structure

```
L7   0
     L3
     L5
     L1   0
```

would diagrammatically look like:

L7



ROUTINES

. list of cells which are interpreted as instructions

. pre-defined routines are named JXXXX and are part of IPL system (most all are machine code)

notation: Let items in H0 stack be named 0, 1, 2, ... from the top. Then the symbol in SYMB field of top item on H0 is (0), etc.

```
J0       noop
J1       execute (0)
J2       test if (0)=(1)  N.B. test symbol only, set H5 + if yes, - if no
J3       set H5 -
J4       set H5 +
J5       reverse H5
J6       interchange (0) and (1)
 .
 .
 .
J110     V(1) + V(2) → V(0)
 pop up ((3) → (1)
 two    {(4) → (2)           V(0) is value of symbol in SYMB field of top of H0
 items  (     .
              .
              .
J111     V(1) - V(2) → V(0)  and pop up two items
J112     V(1) * V(2) → V(0)  and pop up two items
J113     V(1) / V(2) → V(0)  and pop up two items
J114     test if V(0) = V(1)
J115     test if V(0) > V(1)
 .
 .
 .
J121     V(1) → V(0)  and pop up one item
 .
 .
 .
```

EXAMPLE IPL-V PROGRAMS

These programs calculate n!, the first one iteratively and the second one recursively.

A. Assume that C1 has the value 1, M1 contains n, and A1 is to receive answer.

```
F1    50   M1      M1 → (0)               )
           10   A1      A1 pushed onto H0      }      A1 = M1
      00   J121    V(M1) → V(A1)          )
9-1   50   C1      C1 → (0)               |
      10   M1      M1 pushed onto H0      |      IF(M1.EQ.1) GO TO 9-2
      00   J115    Test V(M1) > V(C1)     |
      70   9-2     Branch if false        |
      50   C1      C1 → (0)               |
      10   M1      M1 pushed onto H0      |      M1 = M1-1
      10   M1      M1 pushed onto H0      |
      00   J111    V(M1)-V(C1) → V(M1)    |
      50   A1      A1 → (0)               |
      10   M1      M1 pushed onto H0      |      A1 = A1*M1
      10   A1      A1 pushed onto H0      |
      00   J112    V(M1)*V(A1) → V(A1)    |
      00   J3      Set H5 to -            }      GO TO 9-1
      70   9-1     Branch if H5 is -      }
9-2   00   J0   0  Noop                   }      END

  IPL-V Language     Annotation                 Equivalent FORTRAN
```

B. Assume that: C1 contains the value 1; M1 contains X1, the <u>name</u> of the cell with the value n; and M2 contains X2, the <u>name</u> of <u>the</u> cell where the answer is to go.

```
F1    50   C1        C1 → (0)                 |     Initialize value
      11   M2        (M2) pushed onto H0      |     of X2 to 1 and
      00   J121      V(C1) → V(X2)            |     put X1 on top
      51   M1        (M1) → (0)                     of stack.
      00   F11   0

F11   40   W1        preserve V(W1)           )     Push W2's (X1's
      10   W1        W1 pushed onto H0        }     the first time thru)
      00   J121      V(X1) → V(W1)            )     current value onto W1
      10   C1        C1 pushed onto H0        )     If value pushed
      00   J114      Test V(W1)=V(C1)         }     was 1 then stop
      70   F12   0   Branch if false          )     recursing down.

F12   10   C1        C1 pushed onto H0        |     Set W2's value to
      10   W1        W1 pushed onto H0        |     W1's current value
      10   W2        W2 pushed onto H0        |     minus 1 and leave the
      00   J111      V(W1)-V(C1) → V(W2)      |     symbol W2 on H0.
      00   F11                                }     Keep recursing
      30   W1        Restore W1's value       |     Form next product
      51   M2        (M2) → (0)               |     in the series
      10   W1        W1 pushed onto H0        |     1·2·3·4·5·...·n
      11   M2        (M2) pushed onto H0      |     and put its value
      00   J112  0   V(W1)*V(X2) → V(X2)      |     in X2

  IPL-V Language       Annotation                     In English
```

MTS SLIP

Dean W. Lucier
June 23, 1972

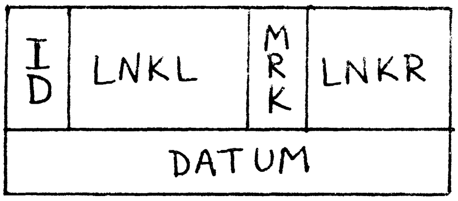
SLIP (Symmetric List Processor) was developed by Joseph Wizenbaum of M. I. T. while he was with General Electric in 1963. He didn't hide the fact that SLIP was derived from at least four earlier list processors, including IPL-V. From Newell's IPL-V, he used the concept of a list of available space and many of the SLIP functions are the same as basic processes in IPL-V. While SLIP was originally developed for use on the IBM 7090, the principles remain the same for MTS, even though the cell structure has changed. The descriptions herein will be for the MTS implementation.

SLIP is a list processing system distinguished by the symmetry of its lists; each element (cell) has both a forward and backward link as well as a datum. It is not an independent language, but is intended to be embedded in a high-level language like FORTRAN which gives it great flexibility. Most SLIP processes (functions) are written in FORTRAN, while the processes which actually deal with the cell structure, called primitives, are written in assembly language.

The description of SLIP can be divided into two parts:

1. The data structure which contains the information to be manipulated.
2. The program structure which is the means of carrying out the manipulations.

First, the cell structure as designed by Wizenbaum and implemented on MTS consists of a pair of consecutive doublewords.



SLIP CELL

The ID field is 3 bits long and designates what type of cell it represents by its numeric value:

ID = 0 : this is a normal cell containing a piece of data.

= 1 : the datum is the name of a list.

= 2 : this cell is the header of a list.

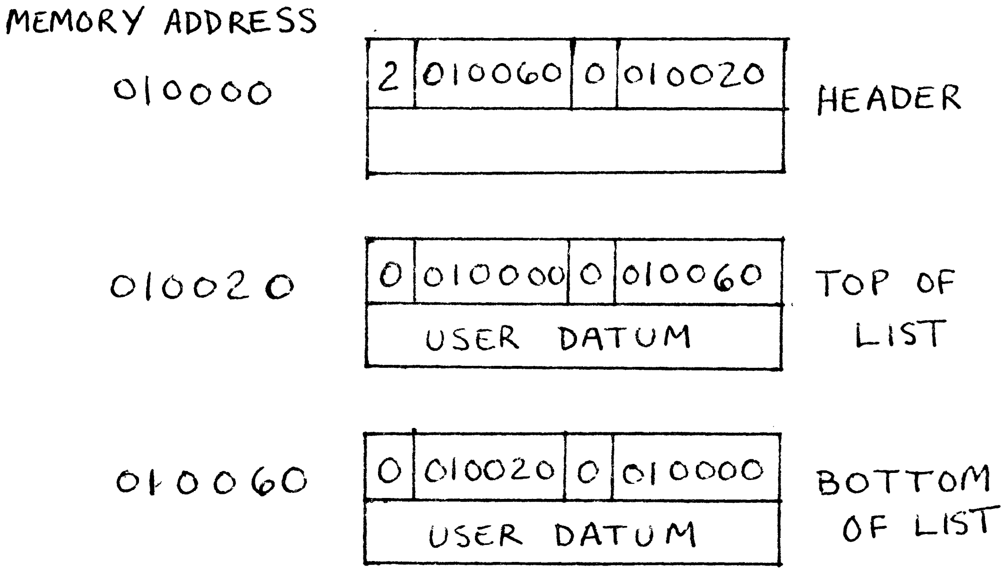
= 3 : this cell is a reader of a list.

The LNKL and LNKR fields contain the addresses of the predecessor cell and successor cell, respectively. On every list there is exactly one header (ID=2) cell. Therefore, an empty list consists of one header cell which points to itself in both the LNKL and LNKR fields.

The MRK field is 3 bits long and contains whatever information the user designs. It is not used by the SLIP routines.

The DATUM field is 64 bits long, (one doubleword or two full-words or eight bytes). In type 0 (ID=0) cells, it may contain anything which the user can fit into 64 bits.

A simple list looks like the following diagram:



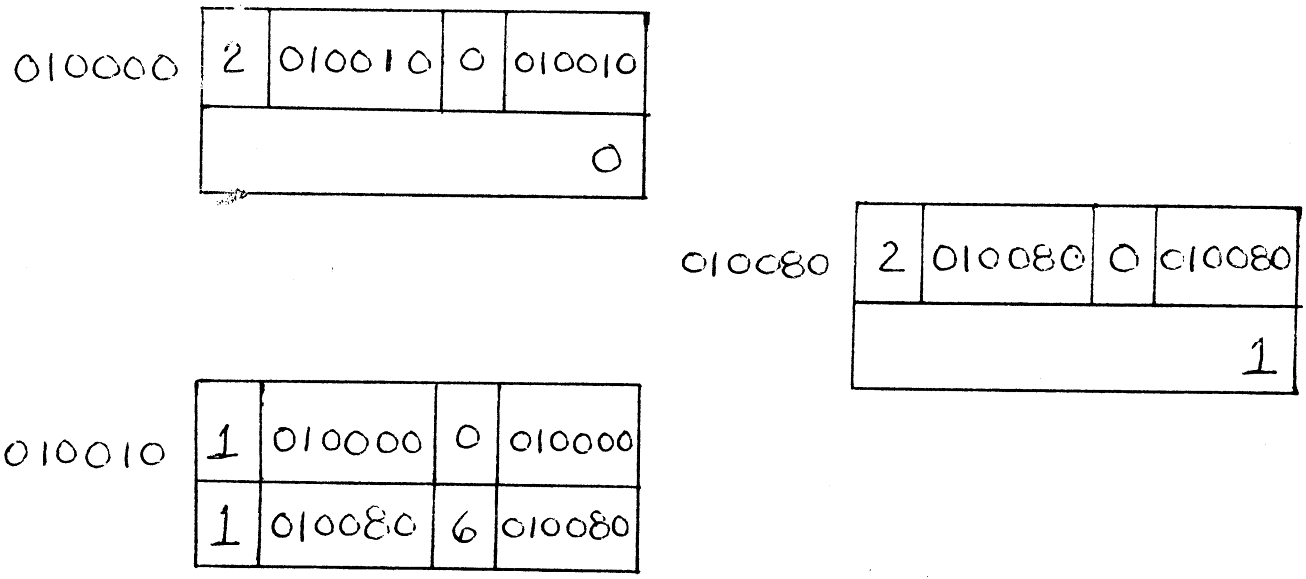
Much of the power of list processing systems is drawn from the ability to process list structures. Therefore, the capability of creating one list and linking it to another list, forming a sublist structure, must exist.

Recall that a name cell (ID=1) has the name of a list in its datum field. In the MTS implementation, the datum of a name cell is divided into 4 fields as shown:

1	LINK TO HEADER	6	LINK TO HEADER

DATUM OF NAME CELL (ID=1)

The first and third fields are three bits long each and are constant. The second and fourth fields are 29 bits each and each contains the address of the same header cell, thereby providing the link between lists. An example of a sublist structure is the following:



010000

2	010010	0	010010
			0

010080

2	010080	0	010080
			1

010010

1	010000	0	010000
1	010080	6	010080

One might raise the question that when adding a cell where does it come from? In SLIP, there exists a list of available space (LAVS) which is actually a one-way linked list, i.e. when cells are returned to the list, they are linked to the bottom, and when they are retrieved, they come from the top of the list. This is possible since SLIP has two reserved fields within its language code which maintain the current addresses of the top and bottom cells of the LAVS.

This scheme enables one to erase an entire list rather than each individual cell. This is due to the reference count field of the header cell of a list. The reference count is maintained as the DATUM portion of a type 2 cell. Each time a list becomes a sublist of another list, this reference count is incremented by one. If the "parent" list is erased and thereby attached to the LAVS, the routine for obtaining new cells from the LAVS will decrement the reference count by one[1], and if the result is zero, then the sublist is erased and linked to the bottom of the LAVS. If the result is greater than zero, it still remains as a sublist to some other list, and is not linked to the LAVS.

These processes bring up the question of who is responsible for maintaining the pointers in the structures and the handling of LAVS and other niceties? SLIP has the routines to handle these responsibilities and therefore all the user has to do is ask for a list to be created or a cell to be inserted or erased. SLIP will obtain the cell or list, process the reference count, update pointers, etc.

[1] at the time the name cells in the "parent" list are reused.

Finally, as part of the data structure services provided by SLIP, it provides working storage cells which can be used as common areas for all user subroutines in which information can be transfered. There are 100 of these cells in consecutive doubleword pair locations, consisting of 100 empty lists.

Looking at the routines in the program structure part of SLIP, they consist of certain assembly language routines, which are called primitives, and the other routines which are written in FORTRAN, and are normally functions. The primitives extract directly the information from the cell and store the information into the cell and, consequently, are machine and cell structure design dependent, whereas the other high level language routines manipulate the lists by calling the primitives where needed. For example, a primitive must be used to obtain the memory address of a cell.

Therefore, one can see how flexible SLIP is in that only the primitives, approximately 12 in all, need really be rewritten if the cell design is altered or a different make of computer is used.

Like other list processing systems, SLIP allocates storage dynamically. Therefore, before any lists can be built, the LAVS must be initialized. This is done by the routine INITAS, acting upon a block of storage provided by the user.

The type of nearly every SLIP quantity and function should normally be DOUBLE PRECISION and explicitly declared for each SLIP function called by the user in his program. A few functions should be declared as INTEGER. Therefore, it is a good practice to refer to CC MEMO #M197 until one becomes familiar with the functions.

The following is an example for setting up a list of available space:

```
DIMENSION   SPACE (1000)
COMMON      AVSL,W(100)
I = 1000
CALL INITAS (SPACE,I)
```

This will create an LAVS of 500 cells and a set of empty lists W(1)...W(100) which can be used as public storage. The COMMON statement should be included in any user program.

In order to create a list (empty initially), there are 3 types of statements which can be used:

```
1       CALL  LIST (LA)
2       LB =  LIST (LA)
3       LB =  LIST (9)
```

The value returned by LIST is the address of the header cell created, stored in a double word. Therefore, the DOUBLE PRECISION symbolic variables, LA and LB, will contain this address after execution and can be used to refer to the list just created, for that reason.

The argument, integer 9, causes the reference count to be set to zero in the new header cell. Otherwise, when 9 is not used, the reference count is initialized at 1. As noted before, the reference count is incremented by one when it is made a sublist of another list. Therefore, sublists initially should have a zero reference count, so that when linked as a sublist, resulting in an updating of the reference count, they will be erased if the "parent" list is erased. If the reference count of a sublist is initially one, and updated to two, the erasing of the main list will result in the reference count decrementing by 1 and not reaching zero, the necessary requirement for erasure.

The statement used to erase a list is:

```
CALL IRALST (LA)
```

This decrements the reference count of a list, LA, by one and if the result is 0, it will erase it by linking it to the bottom of the LAVS.

There are a number of functions for placing cells on an existing list. In all cases, the value returned is the address of the new cell obtained.

```
STUFF = NEWTOP(DATUM,LA)
STUFF = NEWBOT(DATUM,LA)
```

The above statements will cause a new cell to be inserted at the top or bottom of list LA, and the datum portion of the field of the new cell to be filled by the contents of DATUM. A precaution to note here is that SLIP checks the contents of DATUM to determine if it is the address of a header cell. If it is, the cell being created becomes a name cell (ID=1) and the appropriate format is constructed. Therefore, the user must be sure there will be no conflict with his data and the addresses which could be generated. In most cases, this is not a problem. For example, the attached sample program uses the datum field in a manner in which the conflict should not arise.

Suppose a cell needs to be inserted in a location on the list which is neither the top or bottom. This can be accomplished with either of the following two statements:

```
STUFF = NXTRGT(DATUM,KEEP)
STUFF = NXTLFT(DATUM,KEEP)
```

The first statement will obtain a cell from LAVS, and then insert it to the right (successor) of the cell whose address is contained in KEEP. Then the datum is stored in the new cell. Similarly, NXTLFT refers to the insertion left of the existing cell using the predecessor (LNKL) pointer.

The information contained in a cell can be changed by using:

```
STUFF = SUBST (DATUM,KEEP)
```

This results in the contents of DATUM being placed in the cell whose address is in KEEP, and then the old contents of the cell are put in STUFF. Similarly, the following functions will place data in the top and bottom cells of a list:

```
SUBSTP (DATUM,KEEP)
SUBSBT (DATUM,KEEP)
```

There exist other extravagant routines in building lists:

```
LB = LSSCPY(LA)
```

A list, LB, is created and is an exact copy of the list, LA, except for addresses.

```
INLSTL (LA,KEEP)
INLSTR (LA,KEEP)
```

These reoutines insert an entire list (LA), except for the header cell of LA, as the predecessor (left) or successor (right) of the cell whose address is in KEEP. The list LA then becomes an empty list. For example, the bottom cell of list LA is linked to the cell specified by KEEP and the top cell of list LA is linked to the predecessor if INLSTL is used.

```
CALL DELETE (KEEP)
```

This routine will erase the cell specified by KEEP and return as a value the contents of the datum field of the cell.

The actual searching of a list is done by SLIP, after receiving a command from the user program. Therefore, in order to perform these read functions, SLIP requires a cell to be acquired from LAVS which he uses to keep track of the current cell in the list structure where the user is positioned.

The reader cell (ID=3) has the following format:

<table>
<tr><td>ID</td><td>LPNTR</td><td>MRK</td><td>LINK</td></tr>
<tr><td colspan="2">LOFRDR</td><td colspan="2">LCNTR</td></tr>
</table>

READER CELL

The LPNTR field contains the address of the cell currently being pointed to. The LOFRDR contains the address of the header cell of the list currently being looked into. The LCNTR is a count of how many levels ("depth" of sublists) the reader has descended into the list structure. The level of the main list is represented by 0. Each time the reader descends into a sublist, he obtains a new reader cell and forms a stack by using the LINK field after bumping the level counters of previous readers in the stack.

There are two types of scanning or advancing operations available in SLIP, linear and structural. In both cases, a reader for the list must be obtained by using:

```
READER = LRDROV (MAIN)
```

READER contains the address of the initial (in case of a stack) reader cell for the list specified by MAIN.

The mnemonics for the advance operations represent 3 parameters:

1. type of advance
2. type of cell being searched (target)
3. direction of advance

For example:

```
STUFF = ADVLEL (READER,FLAG)
```

The first letter L signifies linear advance, whereas S is structural. The second letter L represents in the left direction (using predecessor links), whereas R is in the right direction. The target specification is the letter E which signifies a search for type 0 cells (ID = 0). The letter N represents name cell search and the letter W represents either type.

The argument READER represents the reader cell for the list to be searched and FLAG designates a DOUBLE PRECISION variable which will be set to zero or non-zero depending on whether the search is successful or not.

The following are some of the advancing operations:

```
ADVLER (READER,FLAG)
ADVLNL (READER,FLAG)
ADVLWR (READER,FLAG)
ADVSEL (READER,FLAG)
ADVSNR (READER,FLAG)
ADVSWL (READER,FLAG)
```

For the linear advance, if the search is successful, the datum of the matching cell is returned as a value and the flag is set to zero. If a header cell is encountered before a match, the flag is set to non-zero and the value returned is zero. The linear advance moves sequentially down or up a list without descending into the sublists.

The structural advance will descend into the sublists if a name cell is encountered and the target cell has not been found. If the target cell has not been found in the sublist, the reader returns to the point of descension in the "parent" list and continues the search from there. All this is done by SLIP before control is given back to the user program. The flag is only set to non-zero, implying failure to find the target, if the header cell of the main list is encountered.

Some of the other routines concerning the reader are:

REED (READER) returns the datum of the cell which READER is currently pointing to.

INITRD (READER) forces the reader to point to the head of the current list.

LVLRVT (READER) returns to main list at the point (cell) where it descended into the sublists.

INITRD (LVLRVT (READER)) returns reader to header cell of main list.

IRARDR (READER) the reader is erased and returned to LAVS.

Looking at the attached program listing and results, the following functions are used:

```
          CALL SETRAC(SPACE,I)

          CALL SLPDMP(SPACE,I)
```

If the LAVS is exhausted and SETRAC has been previously called a SLIP DUMP will be produced. This dump (not a core dump) is formatted so that tracing through the list structure is easily accomplished, and also through the reader stack to determine which cell was currently being pointed to at the time.

The function SLPDMP explicitly calls for a SLIP DUMP and is good for debugging purposes. Another debugging tool is the function F4TRBK, whcih has no arguments, but will receive control on any type of error, and produce a SLIP DUMP before giving control back to MTS.

Some of the advantages of SLIP stem from its feature of being written almost entirely in a high level language. This makes it essentially machine independent, and easily imbedded into a language like FORTRAN. There is no extra compile time since the routines are loaded at execution time along with the user modules, which also makes it easy on core since only those modules used need to be loaded.

In addition, the two-way linked lists allow lists to be bulk erased rather than each individual cell. The retrieval time for the bottom cell on a list is the same as the top cell and is independent of the length of the list.

I feel SLIP would be quite valuable to a number of high level languages like PL/1 or COBOL where the user must perform all the operations required in list processing. Certainly, if the type of list structure SLIP produces fits the application or vice versa, having SLIP available would be ideal.

References

1. A. Newell and F. M. Tonge, "An Introduction to Information Processing Language V", Programming Systems and Languages, edited by Saul Rosen.
2. J. Wizenbaum, "Symmetric List Processor", Comm. ACM, p. 524-544, 1963, no. 6.
3. Douglas K. Smith, "An Introduction to the List Processing Language SLIP", Programming Systems and Languages, edited by Saul Rosen.
4. ComputingCenter Memo #M197, University of Michigan, 10-19-71.

CCS 500
Spring 1972
J.A.Unger

STRING AND LIST PROCESSING FACILITIES OF PL/1

- AN EXTERNAL VIEW -

A. INTRODUCTION

When considering the string and list processing facilities of PL/1, the potential user must remember that PL/1 is a general purpose language and not an amalgam of many special-purpose languages. Thus, although PL/1 can be used to solve almost any type of problem, in general there will be a "better" language for the solution of any particular problem. However, if the application to be implemented encompasses several functional areas having, for example, list processing and computational requirements, then PL/1 with its wide range of capabilities may be the most appropriate implementation vehicle.

Because it is a general purpose language, the various "primitives" usually are exceedingly primitive, and the programmer is often forced to write detail coding for routines which would have been provided had a different language been chosen. This caveat is especially true when the list and string processing facilities of PL/1 are being compared to the facilities of other string and list processing languages: a few powerful, basic manipulations are provided, and it is the responsibility of the programmer to use these building blocks to create his own processing routines.

B. STRING PROCESSING FACILITIES

The following is a minimal set of operations required for the definition and manipulation of string data:

- Definition and initialization of variables and constants
- Assignment of values
- Combination of strings
- Selection of substrings by position or content
- Determination of the current length of a string

The following PL/1 features form the basic support for the above set of operations:

1. String Data Attribute - It is possible to define an identifier as string type data, either CHARACTER or BIT, to define its length and whether this length is fixed or VARYING, and to establish an initial value for the string.

2. String Data Operators - The assignment (=) and concatenation (||) operators, the logical operators (¬,&,|), and the comparison operators (¬=, =, etc.) can be used with string data as operands.

3. String Built-in Functions - Of the approximately 10 built-in functions that operate on string data, the most important and most useful are INDEX, LENGTH, and SUBSTR.

Before any string manipulation examples can be discussed, a few comments should be made about the more important operations and functions. The assignment operation is the copying of data from a source to a receiving string with any necessary padding/truncation on the right. If a fixed length receiving field is longer than the source field, then blanks will be inserted on the right if it is of CHARACTER type and binary zeroes will be inserted on the right if it is of BIT type. If a VARYING receiving field is longer than the source field, no padding will occur since the current length field within the receiving string's dope vector will be set accordingly. If the length of the receiving field is less than that of the sending field, then the excess digit/bits will be truncated. There is no truncation interrupt, so the programmer must test for possible truncation if such an occurrence would cause an error.

Concatenation is the forming of one composite string from 2 or more strings; it can be thought of as a 2-step operation, the first being the formation of a temporary string and the second being the assignment of that temporary string to the indicated receiving field. All the rules of assignment apply, and unintentional error may occur. For example, the following statements

```
DECLARE S1 CHARACTER (3) INITIAL ('ABC'),
        S2 CHARACTER (2) INITIAL ('DE'),
        S3 CHARACTER (5);
S3 = S1||S2;
```

result in string S3 containing 'ABCDE' at the completion of their execution. Note that if S1 were defined

```
DECLARE S1 CHARACTER (5) INITIAL ('ABC'),
```

then the statement

```
S3 = S1||S2;
```

results in S3 containing 'ABC ' because S1 would have been padded with blanks during initialization and the temporary string 'ABC DE' would have been truncated upon assignment.

The SUBSTR function is the selection of a portion of a string based on its relative position within that string. The general form of the function is

```
SUBSTR (string,start[,length])
```

where "string" is the name of the string to be operated upon, "start" indicates the starting location relative to the first position in the string (which is position 1), and the optional "length" indicates the number of units to be in the resultant substring. For example, if string STR were declared to contain 'ABCDE' then the statement

```
SUB1 = SUBSTR (STR,3,2)
```

would result in SUB1 containing 'CD'. SUBSTR is one of a class of function which can be used on the left-hand side of an assignment statement, and, when so used, is known as a pseudo-variable. Given the above definition of STR, if the statement

```
SUBSTR (STR,3,2) = 'EF';
```

were executed, then the result in STR would be 'ABEFE'.

The INDEX function is used to locate a character or bit configuartion within the specified string. A numeric value indicating the starting position of the pattern is returned if the pattern is found; if it is not found, the value returned is zero. For example, if it were desired to locate the first blank character in string S, the following statement could be used

```
LOC = INDEX (S,' ');
```

The following examples demonstrate typical string processing manipulations and techniques:

Example 1. The following code deletes all blanks from a string S by constructing a new string consisting of the P characters to the left of the blank concatenated with the characters which lie to the right of it. Statement numbers 1 and 2 will be repreatedly executed until no blanks remain in the string (a return of 0 from the INDEX function).

```
DO WHILE (INDEX (S,' ')¬= 0);
     P = INDEX (S,' ') - 1;
     S = SUBSTR (S,1,P)||SUBSTR (S,P+2);
     END;
```

Example 2. The following code reverses the characters in string S by treating the string as two symmetric halves and interchanging the contents of the corresponding character positions. The loop control is L/2 since 2 characters are being moved on each iteration. Note that if L is odd the integer arithmetic used in determining the loop counter results in truncation but that the center character need not be interchanged and thus no error occurs.

```
L = LENGTH(S);
DO I = 1 TO L/2;
     T = SUBSTR (S,I,1);
     SUBSTR (S,I,1) = SUBSTR (S,L-I+1,1);
     SUBSTR (S,L-I+1,1) = T;
     END;
```

Example 3. The following code performs initial verification of an input message and has been extracted from the primary message-handling module of an on-line system. When this module is invoked, a message has already been received and assembled in TRANSACTION and its length has been inserted in LENGTH. This code searches for the first non-blank character in the message; if more than one character, special processing is performed. Otherwise, the 1-character transaction code is extracted from TRANSACTION, and its validity is checked by comparing it to the allowable codes in TRANSACTION_CODES. If it is valid, the non-sero value returned from the INDEX function is used as a subscript into the LEGAL_IN_PHASE and LEGAL_FOR_USER bit tables.

```
DECLARE  TRANSCATION CHAR(L_TRAN) EXTERNAL,
         LENGTH FIXED BINARY EXTERNAL,
         TRANSACTION_CODES CHAR(24) INITIAL
              ('BSVCDYGFRPMLIJNAQWTEZUXK') STATIC,
         LIP_TAB (0;4) BIT (24) UNALIGNED INITIAL
              ('000000000000001000001111'B,
               '111111111111101111111111'B,
               '111111111111101111111111'B,
               '001111111111111111111111'B,
               '000000000000000000000000'B) STATIC,
         LEGAL_IN_PHASE (0;4,24) BIT(1) UNALIGNED DEFINED LIP_TAB,
         LFU_TAB (2) BIT (24) UNALIGNED INITIAL
              ('111111000101111111111111'B,
               '111111111111111111111111'B) STATIC,
         LEGAL_FOR_USER (2,24) BIT (1) UNALIGNED DEFINED LFU_TAB,
          (PHASE, USER_TYPE) FIXED BINARY;

SUBSTR(TRANSCATION,LENGTH+1,1) = ' ';
DO I = 1 TO LENGTH-1 WHILE (SUBSTR(TRANSACTION,I,1) = ' ');  END;
J = INDEX (SUBSTR(TRANSACTION,I+1),' ');
IF J  = 1 THEN GO TO BOX9;
XACT_CODE = SUBSTR(TRANSCATION,I,1);
I = INDEX (TRANSACTION_CODES,XACT_CODE);
IF  I = 0 THEN GO TO ERROR8;
IF  LEGAL_IN_PHASE (PHASE,I) THEN GO TO ERROR9;
IF  LEGAL_FOR_USER THEN GO TO ERROR 10;
```

C. LIST PROCESSING FACILITIES

In order to implement list processing applications, the following general types of capabilities are necessary:

- Element definition
- Dynamic creation/deletion of elements
- Modification of element contents
- Selection of list elements according to contents or relative list position

Since PL/1 was not intended to be specifically a list processing language (in fact, the list processing facilities were added after the balance of the language design was substantially complete), it includes only the basic list processing mechanisms, leaving to the programmer the coding of many necessary functions. For example, PL/1 does not provide a routine which automatically adds an element to a list. However, it does provide programmer-accessible pointer data and, by permitting unlike data items to be combined into a logical unit, allows the programmer to design and code his own list building routine.

The BASED storage allocation data attribute specifies that storage is to be assigned to its identifier when that identifier is explicitly allocated (rather than when its containing block is activated) and that that storage is to be released when that identifier is explicitly freed (rather than when its containing block is terminated). Thus, the statement

```
DECLARE 1 ITEM BASED (I_PTR),
```

can be used to identify attributes associated with ITEM and its substructure while providing programmer control over ITEM storage allocation. When a BASED variable is allocated, storage is obtained for an instance of that variable and a pointer is set to locate the beginning of the obtained storage. For example, the statement

```
ALLOCATE ITEM;
```

results in an instantiation of ITEM and the setting of I_PTR to the starting location of this core. It is possible to explicitly name a different pointer variable to be set when an instance of a BASED variable is allocated. For instance,

```
ALLOCATE ITEM SET (P);
```

causes pointer variable P to point to the first byte of the obtained storage, leaving I_PTR undisturbed. The statement

```
FREE ITEM;
```

causes the storage located by the current contents of I_PTR to be released, while

```
FREE P->ITEM;
```

causes the instance of ITEM located by P to be released.

Associated with each reference to a BASED variable, then, is either an explicit or implicit reference to a pointer variable. If only the BASED variable is named in a statement, the implication is that the desired instance of the variable is located by the pointer variable on which it is based. If a different instance of the variable is desired, then an explicit pointer reference must be made.

The following examples demonstrate typical list processing manipulations and techniques:

<u>Example 1.</u> The following user code builds a one-way linked list of ITEM elements. Since pointer variables contain addresses, it is necessary to define a bit configuration that is guaranteed to be the address of nothing. This is the NULL configuration, X'FF000000'. Since there must be some way to recognise an empty list or the end of a list, by convention the NULL pointer indicates that there are no following elements.

```
DECLARE 1 ITEM BASED (I_PTR),
          2  DATA CHAR(50),
          2  NEXT POINTER,
        (I_BASE,I_PREV) POINTER;
 ⋮
I_BASE,I_PREV = NULL;
 ⋮
ALLOCATE ITEM;
IF I_BASE = NULL THEN DO;
                      I_BASE = I_PTR;  I_PREV = I_PTR;
                      END;
                 ELSE DO;
                      I_PREV->ITEM.NEXT = I_PTR;  I_PREV = I_PTR;
                      END;
ITEM.DATA = STUFF;
ITEM.NEXT = NULL;
 ⋮
```

One of the drawbacks of dynamic list element allocation is that it is often difficult if not impossible to predict the amount of storage required for a particular program execution; reserving a fixed amount may result in wasted space, or, conversely, the reserved core may not be sufficient. By specifying that a particular area is to be used for the allocation of BASED variables, the PL/1 list processing programmer is able to control the amount of storage allocated to a list and can take corrective action if it is exceeded. The following declaration

```
DECLARE I_AREA AREA;
```

reserves I_AREA for the allocation of BASED variables, and the statement

```
ALLOCATE ITEM IN (I_AREA);
```

causes space for one instance of ITEM to be obtained from that associated with the variable I_AREA. As before, it is possible to set a pointer other than that named in the declaration of the BASED variable:

```
ALLOCATE ITEM IN (I_AREA) SET (P);
```

If no size is specified in the area declaration, a default size of 1000 bytes is assigned. The amount of AREA storage can be explicitly specified either absolutely or relative to the BASED variable(s) it is to contain. For example,

```
DECLARE I_AREA AREA (5000);
```

causes 5000 bytes of area storage to be associated with I_AREA, while the statement

```
        DECLARE I_AREA AREA ((100) ITEM);
```

causes sufficient storage to contain 100 instances of ITEM to be associated with I_AREA.

If space is not available within the specified area, the AREA ON condition will be raised, and it is the responsibility of the programmer to intercept this interrupt and to take appropriate action. Thus, at some initial point in the program should be a statement such as

```
        ON AREA GO TO EMPTY_AREA;
```

so that if an attempt is made to allocate an ITEM that cannot be contained in I_AREA, control will be transferred to the designated routine.

By combining the BASED and AREA attributes, it is possible to implement an OFFSET linkage scheme. (An OFFSET locator variable gives the displacement past the beginning of a named based variable.)

Example 2. The following code builds a one-way linked list using an offset linkage scheme. Note that the based area A must be explicitly allocated because storage is not associated with it at prologue time; this is in contrast to Example 1 in which I_AREA was declared with the default attribute of AUTOMATIC and had storage associated with it when its containing block was entered. Note also that the declaration of the OFFSET variables, NEXT and I_BASE, must specify the based area referenced by the offset. Two other comments should be made about this example; the first is that in the explicit pointer reference, LINK→ITEM.NEXT = I_PTR, LINK must be a pointer and must contain a pointer value. The second is that an offest variable can be set only by assignment of a pointer variable to it.

```
    DECLARE A AREA BASED (A_PTR),
            1 ITEM BASED (I_PTR),
              2  DATA CHAR (50),
              2  NEXT OFFSET (A),
            LINK POINTER,
            I_BASE OFFSET (A);
    ⌇
    ALLOCATE A;
    I_BASE = NULL ;
    ON AREA GO TO EMPTY_A;
    ⌇
  ALLOCATE:
    ALLOCATE ITEM IN (A);
    IF I_BASE = NULL  THEN DO;
                      I_BASE = I_PTR;  LINK = I_PTR;
                      END;
                      ELSE DO;
                           LINK ->ITEM.NEXT = I_PTR;
                           LINK = I_PTR;
                           END;
         ITEM.DATA = STUFF;
         ITEM.NEXT = NULL ;
         ⌇
         EMPTY_A:  WRITE FILE (LIST_FILE) FROM (A);
                   A = EMPTY;
                   I_BASE = NULL ;
                   GO TO ALLOCATE;
```

In the EMPTY_A routine above, the statement

A = EMPTY;

causes the extent of A, which measures the number of bytes within A already allocated, to be reset to zero, effectively freeing all the based variables within A. Thus, when the AREA ON condition is recognized, the current contents of A are written onto an external file, and A is emptied so that additional allocations can be performed. Notice that since this chain linkage is via displacements past the beginning of A, input/output operations can be performed without destroying the chain.

The power of the PL/1 list processing facility lies in its flexibility. Because the programmer must define and maintain his own chains, he is free to logically structure his data in the way most appropriate to his particular application. Possible list structures which can be built and maintained using PL/1 are: one-way linked list, multithreaded list, ring, binary tree, etc.

Example 3. In defining a tree structure, it is possible for each node to contain pointers to all its children. Figure 1 below is a schematic of such a representation. If the tree structure has a fixed branching pattern, e.g., a binary tree, then such an approach is reasonable. If, however, the number of daughter nodes associated with any one node is variable, then such a structure becomes unwieldy and difficult to manage. In such cases, each node contains exactly three pointers: to its parent, to its next sibling, and to its first child. Figure 2 below is a schematic of such a logical structure, and the PL/1 procedure CHAIN will create a tree structure of this sort. It is assumed that elements are presented to CHAIN in order from top to bottom, left to right. The placement of the current element is determined by examining its level number. If the level number is 1, then this node is the highest node. Otherwise there are 3 possibilities: this is a sister of the most recently-added node (current level number = previous level number), this node is a daughter of the most recently-added node (current level number > previous level number), or this node is a sister of some previously-added node (current level number < previous level number).

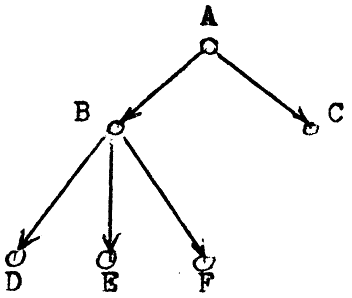


Figure 1

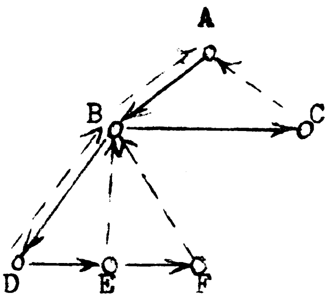


Figure 2

```
CHAIN: PROCEDURE (ELEMENT);
DECLARE 1 ELEMENT,
          2  LEVEL_NUMBER FIXED,
          2  IDENTIFIER CHAR (*),
        1 TREE_NODE BASED (CURRENT),
          2  DAUGHTER POINTER,
          2  SISTER   POINTER,
          2  MOTHER   POINTER,
          2  LEVEL_NUMBER FIXED,
          2  IDENTIFIER CHAR (LENGTH(ELEMENT.IDENTIFIER)),
        LAST POINTER STATIC EXTERNAL,
        TEMP POINTER;

    ALLOCATE TREE_NODE;
    TREE_NODE.LEVEL_NUMBER = ELEMENT.LEVEL_NUMBER;
    TREE_NODE.IDENTIFIER = ELEMENT.IDENTIFIER;
    IF TREE_NODE.LEVEL_NUMBER = 1 THEN DO;
        MOTHER = NULL;
      L1:DAUGHTER = NULL;
        LAST = CURRENT; <——    SISTER = NULL;
        RETURN;
        END;
    IF TREE_NODE.LEVEL_NUMBER = LAST->TREE_NODE.LEVEL_NUMBER THEN DO;
        LAST ->SISTER = CURRENT;
        MOTHER = LAST->MOTHER;
        GO TO L1;
        END;
    IF TREE_NODE.LEVEL_NUMBER > LAST->TREE_NODE.LEVEL_NUMBER THEN DO;
        LAST ->DAUGHTER = CURRENT;
        MOTHER = LAST;
        GO TO L1;
        END;
    DO WHILE (LAST->TREE_NODE.LEVEL_NUMBER > TREE_NODE.LEVEL_NUMBER);
        TEMP = LAST;
        LAST = LAST->MOTHER;
        END;
    IF LAST ->TREE_NODE.LEVEL_NUMBER¬ = TREE_NODE.LEVEL_NUMBER THEN
        LAST = TEMP;
    LAST ->SISTER = CURRENT;
    MOTHER = LAST->MOTHER;
    DAUGHTER = NULL;
    SISTER = NULL;
    LAST = CURRENT;
    END CHAIN;
```

<u>Example 4.</u> It is possible to combine simple list structures in PL/1 and to define and manipulate a compound structure. For example, the structure represented in Figure 3 below consists of a two-way linked list

with rings attached to each node. In this example, this structure represents the logical relationships among corporate data; in the two-way list is stored data about corporate divisions, while each ring corresponds to the departments within a division. The following code assumes that the lists have already been created and that the nodes are in numeric order on division or department code. The procedure RETRIEVE is called to access a specified node and return the indicated field value. An accessing bias is assumed, and it is anticipated that division requests are likely to be clustered; that is, the division currently requested is likely to be close to that previously requested, and thus the current search starts where the previous search ended. RETRIEVE searches up or down the division chain until the desired division is located and then either searches the department ring until the indicated department is found or simply returns the requested division value.

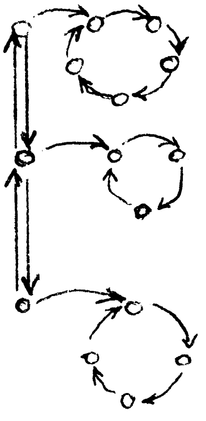


Figure 3

```
RETRIEVE: PROCEDURE (CODE1,CODE2,STATISTIC) RETURNS (VALUE);
DECLARE CODE1 PICTURE (A999),
        CODE2 PICTURE (A999),
        STATISTIC FIXED,
        VALUE FIXED (10,2),
        TANDEM POINTER STATIC EXTERNAI INITIAL (HEAD),
        HEAD POINTER STATIC EXTERNAL,
        1  DIVISION BASED (TANDEM),
           2  UP    POINTER,
           2  DOWN POINTER,
           2  CODE PICTURE (A999),
           2  DEPT_CHAIN POINTER,
           2  VALUE(5) FIXED (10,2),
        1  DEPARTMENT BASED (DEPT_RING),
           2  NEXT POINTER,
           2  CODE PICTURE (A999),
           2  VALUE (5) FIXED (10,2);

L1:  IF CODE1 = DIVISION.CODE THEN GO TO DIV_FOUND;
     IF CODE1 > DIVISION.CODE THEN DO;
          TANDEM = DIVISION.DOWN;
          IF TANDEM = NULL | CODE1 < DIVISION.CODE THEN DO;
           L3: TANDEM = HEAD;
               SIGNAL CONDITION (NOT_FOUND);
               END;
          GO TO L1;
          END;
     TANDEM = DIVISION.UP;
     IF TANDEM = NULL | CODE1 > DIVISION.CODE
          THEN GO TO L3;
          ELSE GO TO L1;
DIVISION_FOUND:
     IF CODE2 = 0 THEN DO;
          VALUE = DIVISION.VALUE(STATISTIC);
          RETURN;
          END;
     DEPT_RING = DEPT_CHAIN;
L4:  IF CODE2 = DEPT_CODE THEN GO TO DEPT_FOUND;
     DEPT_RING = DEPARTMENT.NEXT;
     IF DEPT_RING = DEPT_CHAIN THEN GO TO L3;
     GO TO L4;
DEPT_FOUND:
     VALUE = DEPARTMENT.VALUE(STATISTIC);
     RETURN;
     END;
```

D. CONCLUSION

Many special purpose languages have been designed to solve specific problems, but their design goals have led to rather rigid data organisations and thus to reduced general applicability. PL/1, being a general purpose language, is generally applicable to any type of problem. While it allows the programmer great flexibility in manipulating data, it does impose on him the coding of functions considered as primitives in other languages. This is especially true when its list and string processing facilities are being considered, for only the most basic operations are provided.

REFERENCES

PL/1 Reference Manual, IBM System/360, IBM Form Number C28-8201

PL/1 Programmer's Guide, IBM System/360 Operating System, IBM Form Number C28-6594

Elson, Mark, The PL/1 Language - Analysis and Evaluation, The University of Michigan Engineering Summer Conference advanced topics course in Computer and Program Organization, June 19-30, 1967

Flanigan, Larry K., Notes on the IBM PL/1 Language, University of Michigan, June 1969 (unpub.)

Lawson, Harold W.,"PL/1 List Processing", CACM 10:358, 1967

Wegner, Peter, Programming Languages, Information Structures, and Machine Organization, McGraw Hill, New York, N.Y., 1968

II. LISP

L. K. Flanigan
5/25/72

In using a digital computer for numerically oriented tasks, a programmer usually arranges the data to correspond in some reasonable fashion to the linear layout of the computer memory. The data structure most used is that of the n-dimensional array, where given n subscripts a simple computation produces a single displacement for the actual reference to memory. Most algebraic compilers have such data structures as part of the language they accept, thus making it quite easy to use these structures. At the same time, these data structures are limited in their usefulness, since both the structure (here the number of dimensions) and the size are fixed at the time the structure is created; and this time of creation is often during translation, rather than during execution, of a program. These structures have, nonetheless, served well in numerically oriented tasks.

If, however, one turns to areas such as symbol manipulation, information retrieval, language translation, artificial intelligence, and so on, one quickly finds that the usual data structures are unable to handle the demands of the programs. This occurs both due to the fixed size of the structures and to the nature of the structure. That is, in the above areas we find that the size of the data may vary tremendously, from moment to moment during execution, and it becomes imperative to have variable-size data structures and to be able to create and destroy these data structures at will during execution. Thus, the discipline by which memory is used must be dynamic, at any given time allowing the current program to acquire more memory or give up memory (but be able to retrieve it later if needed).

There is no reason why n-dimensional arrays could not be handled in a dynamic manner, being created upon demand and destroyed when the need for them disappears; this is, in fact, possible in most recent operating systems. However, the fact that arrays have a fixed structure denies these data structures the utility necessary for symbol manipulation and other such applications. In fact, in many non-numeric computer applications, it is the structure, and not the value, of the data which is manipulated. Consider, as an example, the organization of an encyclopedia. Here we have the gross structure:

paragraph = set of sentences
section = set of paragraphs
volume = set of sections
encyclopedia = set of volumes

where we assume the sentence as the basic element. Over all this structure is imposed the usual ordering of information by alphabetic criteria at several levels; a dictionary produces keys for finding specific sections. Note, now, that the problem of information retrieval is one of structure; that is, obtaining information from the encyclopedia (as a data structure) is independent of the specific information (except for keywords used in a dictionary search) and dependent upon the structure of the data. To add new elements to the encyclopedia, it is necessary to insert new data at several levels (volume, section, paragraph, sentence); fixed structures are not convenient for such tasks.

To handle data in which structure, as well as content, is important, programmers have turned to list representations of the data. Lists (and list structures) provide the flexibility needed, often at the cost of storage (for pointers) and time (for list processing). It is interesting to note that to solve the problems of storage handling in multiprogramming and timesharing systems, with their dynamic demands upon memory, these systems are in general using list-type storage organization.

To make the use of list structures readily available, a number of list processors have been produced. These processors tend to fall into one of two groups: those that are part of an algebraic language (often via subroutines), such as FLPL (4) or SLIP (5); and those that are written to directly manipulate lists and list structures of varying types, such a IPL-V (6), CORAL (7), L^6 (8), and LISP 1.5 (9, 10, 11) or LISP 2 (12, 13). We will here consider only LISP 1.5; for comparisons of the various list processors, see references 14 and 15. For a general introduction to list processing independent of particular languages, see Foster (3). Also, the section on list and string processing in Rosen (1) contains several papers of interest.

In the remainder of this paper, "LISP 1.5" will be abbreviated to "LISP".

LISP is a formal mathematical language, designed originally to represent the partial recursive functions defined on a class of symbolic expressions known as S-expressions (11). Programs in LISP are written as S-expressions, and data in LISP is also in the form of S-expressions. LISP has been applied to a large number of non-numeric areas (10).

The most elementary type of S-expression is the atomic symbol or the atom. An atomic symbol is a string of one to thirty capital alphabetic characters (A...Z) and/or numeric characters (0...9), the first of which is alphabetic. Such symbols are atomic in that they are viewed as distinct, separate entities in LISP. (An atomic symbol may itself represent a complex structure; it is atomic because it is viewed as a single element, and its structure is thus not part of the structure of containing elements.) An S-expression is built out of atomic symbols and the three delimiters "(",")", and "." as follows:

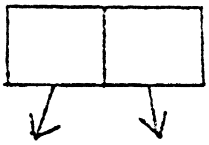
a. An atomic symbol is an S-expression;

b. If s_1 and s_2 are S-expressions, then $(s_1.s_2)$ is an S-expression.

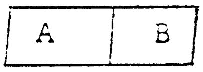
Only expressions constructed from application of these two rules are S-expressions. Some sample S-expressions are:

```
 A
(A.B)
(A.(B.C))
((A.B).C)
((A.B).(C.D))
((X.Y).(X.(Y.(Z.A))))
```

Internally, S-expressions are represented as binary trees (i.e., as tree structures in which each node has two branches). Because of this, there is a graphical notation (9,16) which indicates this type of structure and which often aids in understanding LISP structures. In this notation, we represent each node in the binary tree structure with the symbol



which indicates the left and right branches from the node. For an atomic symbol, we simply write in the symbol, so that (A.B) is represented as



(Such notation should not be confused with the true situation; an atomic symbol is represented internally as a pointer to the definitio of the symbol.) Figure II-1 gives more examples of the graphical structure. Note in the last two examples of Figure II-1 that subexpressions may or may not be repeated in the structure without effect upon the S-expression.

The dot notation as defined above is sufficient to represent all list structures in LISP and is the basic concept of the programming language. For convenience, however, LISP allows a second way to writ S-expressions: that of <u>list notation</u>. A list is written in the form

$(s_1,s_2,s_3,\ldots,s_n)$

or in the form

$(s_1 \square s_2 \square s_3 \square \ldots \square s_n)$ where □ represents one or more blanks

where s_i is an S-expression. Such a list is interpreted as the S-expression

$(s_1.(s_2.(s_3.(\ldots(s_n.NIL)\ldots))))$

where NIL is a special atomic symbol recognized as the terminator of a list. The empty list () and NIL represent the same element. Thus we have:

(A B) ≡ (A.(B.NIL))
(A B C) ≡ (A.(B.(C.NIL)))
(A) ≡ (A.NIL)

A list may have sublists:

(A (B C)) ≡ (A.((B.(C.NIL)).NIL))
((A B) C) ≡ ((A.(B.NIL)).NIL)
((A) (B)) ≡ ((A.NIL).((B.NIL).NIL))
((A) B) ≡ ((A.NIL).(B.NIL))

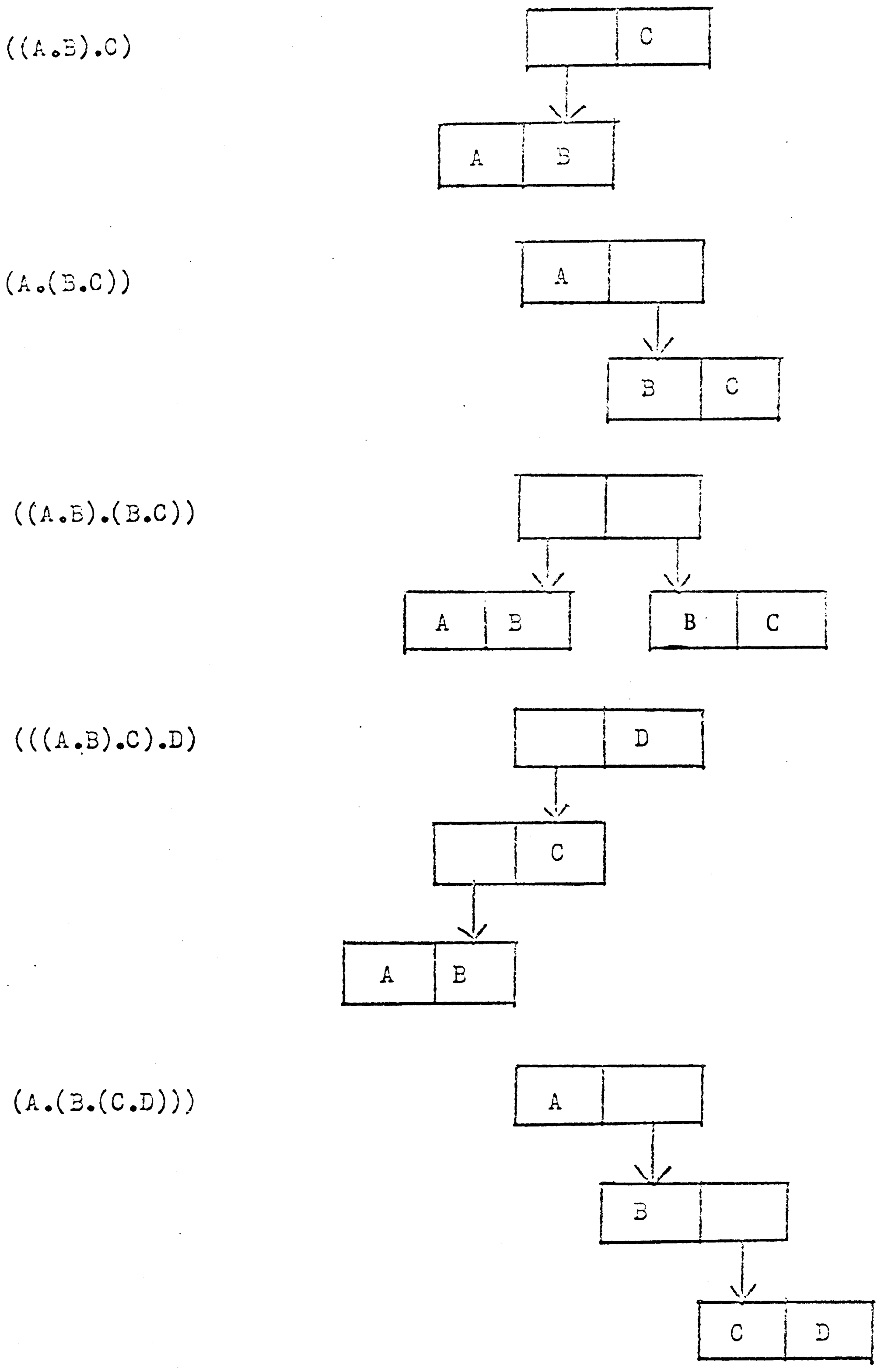


Figure II-1: Part 1 of 2.

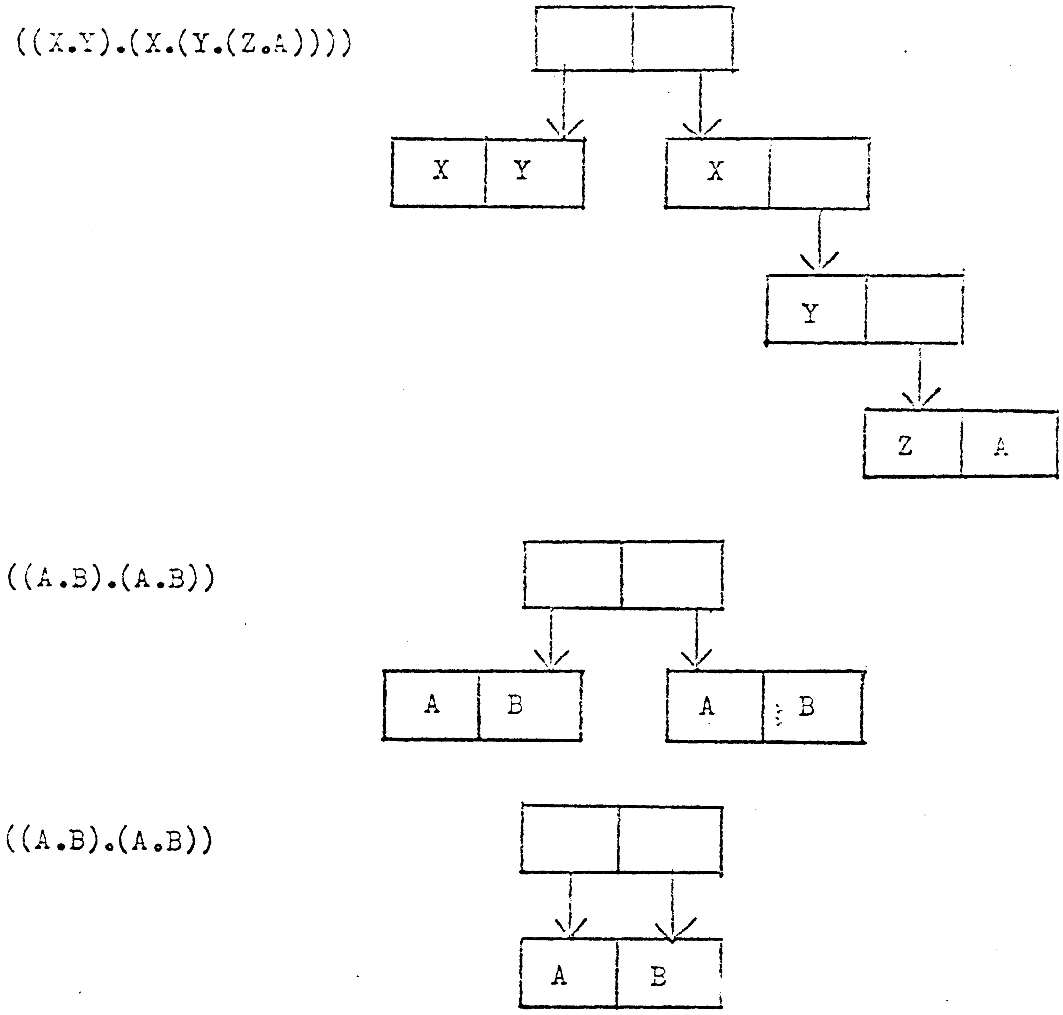


Figure II-1: Part 2 of 2.

Finally, list notation and dot notation may be combined in writing S-expressions:

```
    (A (B.C )) ≡(A.((B.C).NIL))
    (A (B.C).(D  ≡ (A.((B.C)  .NIL)))
```

Note that any list structure or S-expression may be written in the dot notation, and thus any S-expression may be converted to the pure dot notation. Not all S-expressions may be written in list notation, however. In particular, a list structure can be written in list notation only if each sublist, and the list itself, terminates with a NIL. Thus, (A.B) cannot be written in list notation. Figure II-2 shows the graphics for some list and mixed notation S-expressions.

As stated earlier, LISP is a language in which functions of S-expressions may be written; we now introduce some elementary functions of S-expressions. Function names will be written with lower case letters; their arguments will be placed in square brackets separated by semicolons. S-expressions will be written with capital letters as previously defined.

The first elementary function is one which constructs S-expressions and is hence named cons. It has two arguments from which it builds an S-expression:

```
    cons[X;Y] = (X.Y)
    cons[(A.B);C] = ((A.B).C)
```

Note that the arguments and the value of cons are S-expressions; hence, composition is possible:

```
    cons[cons[A;B];C] = ((A.B).C)
    cons[(A);cons[B;NIL]] = ((A).(B.NIL))
                          = ((A).(B))
```

Two additional primitive functions, car and cdr, are defined to produce subexpressions of an S-expression; they each have a single argument, and each is undefined if the argument is atomic. They are defined by:

```
    car[(A.B)] = A
    cdr[(A.B)] = B
```

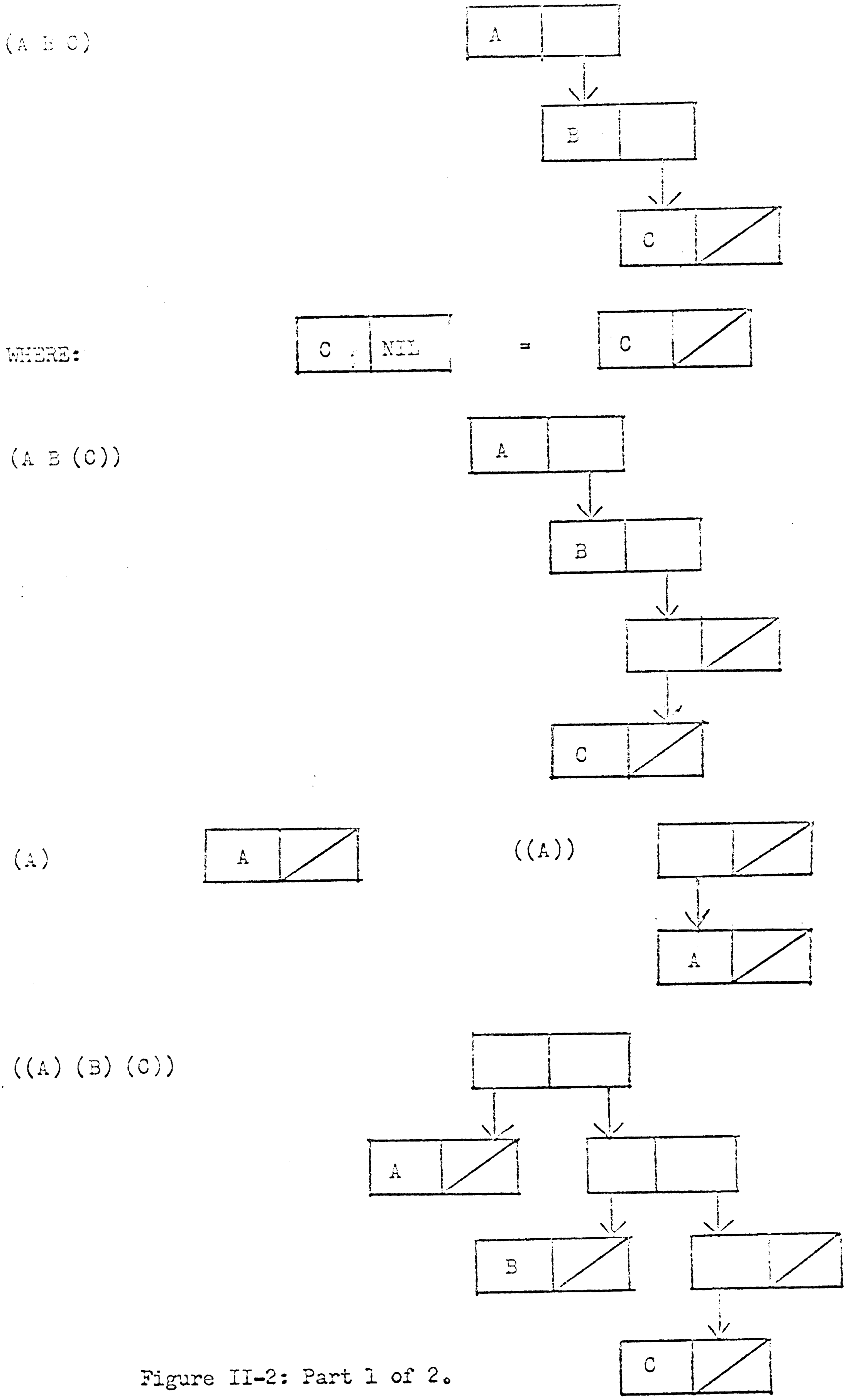


Figure II-2: Part 1 of 2.

(A.((B) C))

((A B).(C D))

(((A).(B)).((C).(D)))

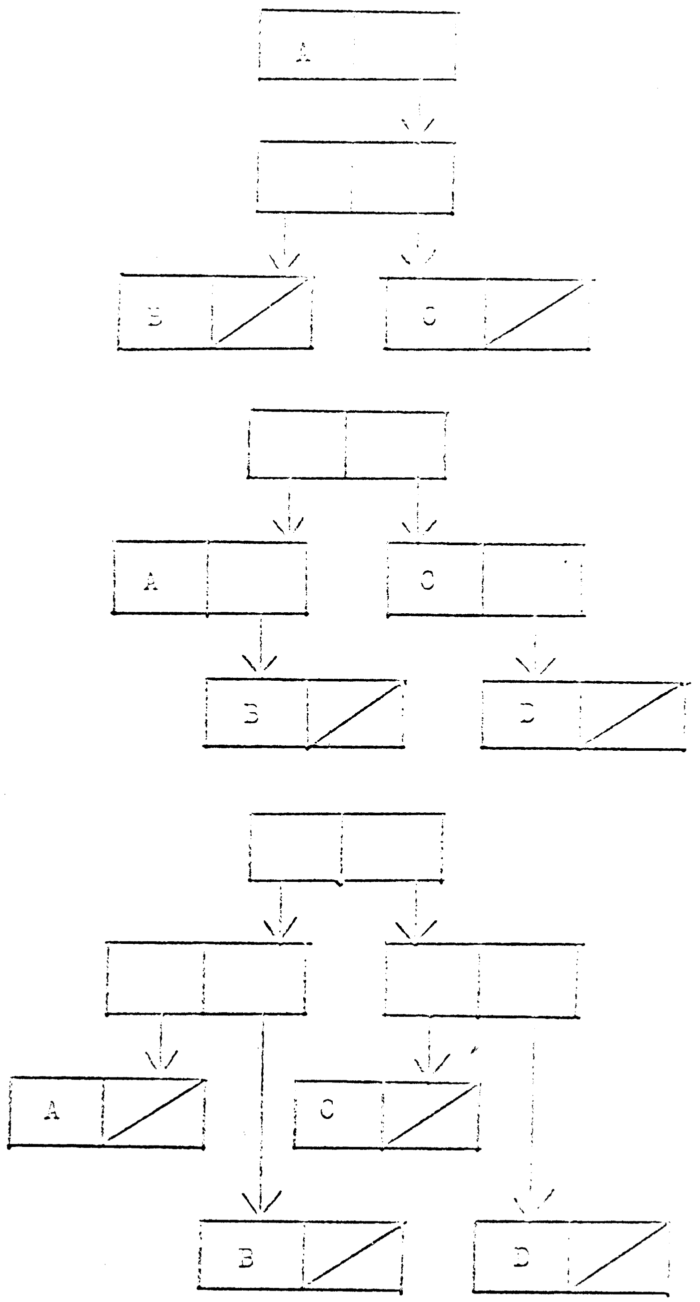


Figure II-2: Part 2 of 2.

Again, composition is available:

```
    car[cdr[(A.(B1.B2))]] = B1
```

Note carefully the distinctions between list notation and dot notation:

```
    car[(A.B)] = A
    cdr[(A.B)] = B
    car[(A B)] = A
    cdr[(A B)] = (B)
```

From the definitions, we have the following identities:

```
    car[cons[x;y]] = x
    cdr[cons[x;y]] = y
    cons[car[z];cdr[z]] = z
```

where x,y, and z are any S-expressions and z is composite (non-atomic)
For convenience, multiple car's and cdr's are often abbreviated:

```
    caddr[(A B C)] = C
    cadadr[(A (B C) D)] = C
```

(See Figure II-3 for graphic interpretations.)

A *predicate* is a function whose value is either true or false. In LISP, two special atoms, T and F, are used to represent true and false respectively. Two predicates of interest are eq and atom. The predicate *eq* has two arguments, which must be atomic, and has value T if they are equal:

```
    eq[X;Y] = F
    eq[B;B] = T
    eq[X;(Y.Z)] = undefined
```

The predicate *atom* has a single argument and has value T if that argument is an atom:

```
    atom[(X Y)] = F
    atom[(Z)] = F
    atom[Z] = T
    atom[NIL] = T
```

Using the above functions, it is possible to define new functions

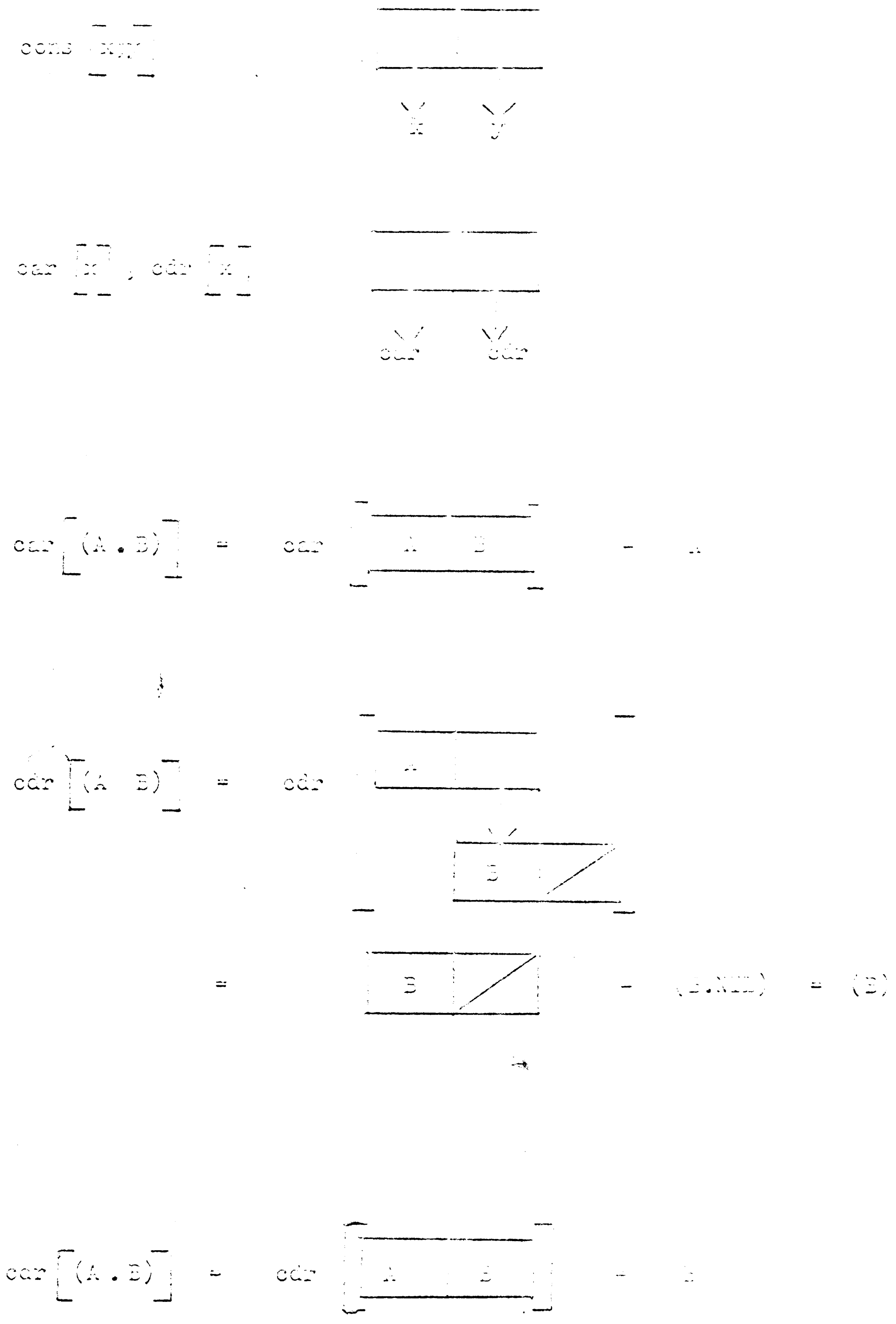

cons [x,y]
x
y
car [x] , cdr [x]
car
cdr
car [(A . B)] = car [A B] = A
cdr [(A . B)] = cdr [A]
B
= B = (B.NIL) = (B)
car [(A . B)] = cdr [A B] = B

Figure II-3.

through composition. The class of functions so obtained is not particularly interesting, however. To obtain a larger class, the concept of <u>conditional expression</u> is introduced. A conditional expression has the form

$$[p_1 \rightarrow e_1; p_2 \rightarrow e_2; \dots ; p_n \rightarrow e_n], n \geq 1$$

where p_i is an expression whose value is T or F and e_i is any expression. The interpretation of this conditional expression is: if p_k is the first p_i, $i=1,2,\dots,k \leq n$, to have value T, then the value of e_k is taken as the value of the conditional expression. If no p_i is true, the conditional expression is undefined. Note that at most one e_i is evaluated. Each p_i or e_i may itself be an S-expression, a function, a composition of functions, or another conditional expression. Thus, the function eql, defined as

eql[x;y] = [atom[x]∧atom[y]→eq[x;y];T→COMP] is the same as eq except that its value is COMP if x or y is composite. A more interesting use of conditional expressions is in defining recursive functions. Consider:

ff[x] = [atom[x]→x; T→ff[car[x]]]

This function selects the first atomic symbol from its argument:

ff[A] = A
ff[((A.B).C)]= A

Additional examples of function definitions using conditional expressions are:

|x| = [x<0→-x; T→x]
n! = [n≤0→1; T→n*[n-1]!]
gcd[x;y] = [x>y→gcd[y;x];rem[y;x]=0→
        x; T→gcd[rem[y;x];x]]
where rem[m;n] = remainder after
        dividing m by n.

The last function (gcd) is the Euclidean algorithm for finding the greatest common divisor of two positive integers x and y.

Since LISP is based on the evaluation of functions whose arguments and values are S-expressions, it becomes necessary to develop a notation for function definition and function evaluation. This is done by using the lambda notation of Church (28). In this notation, an expression like y + 5x is called a form. To convert a form to a function, one must specify the dummy arguments and the form. This is done via the notation

lambda[[x_1;x_2;...;x_n];f]

where the x_i are arguments and f is a form. Thus,

lambda[[x;y];y+5x]

is the name of a function of two arguments whose value is found by substituting the values of the arguments into the form y+5x. Note that we may now add the values of the arguments to the above notation to get a function value:

lambda[[x;y];y+5x][3;10] = 25

That is,

lambda[[x_1;...;x_n];f]

is the name of a function, while

lambda[[x_1;...;x_n];f] [y_1;...;y_n]

is the value of the function for the argument values [y_1;...;y_n]. Therefore, the definition of absolute value now becomes

lambda[[x];[x<0→-x;T→x]]

and this may be applied to arguments.

To use this notation with recursive function definitions requires further notation, since we must be able within the definition to indicate (recursively) that the whole definition is to be applied. To solve this problem, the *label* notation is introduced:

label [a;e]

states that the expression e has the name a. We may now write the previous function definitions as follows (this is now LISP-like notation):

```
label[ff;lambda[[x];[atom[x]→x;T→ff[car[x]]]]]
label[abs;lambda[[x];[x<0→-x;T→x]]]
label[fact;lambda[[n];[n≤0→1;T→n*fact[n-1]]]]
label[gcd;lambda[[x;y];[x>y→gcd[y;x];
     rem[y;x]=0→x;T→gcd[rem[y;x];x]]]]
```

Obviously, one of the problems in writing LISP programs is proper bracketing! One may write function definitions using either the lambda or the label format; the label format is needed only if the function is to be given a name. This naming is necessary in recursive function definitions since one must be able to reference the definition from within the definition.

Now let us consider some functions which operate on S-expressions:

a. subst [x;y;z]

This function gives the result of substituting the S-expressic x for all occurrences of the atomic symbol y in the S-expression z: the definition is

```
subst = lambda[[x;y;z];[atom[z]→
  [eq[z;y]→x;T→z];T→
  cons[subst[x;y;car[z]];
  subst[x;y;cdr[z]]]]]
```

b. equal [x;y]

This is a predicate whose value is T if x and y are the same S-expression; its definition:

```
equal = lambda[[x;y];[atom[x]→
  [atom[y]→eq[x;y];T→F];
  equal[car[x];car[y]]→equal
  [cdr[x];cdr[y]];T→F]]
```

c. null [x]

This is a predicate whose value is T if x is the S-expression NIL; its definition:

```
null = lambda[[x];[atom[x]→
  eq[x;NIL];T→F]]
```

d. maplist [x;f]

This is a function with arguments x and f, where x is an S-expression and f is a function from S-expressions to S-expressions; maplist applies the function f to each subexpression of the S-expression x, producing an S-expression whose elements are the corresponding results; its definition:

```
    maplist = lambda[[x;f];[null[x]→
      NIL;T→cons[f[x];maplist[cdr[x];f]]]]
```

Now let us consider a more complicated function, one which computes the partial derivative of an expression written in a Polish-prefix-like notation. The S-expressions to be differentiated are written according to the rules:

1. an atomic symbol is an allowed expression;
2. If $e_1, e_2, \ldots, e_n$ are allowed expressions, then so are
   (PLUS e_1 e_2 ... e_n)
   (TIMES e_1 e_2 ... e_n)

Thus, the mathematical expression

X(X+A)(X+B)

would be written

(TIMES X (PLUS X A) (PLUS X B))

Our rules of differentiation are the standard rules for partial differentiation:

$$\frac{dx}{dx} = 1$$

$$\frac{dy}{dx} = 0 \quad (y \neq x)$$

$$\frac{d(u+v)}{dx} = \frac{du}{dx} + \frac{dv}{dx}$$

$$\frac{d(u*v)}{dx} = v\frac{du}{dx} + u\frac{dv}{dx}$$

The definition of the differentiation function is:

```
    diff = lambda[[y;x];[atom[y]→
      [eq[y;x]→ONE; T→ZERO];
      eq[car[y];PLUS]→cons[PLUS;
      maplist[cdr[y];lambda[[z];
      diff[car[z];x]]]];eq[car[y];
      TIMES]→cons[PLUS;maplist
      [cdr[y];lambda[[z];cons
      [TIMES;maplist[cdr[y];lambda
      [w];¬eq[z;w]→car[w];T→
      diff [car [ w ];x]]]]]]]]
```

If this function is applied to

```
    (TIMES X (PLUS X A) (PLUS X B))
```

the resulting S-expression is

```
    (PLUS (TIMES (PLUS X A) (PLUS X B))
      (TIMES X (PLUS X B))  (TIMES
      X (PLUS X A)))
```

It is possible to write a universal LISP function evalquote [fn;x] which for any two S-expressions fn and x, where fn must have a value which is a function name, produces the result of applying the function fn to the arguments x. That is, given a function definition fn and a set of arguments x, evalquote is able to produce the result of fn[x]; hence evalquote is a universal LISP function. Note, howeve that we have not been writing functions as S-expressions so far; in fact, we have used what Mc Carthy calls M-expressions to write functions. M-expressions use small alphabetic characters, square brackets, and semicolons. Therefore, to use our universal function evalquote, it is necessary to rewrite our M-expressions as S-expressi The following transformation rules effect this conversion:

1. If ε is an S-expression, it is replaced by (QUOTE ε)
2. Lower case variable and function names are written in upper case; thus, car becomes CAR
3. $f[e_1;...;e_n]$ becomes $(f'\ e'_1\ ...\ e'_n)$, where prime denotes transformation; thus, cons[car[x];cdr[x]] becomes (CONS (CAR X) (CDR X))
4. $[p_1 \rightarrow e_1;...;p_n \rightarrow e_n]$ becomes $(\text{COND}\ (p'_1\ e'_1)\ ...\ (p'_n\ e'_n))$
5. $\text{lambda}[[x_1;...;x_n];\varepsilon]$ becomes $(\text{LAMBDA}\ (x'_1\ ...\ x'_n)\ \varepsilon')$
6. label[a;ε] becomes (LABEL a' ε')

Applying these rules, T becomes (QUOTE T), ff[car[x]] becomes (FF (CAR X)), and [atom[x]→x;T→ff[car[x]]] becomes (COND ((ATOM X) X) ((QUOTE T) (FF (CAR X)))). Further, our function ff now becomes (LABEL FF (LAMBDA (X) (COND ((ATOM X) X) ((QUOTE T) (FF (CAR X)))))), and our function equal becomes (LABEL EQUAL (LAMBDA (X Y) (COND ((ATOM X) (COND ((ATOM Y) (EQ X Y)) ((QUOTE T) (QUOTE F)))) ((EQUAL (CAR X) (CAR Y)) (EQUAL (CDR X) (CDR Y))) (QUOTE T) (QUOTE F))

Thus, we have that the functions which we have been writing in M-expressions can be written as S-expressions. Note that this means that LISP is a language in which data and program are written in the same languages: S-expressions. Further, the LISP interpreter itself is nothing more than the function evalquote, as far as the user is concerned. Thus, knowing this function definition tells the user all he need know about the LISP language. Due to its complexity, the definition of evalquote is not given here. (See McCarthy, et. al., (9).)

The LISP system has a rather large set of builtin (predefined) functions and predicates for the user. Both integer and floating point arithmetic and constants are available in the system. In pure LISP, a function definition, via label or lambda, is presented, together with arguments, and the function is evaluated for those arguments according to evalquote. The function definitions are then lost; that is, function definition via label and lambda is one-shot affair. Since this is obviously an inconvenient mode of operation, the programmer may use a define function to permanently define (for a run) a set of functions, thereafter calling these functions by name as he does for cons, car, cdr, et. al. Also, a program mode is available through which a limited amount of control transfer among LISP statements may be excercised during function evaluation. This is similar to procedures in ALGOL and PL/I. Thus, for example, a function LENGTH, which counts the number of elements in the top level of a list, may be written as:

```
    DEFINE (((LENGTH (LAMBDA (X)
    (PROG (U V)
          (SETQ  V  0)
          (SETQ  U  X)
   A      (COND ((NULL U) (RETURN V)))
          (SETQ  U  (CDR U))
          (SETQ  V  (ADD1 V))
          (GO A))))))
```

In words, this may be read as:

length is function of one arg x

```
          it is a program with two
             program variables u and v
      store 0 in v
      store x in u
A     if u contains NIL, program is finished,
          value is content of v
      store cdr u into u
      store v+1 in v
      loop back to A and continue
```

The programmer may mix defines and progs to his heart's content to produce LISP programs! In the absence of the prog feature, the above function would have to be recursively defined:

```
    DEFINE (((LENGTH (LAMBDA (M)
    (COND ((NULL M) 0) (T (ADD1
    (LENGTH (CDR M)))))))))
```

Once defined, the length function may be applied as if it were a builtin function:

```
    (LENGTH (A B C)) = 3
    (LENGTH ((A B) (C D)) = 2
```

The representation of an atom in LISP is actually a list structure. The pointer to an atom points to this structure. An atom has an association list on which is kept its current definition and other information (including the fact that it is an atom and the BCD print name of the atom). There are LISP functions which allow the user to manipulate the association list of an atom, so control may be excercised over the current contents of this association list for any atom.

Storage in LISP is quite dynamic. LISP places a large area of memory into free storage, and this area is formed into a free storage list. When a cons is executed, an element is taken from the free storage list and used to create the new element. If there is no element remaining on the free storage list, the the LISP interpreter sweeps through its list storage (plus some other areas), finds those cells which are currently not being used, and returns these cells to the free storage list; this operation is known as garbage collection. It is done automatically, so the user need never be aware of it or concerned with it. Here again the use of list structures provides a facility vital to the interpreter.

V. BIBLIOGRAPHY

1. Rosen, Saul (editor): Programming Systems and Languages, McGraw-Hill Book Co., New York, 1967

2. Notes for the advanced topics course in Computer and Program Organization, The University of Michigan Engineering Summer Conferences, June 19-30, 1967.

3. Foster, J. M.: List Processing, Macdonald & Co., London, 1967.

4. Gelernter, H.: A FORTRAN Compiled List Processing Language, JACM, 7:87, 1960.

5. Weizenbaum, J.: Symmetric List Processor, CACM, 6:524, 1963.

6. Newell, A. (editor): Information Processing Language -V Manual, Prentice Hall, Englewood Cliffs, N. J., 1963.

7. Sutherland, W. R.: The Coral Language and Data Structures, in Tech. Report 405, MIT Lincoln Laboratory, Lexington, Mass., 1966.

8. Knowlton, K. C.: A Programmer's Description of L^6, CACM, 9:No. 8, 1966.

9. McCarthy, John, et. al.: LISP 1.5 Programmer's Manual, The MIT Press, Cambridge, Mass., 1965.

10. Bobrow, D. G., and E. C. Berkeley, (editors): The Programming Language LISP: Its Operation and Its Applications, The MIT Press, Cambridge, Mass., 1964.

11. McCarthy, J.: Recursive Functions of Symbolic Expressions and Their Computation by Machine, CACM, 3:184, 1960. (Reprinted in (1)).

12. Abrahams, P. W. et. al.: The LISP 2 Programming Language and System, Proc. AFIPS FJCC, 29:661, 1966.

13. A set of technical working papers on various aspects of the LISP 2 system; reprinted in (2).

14. Raphael, B. and D. G. Bobrow: A Comparison of List Processing Computer Languages, CACM, 7:231, 1964. (Reprinted in (1) with an expanded bibliography).

15. Raphael, B.: Survey of Computer Languages for Symbolic and Algebraic Manipulations, Final Report, Stanford Research Institute Project 6084, March, 1967 (Reprinted in (2)).

16. Weissman, Clark: LISP 1.5 Primer, Dickenson Publishing Co., Inc., Belmont, Cal., 1967.

17. Strachey, C.: A General Purpose Macrogenerator, Computer J., Oct., 1965.

18. Mooers, C. N.: TRAC -A Procedure Describing Language for the Reactive Typewriter, CACM 9, No. 3, March, 1966.

19. Waite, W.: A Language-Independent Macro Processor, CACM, July, 1967.

20. COMIT Programmers Reference Manual, MIT Press, Cambridge, Mass., 1961.

21. Introduction to COMIT Programming, MIT Press, Cambridge, Mass., 1961.

22. Farber, D. J., et. al.: SNOBOL, a String-Manipulation Language, JACM 11, No. 1, Jan., 1964.

For those familiar with LISP, it may be pointed out that CAR and CDR of LISP can be written:

```
     CAR:  PROCEDURE (P) RETURNS(POINTER)
        DECLARE 1 ELEM BASED (P),
          (2 LEFT, 2 RIGHT) POINTER;
        RETURN (LEFT);
     CDR:  ENTRY (P) RETURNS(POINTER)
        RETURN (RIGHT); END CAR;
```

and the LISP CONS function may be written as

```
     CONS:  PROCEDURE (P,Q) RETURNS(POINTER)
        DECLARE 1 ELEM BASED (PT),
          (2 L, 2 R, P, Q) POINTER;
        ALLOCATE ELEM; L = P; R = Q;
        RETURN (PT); END;
```

Actually, to be precise, a LISP node should be defined as

```
   DECLARE 1 ELEM BASED (P),
           2 LEFT POINTER,
           2 RIGHT POINTER,
           2 IND BIT (6);
```

where IND is an indicator which identifies the type of information pointed to by the LEFT and RIGHT pointers in the node.

car, cdr, cons as written in PL/I

Assumes that cells always represented by pointers, which are passed around to represent cells.

SIMSCRIPT II

CCS 500
Professor Riddle

by John Rinn

INTRODUCTION

First I shall list items not discussed in the paper:

1. I have not gone into the simulation capabilities of the language at all.
2. Noise words may be left out, or certain alternatives used. These possibilities have not been discussed, except as they appear in the examples.
3. Many functional words have alternates (symbols or words). Again these have not been discussed except for examples.
4. Many other capabilities of the language have not been discussed due to a lack of "time and space". Discussion usually includes the more simple aspects of the language.

Complaints include:

1. The language is very incomplete on the 360. Copy *SIM2NEWS for details.
2. It is fairly difficult to get a non-trivial (and in many cases a trivial) program to run.
3. Simscript is very expensive. Example V cost over $10.00 to compile and run!!!
4. Error messages are very poor. In the example on the following page, the error is in line 27. the "ONE" should be a "1".

```
27          PRINT ONE LINE AS FOLLOWS

28             READ C IN SCIENTIFIC NOTATION, I.E. AS E(9,3)
29          READ C AS /, E(9,3)
30          WRITE C AS B 12, E(9,3)
31          PRINT 1 LINE AS FOLLOWS

32             TYPE 0 TO STOP, OR 1 TO CONTINUE
33          READ E
34          IF E EQUALS 0, GO TO STOP
35          OTHERWISE, GO TO START

36      'STOP' STOP
37          END
```

```
**** ERROR OF TYPE 1 INVOLVING 'PRINT' AT STATEMENT 12.

**** ERROR OF TYPE 1 INVOLVING 'ONE' AT STATEMENT 12.

**** ERROR OF TYPE 1 INVOLVING 'LINE' AT STATEMENT 12.
**** ERROR OF TYPE 1 INVOLVING 'AS' AT STATEMENT 12.
**** ERROR OF TYPE 1 INVOLVING 'FOLLOWS' AT STATEMENT 12.

**** ERROR OF TYPE 1 INVOLVING 'IN' AT STATEMENT 13.

**** ERROR OF TYPE 1 INVOLVING 'SCIENTIFIC' AT STATEMENT 13.
**** ERROR OF TYPE 1 INVOLVING 'NOTATION' AT STATEMENT 13.
**** ERROR OF TYPE 1 INVOLVING ',' AT STATEMENT 13.
**** ERROR OF TYPE 1 INVOLVING 'I.E' AT STATEMENT 13.

**** ERROR OF TYPE 1 INVOLVING 'AS' AT STATEMENT 13.
**** ERROR OF TYPE 1 INVOLVING 'E' AT STATEMENT 13.
**** ERROR OF TYPE 1 INVOLVING '(' AT STATEMENT 13.
**** ERROR OF TYPE 1 INVOLVING '9' AT STATEMENT 13.

**** ERROR OF TYPE 1 INVOLVING ',' AT STATEMENT 13.
**** ERROR OF TYPE 1 INVOLVING '3' AT STATEMENT 13.
**** ERROR OF TYPE 1 INVOLVING ')' AT STATEMENT 13.
```

SECTION I

In this section we shall discuss the aspects of SIMSCRIPT II which are prerequisite to the utilization of its set abilities.

Input/Output

The program in EXAMPLE 1 gives many examples of different input and output formatting, and the two pages which follow it consist of two runs of the program with sample data.

Notice that spacing between data items is not critical with the simple read statement. One blank is all that is necessary, but more may be used, if desired. The only time that I ran into trouble with the simple statements was in the run at the bottom of the third page of this example, where I read ASDFG into A. I did not, however, have time to trace this down before the writting of this paper.

In the first run, 12 was read into the left two digits of a five digit integer using a B 8, I 5 format. The result was the storing of 12000. It should also be pointed out that unless a new record is specified ("/"), a B n format item in a read statement will cause the next piece of data to be read beginning at column n of the same record from which the last read statement read.

I encountered much trouble with exponential (scientific) notation on input (possibly on output, also). Again I lacked time to trace this down to determine if it was my problem, or the compiler's.

EXAMPLE I

```
>      1          PREAMPLE
>      2           DEFINE A AS AN ALPHA VARIABLE
>      3           NORMALLY, MODE IS INTEGER
>      4           DEFINE B AND E AS VARIABLES

>      5           DEFINE C AND F AS REAL VARIABLES
>      6           DEFINE D AS A 1-DIMENSIONAL ARRAY
>      7          END
>      8           PRINT 1 LINE THUS
>      9             B IS I2 AND THE SUBSCRIPT SIZE OF ARRAY D
>     10           READ B
>     11           RESERVE D(*) AS B
>     12       'START' PRINT 1 LINE WITH B AS FOLLOWS
>     13             A IS A4, C IS D(7,2), AND D IS OF LENGTH ** AND A3
>     14           READ A,C AND D
>     15           SKIP 3 OUTPUT LINES
>     16           PRINT 1 LINE WITH A,B,C AS FOLLOWS
>     17             **** IS ALPHA, B = **, AND C = ***.**
>     18           SKIP 2 OUTPUT LINES
>     19           FOR I = 1 TO B, PRINT 1 LINE WITH I AND D(I) THUS
>     20             THE ** VALUE OF D IS *** .
>     21           SKIP 4 OUTPUT LINES
>     22           PRINT 4 LINES WITH B AS FOLLOWS
>     23             A IS 5 LETTERS BEGINNING IS CLOUMN 1
>     24             E IS I5 BEGINNING IN CLOUMN 8
>     25             C IS D(7,2) BEGINNING IN COLUMN 16
>     26             D IS A 1 BY ** ARRAY BEGINNING IN COLUMN 27 AS I3 AND
*EVERY 4TH COLUMN
>     27           READ A,E AND C AS /, B 1, A 5, B 8, I 5, B 16, D(7,2), B
* 26
>     28           FOR I = 1 TO B, READ D(I) AS S 1, I 3
>     29           WRITE A,E AND C AS *, B 25, A 5, B 35, I 5, B 45, D(7,2)
*, /, /
>     30           FOR I = 1 TO B, WRITE D(I) AS I 3, S 3
>     31           SKIP 3 OUTPUT LINES
>     32           PRINT 1 LINE AS FOLLOWS
>     33           READ F IN SCIENTIFIC NOTATION, I.E. AS E(12,3)
>     34        READ F AS /, E(12,3)
>     35           WRITE F AS B 12, E(12,3)

>     36           PRINT 1 LINE AS FOLLOWS
>     37             TYPE 0 TO STOP, OR 1 TO CONTINUE
>     38           READ E
>     39           IF E EQUALS 0, GO TO STOP
>     40           ELSE GO TO START
>     41       'STOP' STOP

>     42           END
```

```
      B IS I2 AND THE SUBSCRIPT SIZE OF ARRAY D
4
      A IS A4, C IS D(7,2), AND D IS OF LENGTH  4 AND A3
DFGH 32.1 45  67   78 345



      DFGH IS ALPHA, B =  4, AND C =  32.10



      THE  1 VALUE OF D IS  45 .

      THE  2 VALUE OF D IS  67 .
      THE  3 VALUE OF D IS  78 .
      THE  4 VALUE OF D IS 345 .




      A IS 5 LETTERS BEGINNING IS CLOUMN 1
      E IS I5 BEGINNING IN CLOUMN 8

      C IS D(7,2) BEGINNING IN COLUMN 16

      D IS A 1 BY  4
  ARRAY BEGINNING IN COLUMN 27 AS I3 AND EVERY 4TH COLUMN
ABCDE  12       34.8        456 234  12   2






                          ABCD       12000        34.80



 456   234    12     2



      READ C IN SCIENTIFIC NOTATION, I.E. AS E(12,3)
1.000E 00
                  1.000E 00     TYPE 0 TO STOP, OR 1 TO CONTINUE
```

```
       A IS A4, C IS D(7,2), AND D IS OF LENGTH  4 AND A3
 KKL    56.1 6   45    234   1




       KKL  IS ALPHA, B =  4, AND C =  56.10



       THE  1 VALUE OF D IS   6 .
       THE  2 VALUE OF D IS  45 .
       THE  3 VALUE OF D IS 234 .
       THE  4 VALUE OF D IS   1 .




       A IS 5 LETTERS BEGINNING IS CLOUMN 1

       E IS I5 BEGINNING IN CLOUMN 8
       C IS D(7,2) BEGINNING IN COLUMN 16
       D IS A 1 BY  4
  ARRAY BEGINNING IN COLUMN 27 AS I3 AND EVERY 4TH COLUMN
ABCDEFG1234567845.33333333123456789012345678 9



                             ABCD       12345       45.33



 123    567    901    345



       READ C IN SCIENTIFIC NOTATION, I.E. AS E(12,3)
1.456E 05
 AT LOCATION  50094A
********** ERROR NUMBER 124 **********
REAL NUMBER TOO LARGE FOR INPUT

ERROR RETURN
```

```
        B IS I2 AND THE SUBSCRIPT SIZE OF ARRAY D
  4

        A IS A4, C IS D(7,2), AND D IS OF LENGTH  4 AND A3

  ASDFG  2.4 2 3 4 5
   AT LOCATION  504C3C
   CALLED FROM  500338

  ********** ERROR NUMBER 128 **********

  INVALID CHARACTER IN "D" OR "E" FORMAT DURING INPUT
 #ERROR RETURN
 #
```

The Preamble

The Preamble of a SIMSCRIPT II program written at this level is used to declare mode (the normal mode is real if not explicitly changed), to declare that a particular name is associated with an array, and to declare the dimension of each array. The dimension of each array must be declared in the preamble, but the subscript size is declared in an executable RESERVE statement when storage is actually allocated. The preamble must end with an END statement. The sample programs is this paper contain several samples of preambles.

General Information and Examples

Table I describes the fact that arrays are stored with a set of pointers for each dimension. This design coupled with the fact that a RELEASE statement must be executed to return arrays to free storage, while resetting an array pointer without a release statement will not return the array to free storage, allows dynamic construction of a "ragged table". The first program of EXAMPLE II will construct such a table in the form of a tree, while the second program will search the tree for a given entry.

Tables II and III list functions available to the SIMSCRIPT II user. Functions of Table II use a calling sequence such as:

```
        LET SQ.N=SQRT.F(N)
   or   ADD SQRT.F(N² + M²) TO SUM
```

while functions of Table III are called with a COMPUTE statement. For example:

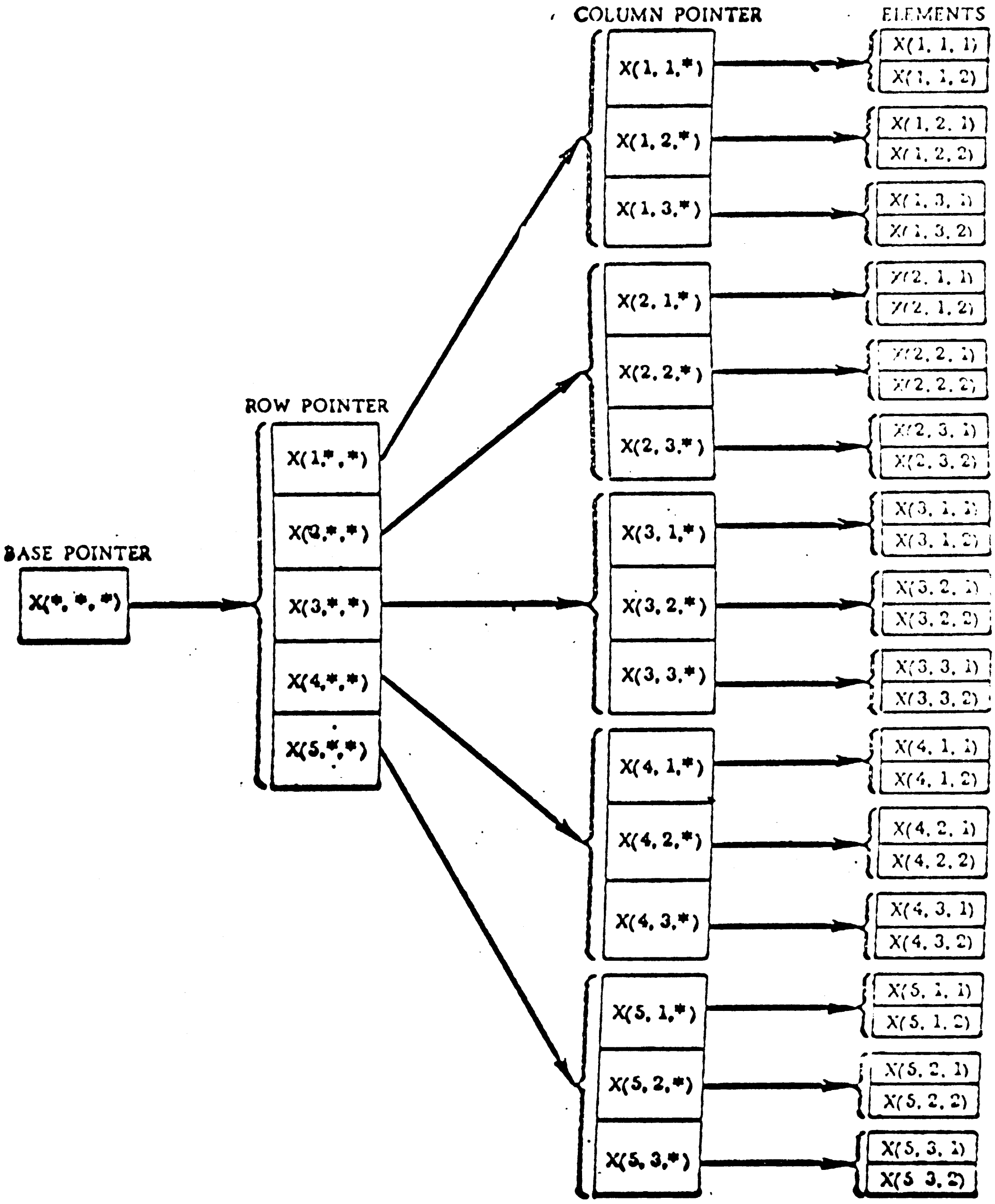
```
          FOR I = 1 TO 100 WITH X(I) LESS THAN N
          COMPUTE NX = THE NUMBER, SUMX AS THE SUM,
          MEANX AS THE MEAN OF X(I)
```

After execution of these statements, NX will contain the number of X(I) which were less than N, SUMX will be the sum ot these X(I), and MEANX will be their average.

EXAMPLE III is fairly straight forward. Comments are enclosed in double quote marks ("..."), while statement labels are between single quotes ('...'). Labels are not, however, enclosed when referred to within a statement.

In EXAMPLE III, I suspect that information should be read into the 4-dimensional array called CAPACITY in the statement noted "READ INITIAL DATA." This information would consist of the initial number of seats available for each route, with respect to the class of travel and the type of carrier (e.g. train, airplane, bus).

Table I[1]


BASE POINTER
X(*,*,*)
ROW POINTER
X(1,*,*)
X(2,*,*)
X(3,*,*)
X(4,*,*)
X(5,*,*)
COLUMN POINTER
X(1,1,*)
X(1,2,*)
X(1,3,*)
X(2,1,*)
X(2,2,*)
X(2,3,*)
X(3,1,*)
X(3,2,*)
X(3,3,*)
X(4,1,*)
X(4,2,*)
X(4,3,*)
X(5,1,*)
X(5,2,*)
X(5,3,*)
ELEMENTS
X(1,1,1)
X(1,1,2)
X(1,2,1)
X(1,2,2)
X(1,3,1)
X(1,3,2)
X(2,1,1)
X(2,1,2)
X(2,2,1)
X(2,2,2)
X(2,3,1)
X(2,3,2)
X(3,1,1)
X(3,1,2)
X(3,2,1)
X(3,2,2)
X(3,3,1)
X(3,3,2)
X(4,1,1)
X(4,1,2)
X(4,2,1)
X(4,2,2)
X(4,3,1)
X(4,3,2)
X(5,1,1)
X(5,1,2)
X(5,2,1)
X(5,2,2)
X(5,3,1)
X(5,3,2)

EXAMPLE II[2]

```
           PREAMBLE NORMALLY, MODE IS INTEGER
           DEFINE LEVEL AND TREE AS 1-DIMENSIONAL ARRAYS
           END

           READ N   RESERVE LEVEL(*) AS N
           FOR I=1 TO N,
           DO
               RESERVE TREE(*) AS 2**(I-1)  READ TREE
               LET LEVEL(I)=TREE(*) LET TREE(*)=0
           LOOP
           END



           READ CODE
           FOR I=1 TO N,
           DO
                LET TREE(*)=LEVEL(I)
                FOR J=1 TO 2**(I-1),
                DO
                     IF TREE(J) EQUALS CODE, GO TO PRINT
                     OTHERWISE
                LOOP
           LOOP
           PRINT 1 LINE WITH CODE AS FOLLOWS
           UNABLE TO FIND AN ANCESTOR WITH THE CODE **
           STOP
'PRINT'    PRINT 1 LINE WITH CODE, J AND I AS FOLLOWS
     ANCESTOR ** FOUND IN POSITION * OF LEVEL *
           STOP
           END
```

Table II[3]

SIMSCRIPT II LIBRARY FUNCTIONS

Name	Arguments	Operation	Function Mode	Restrictions
ABS.F[†]	e	$\lvert e \rvert = \begin{cases} e & \text{if } e \geq 0 \\ -e & \text{if } e < 0 \end{cases}$	mode of argument	none
MAX.F[†]	$e_1, e_2, \ldots, e_n$	value of largest argument	INTEGER if all arguments INTEGER REAL if one argument REAL	none
MIN.F[†]	$e_1, e_2, \ldots, e_n$	value of smallest argument	INTEGER if all arguments INTEGER REAL if one argument REAL	none
MOD.F[†]	e_1, e_2	e_1-TRUNC.F(e_1/e_2)*e_2	INTEGER if all arguments INTEGER REAL if one argument REAL	$e_2 \neq 0$
DIV.F[†]	e_1, e_2	TRUNC.F(e_1/e_2)	INTEGER	e_1 and e_2 INTEGER; $e_2 \neq 0$
INT.F[†]	e	value of e rounded to an integer	INTEGER	none
REAL.F[†]	e	value of e expressed as a decimal number	REAL	none
FRAC.F	e	fractional part of e; e-TRUNC.F(e)	REAL	e must be REAL
TRUNC.F	e	integer part of e; e-FRAC.F(e)	INTEGER	e must be REAL
SIGN.F	e	1 if e > 0 0 if e = 0 -1 if e < 0	INTEGER	none
SFIELD.F	none	see Sec. 2-13	INTEGER	free-form input only
EFIELD.F	none	see Sec. 2-13	INTEGER	free-form input only
DIM.F	v	number of elements in array pointed to	INTEGER	v a pointer
SQRT.F	e	$\sqrt{e}$	REAL	$e \geq 0$ and REAL
EXP.F	e	exp(e) = EXP.C**e	REAL	e must be REAL
LOG.E.F	e	$\log_e(e)$	REAL	e > 0 and REAL
LOG.10.F	e	$\log_{10}(e)$	REAL	e > 0 and REAL
SIN.F	e	sin(e)	REAL	e REAL and expressed in radians
COS.F	e	cos(e)	REAL	e REAL and expressed in radians
TAN.F	e	tan(e)	REAL	e REAL and expressed in radians
ARCSIN.F	e	arcsin(e)	REAL	$-1 \leq e \leq 1$ and REAL
ARCCOS.F	e	arccos(e)	REAL	$-1 \leq e \leq 1$ and REAL
ARCTAN.F	e_1, e_2	arctan(e_1/e_2)	REAL	$(e_1, e_2) \neq (0,0)$ and REAL

[†] Denotes a function compiled in-line rather than called as a routine.

Table III[4]

STATISTICAL NAMES USED IN THE COMPUTE STATEMENT

Statistic	Alternative or Abbreviation	Computation
NUMBER	NUM	Number of items selected in the iteration
SUM		Sum of the selected values of the expression
MEAN	AVERAGE, AVG	SUM/NUMBER
SUM.OF.SQUARES	SSQ	Sum of squares of the selected values of the expression
MEAN.SQUARE	MSQ	SUM.OF.SQUARES/NUMBER
VARIANCE	VAR	MEAN.SQUARE - MEAN**2
STD.DEV	STD	SQRT.F(VARIANCE)
MAXIMUM	MAX	Maximum value of the selected values of the expression
MINIMUM	MIN	Minimum value of the selected values of the expression
MAXIMUM(e)	MAX(e)	Value of computed e using the control variable values that produce the expression with the MAXIMUM value
MINIMUM(e).	MIN(e)	Same as MAX(e) but for minimum

EXAMPLE III[5]

```
PREAMBLE
NORMALLY, MODE IS INTEGER
DEFINE COSTS AS A 2-DIMENSIONAL ARRAY
DEFINE MODE.FACTORS AND CLASS.FACTORS AS 1-DIMENSIONAL ARRAYS
DEFINE CAPACITY AS A 4-DIMENSIONAL ARRAY
DEFINE FROM, TO,MODE AND CLASS AS ''GLOBAL'' VARIABLES
END


MAIN
READ NFROM, NTO, NMODE,NCLASS   ''READ MAXIMUM DIMENSIONS
RESERVE COSTS(*,*) AS NFROM BY NTO, MODE.FACTORS(*) AS NMODE,
      CLASS.FACTORS(*) AS NCLASS,CAPACITY(*,*,*,*) AS NFROM
      BY NTO BY NCLASS BY NMODE
READ COSTS, CLASS.FACTORS,MODE.FACTORS   ''READ INITIAL DATA
'REQUEST'   READ FROM,TO,MODE,CLASS
'INQUIRE'   CALL RESERVATION YIELDING ANSWER
      IF ANSWER EQUALS 1
            NOW FIND.COST YIELDING PRICE
            PRINT 1 LINE WITH MODE,CLASS,FROM,TO,PRICE THUS
MODE * CLASS * RESERVATION FROM ** TO ** IS AVAILABLE FOR *** DOLLARS
            GO TO ''NEXT CUSTOMER'' REQUEST
      OTHERWISE ''FIND OTHER SPACE
            SUBTRACT 1 FROM CLASS
            IF CLASS IS GREATER THAN 0   GO TO INQUIRE
            OTHERWISE   LET CLASS=NCLASS
            SUBTRACT 1 FROM MODE
            IF MODE IS GREATER THAN 0   GO TO INQUIRE
            OTHERWISE PRINT 1 LINE WITH FROM AND TO LIKE THIS
THERE IS NO TRANSPORTATION AVAILABLE FROM ** TO ** TODAY
            GO TO REQUEST
            END ''OF MAIN ROUTINE''


ROUTINE FOR RESERVATION YIELDING ANSWER
IF CAPACITY(FROM,TO,CLASS,MODE) IS GREATER THAN 0
            SUBTRACT 1 FROM CAPACITY(FROM,TO,CLASS,MODE)
            LET ANSWER=1   RETURN
ELSE   LET ANSWER=0   RETURN
END


ROUTINE TO FIND.COST YIELDING SUM
LET SUM=COSTS(FROM,TO)*CLASS.FACTORS(CLASS)*MODE.FACTORS(MODE)
RETURN   END
```

SECTION II

In this section we will be discussing Entities, Attributes, and Sets, and their associated declarations and operations. We begin with these definitions:

> "An ENTITY is a program element, much like a variable, that exixts in a modeled system. It is like a subscripted variable in that it has values, called ATTRIBUTES, associated with it that, when assigned specific values, define a particular configuration or state of the entity." 6

> "Sets...are collections of entities organized by systems of pointers. Set owners point to the first and last members of sets; set members point to one another. Sets are like arrays in that they are composed of elements that can be identified and manipulated, but are unlike arrays in their method of organization and their dynamic and changeable, rather than static and fixed, nature." 7

We begin the study of these "objects" with a discussion of the Preamble in order to understand how they are declared.

The Preamble

Entity classes are given attribute catagories in the Preamble. Any attribute that any member of an entity class may have must be declared in the Preamble if a member is later to be given it. For example, suppose an entity class--MAN has the following attributes: Age, Dependents, and a Social Security Number. This could be declared as follows:

```
EVERY MAN HAS AN AGE, SOME DEPENDENTS AND A
SOCIAL.SECURITY.NUMBER
```

If another entity has the attribute AGE, it must occur in the same relative position as it did in MAN. (i.e.

EVERY DOG HAS AN AGE AND A BREED

is o.k., but

EVERY DOG HAS A BREED AND AN AGE

is not.) This is because AGE(entity) is translated into "the value found in the first word of the record indexed by the value entity" (where entity may be MAN or DOG in the above example)..

There are two types of entities, temporary entities, and permanent entities, which will be discussed later. Every entity must be declared to be one or the other of these in the Preamble. The statements:: TEMPORARY ENTITIES and PERMANENT ENTITIES precede the lists of each type (i.e. the EVERY statements).

It is important to remember that entities are not created in the Preamble (i.e. no storage space is allocated to an entity because it appears in the Preamble), but a class of entities must appear in the Preamble before it can be created in the program. Also remember that if only one member of an entity class has a particular attribute, that attribute still must be declared for that entity class (i.e. it is possible for any member of that class to have that attribute).

Entities may "own" sets of entities, and they may be "owned" by other entities, as in:

EVERY COMMUNITY OWNS A MASONS

EVERY MAN MAY BELONG TO THE MASONS

As with attributes, the possibility of set ownership or membership must be declared in the Preamble.

Note that the words HAVE and HAS denote attributes assigned to entities; the words OWN and OWNS denote set ownership by an entity; and the words BELONG and BELONGS denote set membership of entities to the sets following these words.

Temporary Entities

If the temporary entity MAN is declared in the Preamble as follows:

EVERY MAN HAS AN AGE, OWNS A FAMILY, MAY BELONG TO THE MASONS AND HAS A BIRTH.DATE

the statement:

CREAT A MAN CALLED JOHN

results in JOHN pointing to a record in storage which looks like:

AGE
F.FAMILY
L.FAMILY
N.FAMILY
P.MASONS
S.MASONS
M.MASONS
BIRTH.DATE

In this example, AGE is the value of the attribute AGE; F.FAMILY is a pointer to the first element in the set FAMILY(JOHN); L.FAMILY is a pointer to the last element in the set FAMILY(JOHN); P.MASONS is a pointer to the previous element in the set of masons in his community (MASONS(ANN.ARBOR)) (this pointer is zero if JOHN is not in the set, or if he is the first element of the set); S.MASONS is a pointer to the next member of the set MASONS(ANN.ARBOR) (this pointer is zero if JOHN is not a member of the set, of if he is the last member of the set); M.MASONS is equal to one (1) if JOHN is a member of the masons, and is zero otherwise; N.FAMILY is the number of elements in the set FAMILY(JOHN); and BIRTH.DATE contains the value of the attribute BIRTH.DATE.

A temporary entity is created whenever it is needed and destroyed individually with a destroy statement.

e.g. DESTROY THE MAN CALLED JOHN

Permanent Entities

If MAN is declared a permanent entity in the following statement:

```
EVERY MAN HAS AN AGE,. OWNS A FAMILY AND MAY BELONG TO THE
MASONS
```

then every MAN is created at the same time (as contrasted with temporary entities, which are created as needed, and their total number is not fixed, as it is with permanent entities). The following two sets of statements each create the entities MAN:

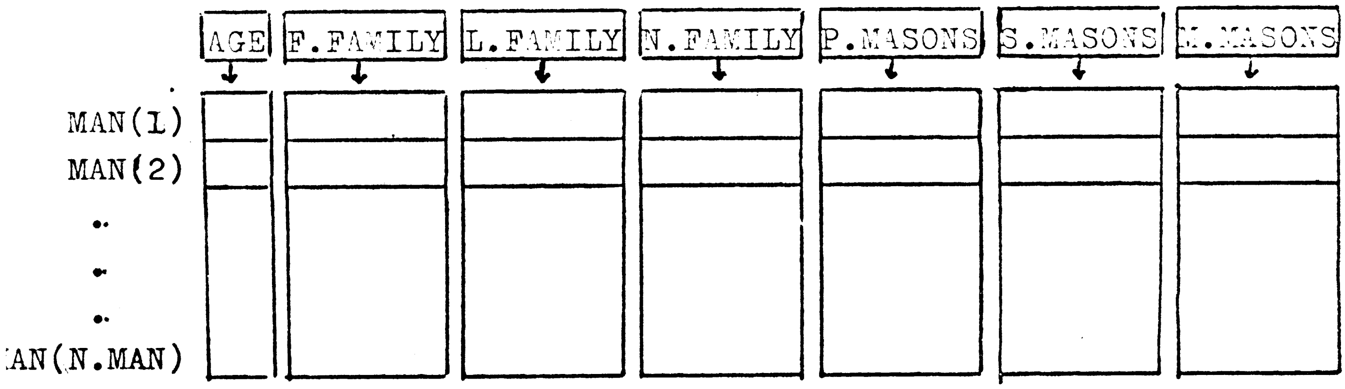
```
READ N.MAN
CREATE EVERY MAN
```

or

```
LET N.MAN = 5
CREATE EVERY MAN
```

The first two statements create the number of men read into the variable N.MAN, a system variable setup automatically when MAN was declared a permanent entity. In the second set of two statements this system variable is set to five, and thus five men are created.

When the CREATE statement is executed, storage is setup in blocks, where each block contains the values of one particular attribute for all of the entities of the type MAN. (This is a completely different scheme than with temporary entities, which have their own record containing all of the attribute and set data for that particular entity).
A permanent entity class looks like this in storage:



To destroy a permanent entity, all entities of that type must be destroyed at the same time. This is done by releasing all of the entity class's attributes. For example:

```
RELEASE AGE,F.FAMILY,L.FAMILY,N.FAMILY,P.MASONS,S.MASONS,M.MASONS
```

Examples of Simple Set Filing Routines

The following are examples of filing and removal routines. Their individual functions are obvious.

```
FILE ROVER FIRST IN KENNEL(1)
FILE ROVER LAST IN KENNEL(1)
FILE ROVER BEFORE SAM IN KENNEL(2)
FILE ROVER AFTER SAM IN KENNEL(2)
REMOVE FIRST DOG FROM KENNEL(1)
REMOVE LAST DOG FROM KENNEL(1)
REMOVE ROVER FROM KENNEL(2)
```

The default for FILE is LAST, so

```
FILE ROVER LAST IN KENNEL(1)
```

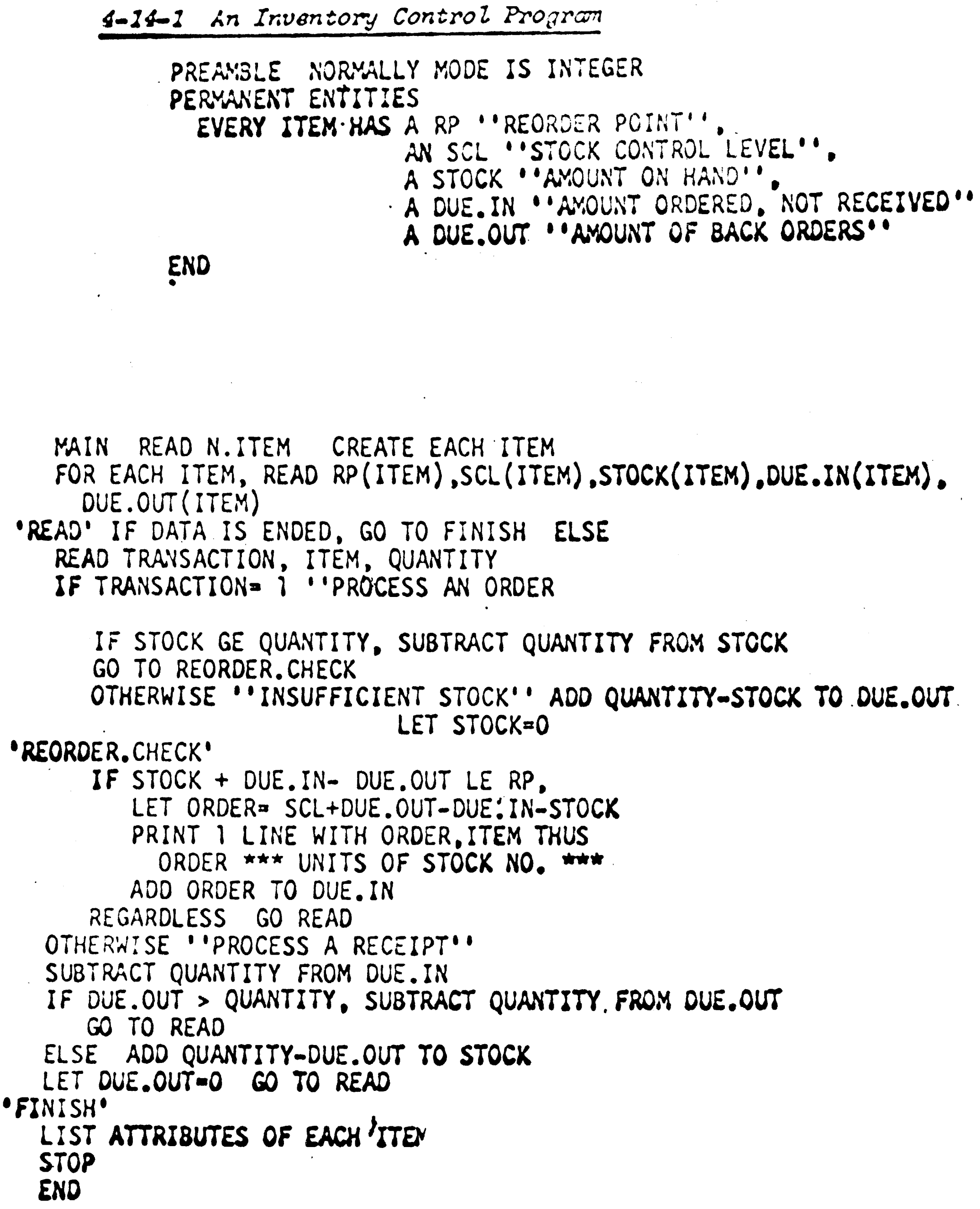
and

```
FILE ROVER IN KENNEL(1)
```

are equivalent.

The final two examples are easily understood and need no further explanation.

EXAMPLE IV[8]



4-14-1 An Inventory Control Program

```
          PREAMBLE  NORMALLY MODE IS INTEGER
          PERMANENT ENTITIES
            EVERY ITEM HAS A RP ''REORDER POINT'',
                           AN SCL ''STOCK CONTROL LEVEL'',
                           A STOCK ''AMOUNT ON HAND'',
                           A DUE.IN ''AMOUNT ORDERED, NOT RECEIVED'',
                           A DUE.OUT ''AMOUNT OF BACK ORDERS''
          END


   MAIN  READ N.ITEM   CREATE EACH ITEM
   FOR EACH ITEM, READ RP(ITEM),SCL(ITEM),STOCK(ITEM),DUE.IN(ITEM),
     DUE.OUT(ITEM)
'READ' IF DATA IS ENDED, GO TO FINISH  ELSE
   READ TRANSACTION, ITEM, QUANTITY
   IF TRANSACTION= 1 ''PROCESS AN ORDER

      IF STOCK GE QUANTITY, SUBTRACT QUANTITY FROM STOCK
      GO TO REORDER.CHECK
      OTHERWISE ''INSUFFICIENT STOCK'' ADD QUANTITY-STOCK TO DUE.OUT
                                LET STOCK=0
'REORDER.CHECK'
      IF STOCK + DUE.IN- DUE.OUT LE RP,
         LET ORDER= SCL+DUE.OUT-DUE.IN-STOCK
         PRINT 1 LINE WITH ORDER,ITEM THUS
           ORDER *** UNITS OF STOCK NO. ***
         ADD ORDER TO DUE.IN
      REGARDLESS  GO READ
   OTHERWISE ''PROCESS A RECEIPT''
   SUBTRACT QUANTITY FROM DUE.IN
   IF DUE.OUT > QUANTITY, SUBTRACT QUANTITY FROM DUE.OUT
      GO TO READ
   ELSE  ADD QUANTITY-DUE.OUT TO STOCK
   LET DUE.OUT=0  GO TO READ
'FINISH'
   LIST ATTRIBUTES OF EACH ITEM
   STOP
   END
```

```
PREAMBLE
PERMANENT ENTITIES

FVERY FARM OWNS A KENNEL
TEMPORARY ENTITIES
EVERY DOG HAS A NAME AND BELONGS TO SOME KENNEL
DEFINE NAME AS AN ALPHA VARIABLE
NO RMALLY MODE IS INTEGER

END
LFT N.FARM = 2
CREATE EVERY FARM
READ NUMBER.OF.DOGS
F OR I = 1 TO NUMBER.OF.DOGS

   DO
   CREATE A DOG
   READ NAME(DOG) AND F AS /, A 4, I 1
   F ILE DOG FIRST IN KENNEL(F)

   LOOP

F OR EACH DOG IN KENNEL(1)
   DO
   PRINT 1 LINE WITH NAME(DOG) AS FOLLOWS

   **** IS IN KENNEL(1)

   LOOP
F OR EACH DOG IN KENNEL(2)
   DO

   PRINT 1 LINE WITH NAME(DOG) AS FOLLOWS
    **** IS IN KENNEL(2)
   LOOP
STO P
EN D

OF FILE
```

EXAMPLE V

```
 RUN *SIMSCRIPT2 SCARDS=-T SPRINT=-A SERCOM=-AA 0=-AAA
#EXECUTION BEGINS
#EXECUTION TERMINATED
#L -A

# END OF FILE
#L -AA
.      1      -**** SIMSCRIPT COMPILATION COMPLETED
.      2      OCPU TIME USED:    30.515 SECONDS
.      3      -**** *ASMG ASSEMBLY COMPLETED

.      4       OCPU TIME USED:    34.514 SECONDS
 END OF FILE
```

```
8
AB  1
DOG 2

FIDO1
ROVE1
SAM 1
JIM 2
DARK2

PAT 1
   PAT  IS IN KENNEI
   SAM  IS IN KENNEI
   ROVE IS IN KENNEI
   FIDO IS IN KENNEI

   AB   'IS IN KENNEI
   DARK IS IN KENNEI
   JIM  IS IN KENNEI
   DOG  IS IN KENNEI

#EXECUTION TERMINATEI
```

REFERENCES

1. Kiviat, P.J., Villanueva, R., and Markowitz, H.M., The Simscript II Programming Language, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1968 (Rand Corp.).

2. Simscript II.5 Reference Handbook, Consolidated Analysis Center, Inc., 12011 San Vicente Blvd., Los Angeles, Cali., 1971.

3. *SIMSCRIPT2, MTS Vol. II, p. 288.1 (in CCMEMO M198).

4. $COPY *SIM2NEWS

FOOTNOTES

All footnotes are taken from reference #1, listed above.

[0] I omitted a reference to more I/O information from SECTION I. pp. 153-192

[1] p.57

[2] pp.62-63

[3] p.91

[4] p.149

[5] p. 108

[6] p. 194

[7] pp. 223-224

[8] pp. 248-249

Paul Brindle

POP-2

A Data Structures Language

I. History and Aims of the language

A. Largely patterned after ALGOL,
also has elements of LISP,CPL,COBOL,PL/I,IPL-V,JOSS,TRAC,FORMULA ALGOL

B. Block structured and algorithmic

C. Immediate execution unless in function definition--interactive

D. Numverical facilities ignored in favor of structure manipulation

II. Basic concepts and facilities.

A. Operates on items (one-word entities)

1. Simple items--integers and reals
represented by themselves

2. Compound items--pointers to data structures and functions

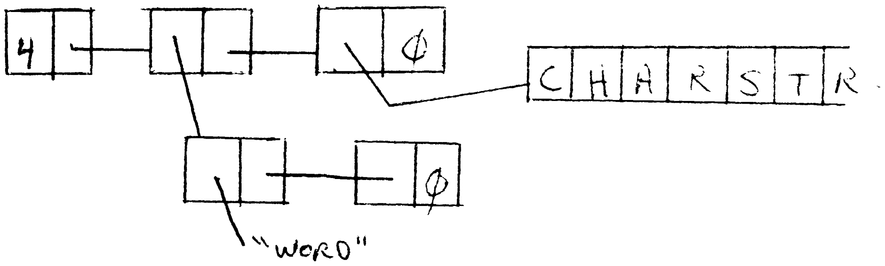
3. No types at declaration--manipulate items at will
types checked at operation execution.

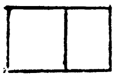
```
Ex.    FUNCTION A  ...
           END;
        VARS B;
         A —> B;   B(17) —> C;
         4 —> B;   B+8 —> D;
```

B. Standard data structures

1. Lists

HD,TL

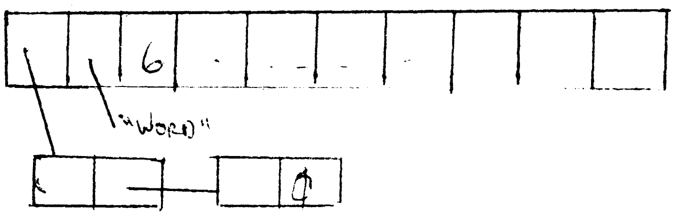


2. Pairs 

FRONT,BACK

3. Strips (vectors) and Character strips

SUBSCR(1,STRIP)



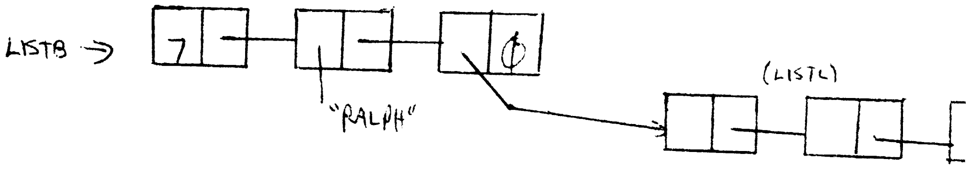
4. Words

C C. Structures have 3 associated functions:

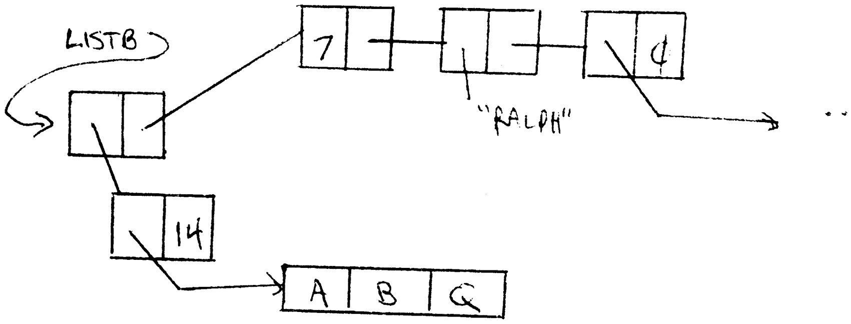
1. Constructor
2. Destructor
3. Doublet--accessor/updater fns for items in structure

Ex.
```
7 -> A;
"RALPH" -> B;

[%A,B,LISTL%] -> LISTB;
```



```
CONS(CONSPAIR('ABQ',14),LISTB) -> LISTB;
```



D. Use of stack

1. Temporary store
   Ex. switch elements `A,B -> A -> B;`

2. Parameter passage
   Ex. F is function of one variable to operate on a number mod B:

```
A // B;  .ERASE;  .F;
```

III. Definition capabilities

A. Functions--implicit assignments in FUNCTION type declaration
explicit assignments in LAMBDA type declaration

Ex.
```
FUNCTION F X Y; blah blah;
      END;

    is same as

VARS F;
(LAMBDA X Y: blah blah; END) -> F;
```

B. Operations

```
   Ex.    VARS OPERATION 7 ==;

             FUNCTION  ==  x y ; ... END;

                 IF J==P THEN  ...
```

C. Partial application

```
    Ex.  Given  SORT(N,TEST,DOUBLET)

             SORT(%SUBSCR%) -> STRIPSORT;
             SORT(%ALPHATEST,SUBSCR%) -> ALPHASORT;

                       ALPHASORT(DATALENGTH(S)) -> S;
```

D. Creation of new data structures

1. RECORDFNS --create new types of beads with custom fields get necessary functions to easily link the beads for yourself

```
          Ex.  RECORDFNS("PAIR", [0 0] ) -> BACK -> FRONT -> DESTPAIR -> CONSPAIR;

               RECORDFNS("PERSON", [0 0 0 1 7] )
                      -> AGE -> SEX -> SPOUSE -> SURNAME -> FORENAME
                       -> DESTPERSON -> CONSPERSON;

               CONSPERSON("JOHN","SMITH",0,0,22) -> JOHN;

                FUNCTION MARRIAGE  BOY GIRL;

                     IF SEX(BOY) = SEX(GIRL) THEN
                                   PRINT('FROWNED UPON');
                            ELSE
                                   SURNAME(BOY) -> SURNAME(GIRL);

                         COMMENT ADMITTEDLY SEXIST;
                                    GIRL -> SPOUSE(BOY);
                                    BOY -> SPOUSE(GIRL);
                           CLOSE;
                END;
```

2. STRIPFNS

```
         Ex.     STRIPFNS("CHARXEXSTR" ,   8) -> INITC -> SUBSCRC  ;
```

3. NEWANYARRAY

```
         Ex.    NEWANYARRAY( [%1,N,N2,N5%] ,(LAMBDA X Y; X+Y; END),
                      INIT,SUBSCR) -> ARRAY;

                NEWANYARRAY(%INIT,SUBSCR%) -> NEWARRAY;
```

IV. Summary--POP-2 presents the user with a very pleasing simplicity of manipulation and ability to define natural structures, making programming easy and transparent.

V. References--POP-2 Reference Manual in Mitchie (ed.), Machine Intelligence, Vol. 2

DYNAMIC STORAGE MANAGEMENT

William E. Riddle
5-24-72

I. Introduction

A. Dynamic storage management is effected by a set of routines (DSM routines):

- which are a part of either the system supervisor or the programming language support package,
- and which provide an ability to acquire and release storage during program execution.

B. The general benefit which accrues is an ability to delay the binding of a storage segment's size and/or location.

- For the system supervisor this facilitates increasing overall system performance by allowing:
  - reuse of a storage segment for different jobs
  - dynamic adjustment to system load
- For the higher-level language programmer this facilitates program generality by allowing:
  - increased generality inherent in not having to specify maximum data structure size
  - reuse of a storage segment for different data structures
  - data structures which have a size or organization which changes during program execution
  - recursive procedures

C. The set of DSM routines must possess some or all of the following capabilities:

- selection of a storage segment from available storage and of sufficient size to satisfy a request
- reclamation of previously allocated but presently unused storage segments
- recombination of (either adjacent or non-adjacent) available storage segments

D. Two gross attributes of a set of DSM routines are:

- their visibility to the user - must he explicitly command their invocation (as in PL/I) or are they implicitly invoked as required (as in LISP)

- the general flavor of the allocation they perform:
  - "large" segments to be used for "long" periods of time such as during the entire execution of the program which requested the storage
  - "small" segments to be used for "short" periods of time such as during only part of the requesting program's execution

E. Most all DSM routines operate in the following general manner:

- maintain some private data structure, AVAIL, which identifies all of the available storage which can be used to satisfy a request
- when a previously allocated storage segment is returned, update AVAIL so that the segment is known to be available
  - the information specifying the size and location of the segment:
    - could have been retained by the routines when the segment was allocated
    - or could have been saved within the segment (either protected or unprotected)
    - or could be stated by the agent who returns the segment
  - the newly returned segment could be combined with other (usually adjacent) available segments
- when a request for a storage segment is received, then it is satisfied by using the information in AVAIL to select a segment
  - the selected segment may be larger than requested.
  - if there are no segments identified by AVAIL which are large enough to satisfy the request:
    - the DSM routines may refuse to satisfy the request and (may or may not) notify the requestor
    - the DSM routines may try to recombine several (adjacent or non-adjacent) segments to obtain a large enough one
    - the DSM routines may try to reclaim segments which were allocated previously and are no longer being used but have not been returned

II. Sequential Allocation

A. All managed storage is assumed to be in one large block and there is a dividing point such that storage above that point has been allocated and storage below that point is still available. AVAIL is just a pointer, FREEPOINT, to the dividing point.

B. A request for n bytes of storage is satisfied by allocating the next n bytes starting at FREEPOINT.

- If n is larger than amount of storage below FREEPOINT, try to recombine unused areas which fall above FREEPOINT before refusing to satisfy request

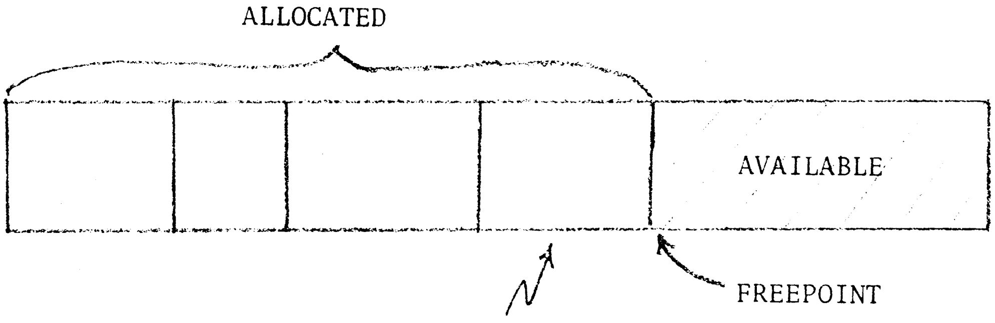
C. Upon return of a storage segment can either:

- mark it unused and leave recombination until later when it is necessary

- recombine the newly returned segment with the rest of available storage

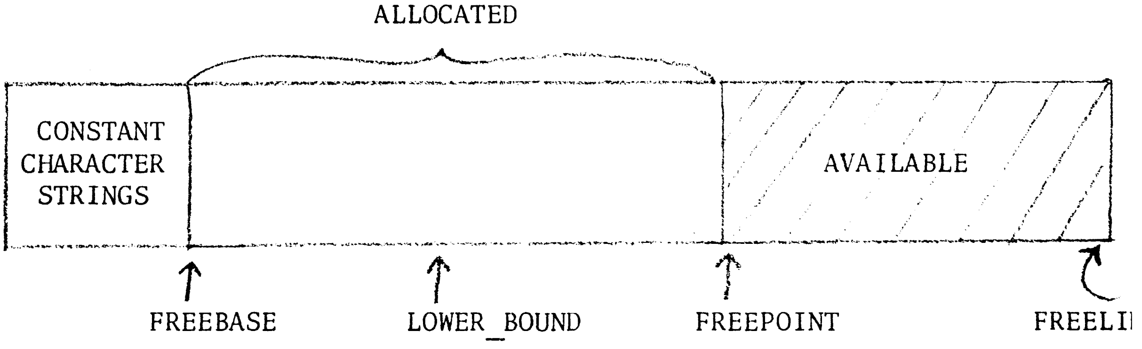
D. Two examples

1. "Stack" allocation - storage segments are returned in the reverse order of that in which they were allocated. Used for automatic variables in PL/I.



Last segment allocated; next to be returned.

2. Sequential allocation without explicit return - must compactify allocated storage, reclaiming and recombining available segments. Used in XPL for strings.



DESCRIPTORS pointers into the allocated area

FREELIMIT has a value which is 256 short of the end of the managed area.

Upon receiving a request for n (≤256) bytes:

A. if FREEPOINT is ≤ FREELIMIT then satisfy request and update FREEPOINT

B. otherwise reclaim and recombine currently unused segments in ALLOCATED portion.

B.1. try a minor compactification first

B.1.1. copy into DX all those descriptors in DESCRIPTORS which point into area between LOWER_BOUND and FREEPOINT

B.1.2. sort DX and use result to control moving of referenced values up toward LOWER_BOUND

B.2. if not enough storage was reclaimed, then try a major compatification - same as above but work on region between FREEBASE and FREEPOINT.

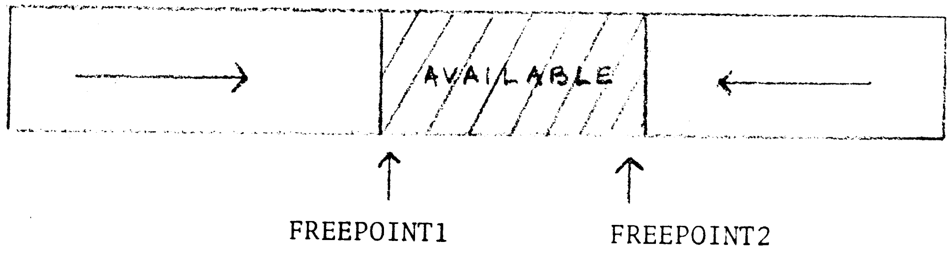
B.3. if still not enough storage was reclaimed then refuse to satisfy request.

B.4. update FREEPOINT and LOWER_BOUND to point at end of ALLOCATED region

Indicates GENERAL RULE: the most work is done for the least benefit.

E. When there is more than one pool.

1. If there are two pools point them at each other so that limit of one is current extend of the other.



2. If there are more than two:

   A. Initially, space them out either evenly or according to their estimated maximum size.

   B. When one runs into another:

      1. allocate more to cramped region by:

         a. moving the information in next one

         b. taking space away from least active region and moving others accordingly

         c. taking some space away from each region in proportion to its activity

III. Non-sequential Allocation

A. Instead of having the unallocated regions (which DSM knows about) all contiguous within one area, they may be scattered through storage. To keep track of them, they are chained together on a linked list and part of AVAIL holds a pointer to the start of the chain.

B. Explicit return - no need to do reclamation since DSM routines assume user will notify when a region is no longer needed.

1. If segments are all the same size then treat list of available segments as a stack, popping one off to satisfy a request and pushing a returned segment onto the stack.

2. If segments vary in size then there are some problems.

   a. If request is for segment of size n, then must first find an available segment of size $m \geq n$. Then allocate n bytes from the segment and return m-n byte segment to list of available segments. (Don't recombine.)

1. Best Fit: choose available segment such that m-n is minimized. This has a long search time and will proliferate small segments.

2. First Fit: choose first segment on list which has $m \geq n$. This has shorter search time and can be expected to generate fewer small segments. Shown in simulation by Knuth to be better.

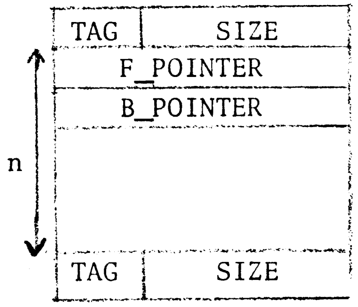
3. Avoid small segment proliferation by never generating a segment of size <K. Must remember actual size.

4. Speed up best fit searching by having list ordered by increasing size. Will increase insertion time so very little overall gain.

b. When a segment is returned, just stick it back onto list. However, if two available segments are actually contiguous, want to combine them and have one entry on list.

   1. If list is kept ordered by starting address then check successor and predecessor and combine if warranted. Will want two way linked list.

   2. Insertion is quicker if list doesn't have to be ordered. Store some information with the block and use BOUNDARY TAG Method (due to Knuth).



TAG = 1 if available
      = 0 if allocated

SIZE = n+8

POINTERs used only in available segments.

Neighbors of a block at location a can be checked by looking at a-1 and a+SIZE.

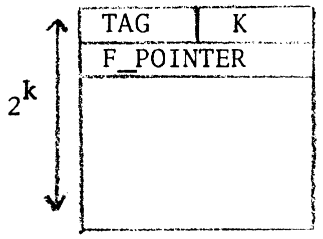
   3. Another way is to use the BUDDY SYSTEM (due to Knowlton).

      . a segment's BUDDY is a contiguous block which the segment may be combined with

         . both segment and its BUDDY have size 2^k for some k.

         . if segment's address is a and segment's size is 2^k then BUDDY'S address is $a+2^k$ (addition without carry).

- segments look like



TAG = 1 if available
    = 0 if allocated

POINTER needed only in available segments.

- F_POINTER is used to maintain a stack of available segments. A separate stack is kept for each size k, $1 \le k \le n$.

- When a request is received for segment of size 2^k first look in stack for that size. If none available, then look for segments of size 2^{k+1}. If one is available with that size, split it in half, allocate one half and put other on 2^k stack. If none available look at 2^{k+2} and split it twice. Etc.

- When a segment is returned, check to see if BUDDY is free and combine if possible. Keep combining more buddies as much as possible.

- Two problems:

  1. adjacent non-BUDDYs may be available but not combined - Knuth's simulations show that this isn't frequent.

  2. there is waste if there are many instances of needing slightly more than 2^k for some k.

C. Implicit return - the primary problem now is reclamation.

1. If segments are of differing sizes need to augment algorithms to take notice of segment size. In the rest of this section, assume that all segments are of the same size.

2. If segments don't all have the same pattern then need some extra information to indicate where pointers are within a segment. In the rest of this section, assume that all segments have the same pattern.

3. Whenever the list of available segments is empty, more segments are reclaimed by a "garbage collection" routine:

   1. scan through entire managed area and mark all of those segments which are still being used (assume all segments are initially unmarked)

   2. scan again, accumulating all unmarked segments into the list of available segments and unmarking all marked segments.

4. In order to be able to mark segments need to set aside a bit for each either within the segment or in a bit table.

5. The first scan through the managed area must be guided by some information concerning the used areas. Usual situation is that managed area is laced with list structures. In such a situation, we assume that some table gives addresses to the heads of all the lists. To mark, we go down each list structure -

   a. At each node, push n-1 of the branch addresses onto a stack and follow the nth. When a terminal node is reached then pop an address off the stack and continue.

      Problem: need free storage (an arbitrary amount of it) to collect free storage.

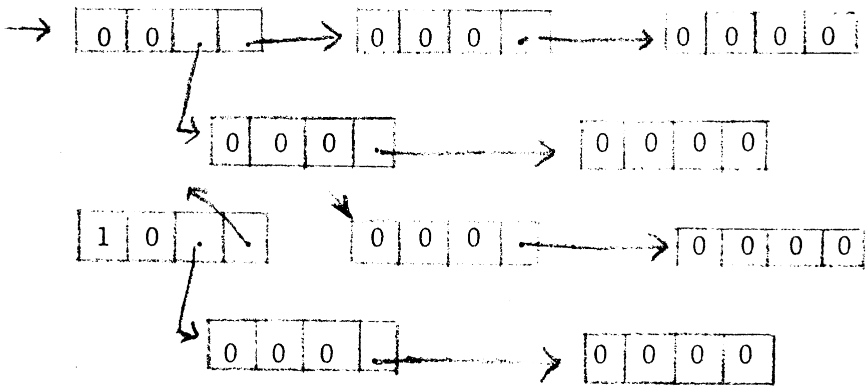
   b. Pointer Reversal Technique (Schorr & Waite)

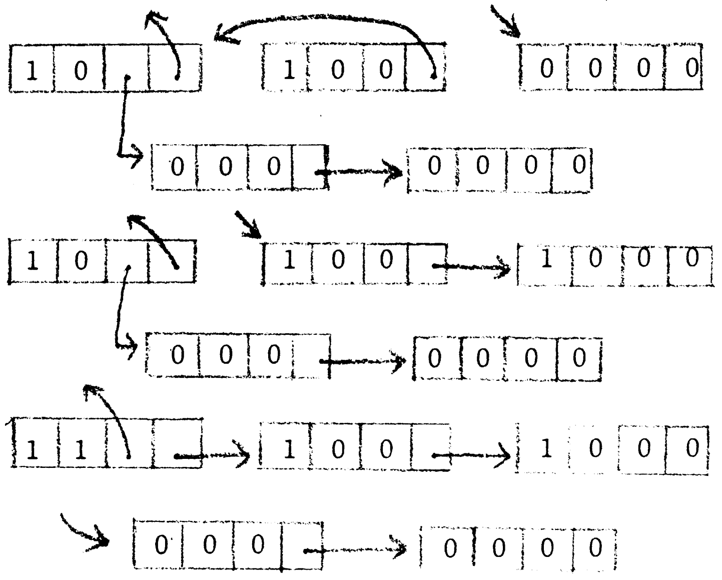
      . Don't need an extra stack. Instead, the pointer fields of the nodes are used to hold a backward pointing chain (built while going forward) which can be followed to return to an as yet unfollowed branch.

      . Need another bit in a two-branch node:

Mark Bit	Flag	Pointer	Pointer

      . Example:





etc.

. If there are n branches from each node, need FLAG field k bits wide where $2^{k-1} < n \le 2^k$

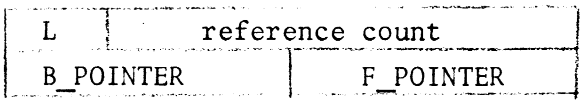
6. Instead of collecting the garbage when there is no more available space, collect it as it is generated by maintaining a reference count. The count is increased whenever a pointer is made to reference the node (or reference a node which references the node). The count is decreased when a pointer is made to not reference the node.

   A. To preserve integrity, some central agent must perform all pointer manipulation.

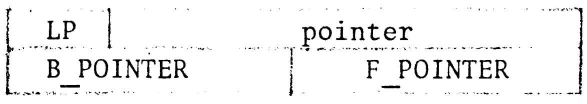
   B. Need extra storage in each node to hold the count

   C. Extra storage required may be reduced by keeping a count only for lists and sublists. (SLIP scheme, Weizenbaum)

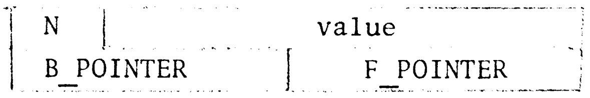
      . three node types



Header



Reference to a list



Value on a list

. when reference count within a header goes to zero, put it onto the free list.

   . because of pointers in header this automatically puts all of the first level nodes on the free list.

   - the reference counts in the sublists referenced by the list are not updated at this time.

- when a node is taken off the free list check to see if it is of type LP.

   - if it is, then decrease reference count of list this node points to and put that list on if warranted.

   - this way, free storage is reclaimed

      - as it is generated

      - in larger segments - whole lists at one time

      - as it is needed

- two problems:

   1. if only part of a list is actually referenced, the whole list must be retained because there can be only one reference count.

   2. a recursive list cannot be handled without special processing - its reference count would always be at least 1.

IV. References

A.J. Bertiss. Data Structures, Theory and Practice. Academic Press, 1971.

Schorr and Waite. An efficient machine-independent procedure for garbage collection in various list structures. CACM 10, 8 (August 1967), 501.

Tonge. Notes on Data Structures, In Engineering Summer Conference Notes, 1970.

Knuth. The Art of Computer Programming, Vol. I. Addison-Wesley, 1968.

Knowlton. A fast storage allocator. CACM 8, 10 (October 1965), 623-625.

Weizenbaum. Symmetric List Processor. CACM 6, 9 (September 1963), 524-544.

PL/I LANGUAGE INTERNALS

by Morton D. Hoffman

PL/I was designed as a general purpose algorithmic language. As such, its list processing facility is designed to fit into a more general scheme. To provide the most general data structuring capability, PL/I provides a pointer data type, bringing into the external language, a level of detail often reserved for the internal structure of other languages. For this reason the system cannot always know when a storage area is no longer useful, and storage management must often be explicitly invoked by the user. Therefore the internals of the list processing facility has two main components — internal data representation and Dynamic Storage Management.

Dynamic storage in PL/I is divided among three classes of variables: AUTOMATIC, CONTROLLED, and BASED. AUTOMATIC storage can be efficiently reclaimed by PL/I since only AUTOMATIC variables are associated with the invocation of a block. AUTOMATIC variables are stored in two structures — Dynamic Storage Areas (DSA) and Variable Data Areas (VDA), which together form the Run Time Stack (RTS).

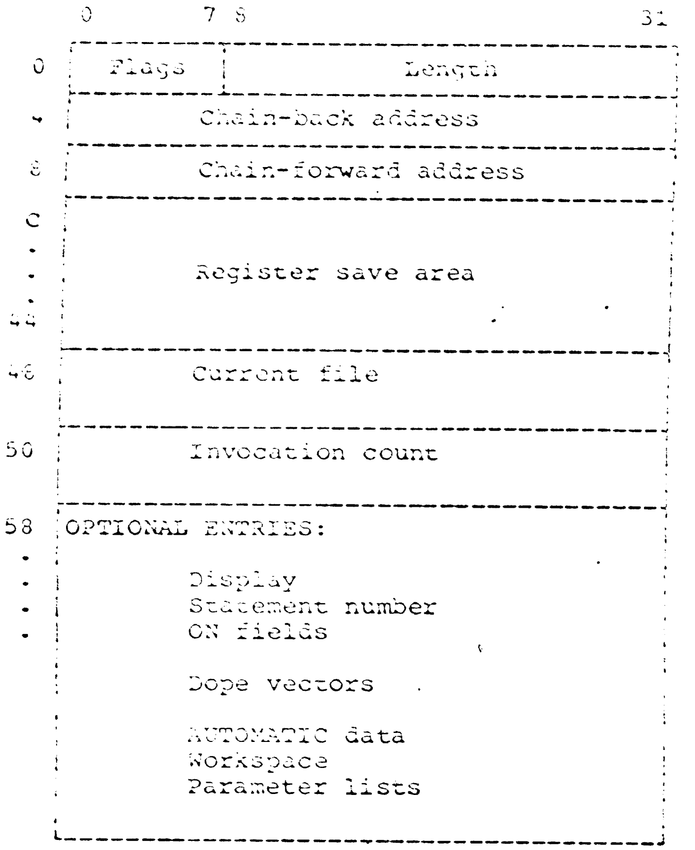


Figure 1 Format of the Dynamic Storage Area (DSA)

The first of these, the Dynamic Storage Area (fig. 1) is associated with the invocation of a block. This is the prime area for storage of automatic variables. The Dynamic Storage Area is also the repository for all information of the environment peculiar to the block. Its address is held in a pseudo-register; together the set of pseudo-registers pointing to Dynamic Storage Areas is called the display, since this construct now contains the description of the complete environment of a block. The Dynamic Storage Areas are chained together to provide housekeeping for dynamic storage (i.e. if several blocks are exited by one statement, the Dynamic Storage Areas for the intervening block can be located through the chaining).

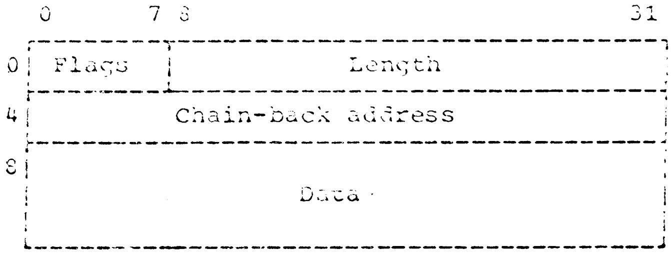


Figure 2 Format of the Variable Data Area (VDA)

The Variable Data Area (VDA) is quite similar to the Dynamic Storage Area except in its uses (fig. 2). One Variable Data Area is created at the invocation of a block. In it reside those automatic variables whose extent (length) is not known at compile time. Additional Variable Data Areas may be created for library work space (since the library routines are re-entrant, they require such external work space).. The first such area is called the Pseudo-Register Vector Variable Data Area (PRV VDA) and also contains the pseudo-registers. This area is allocated only once. Additional Variable Data Areas for library calls may be created in the form of secondary Library Work Space Variable Data Areas (LWS VDA) in blocks as needed. All the Variable Data Areas are chained into the Dynamic Storage Area chain (Run Time Stack) behind the

Dynamic Storage Area for their respective blocks.

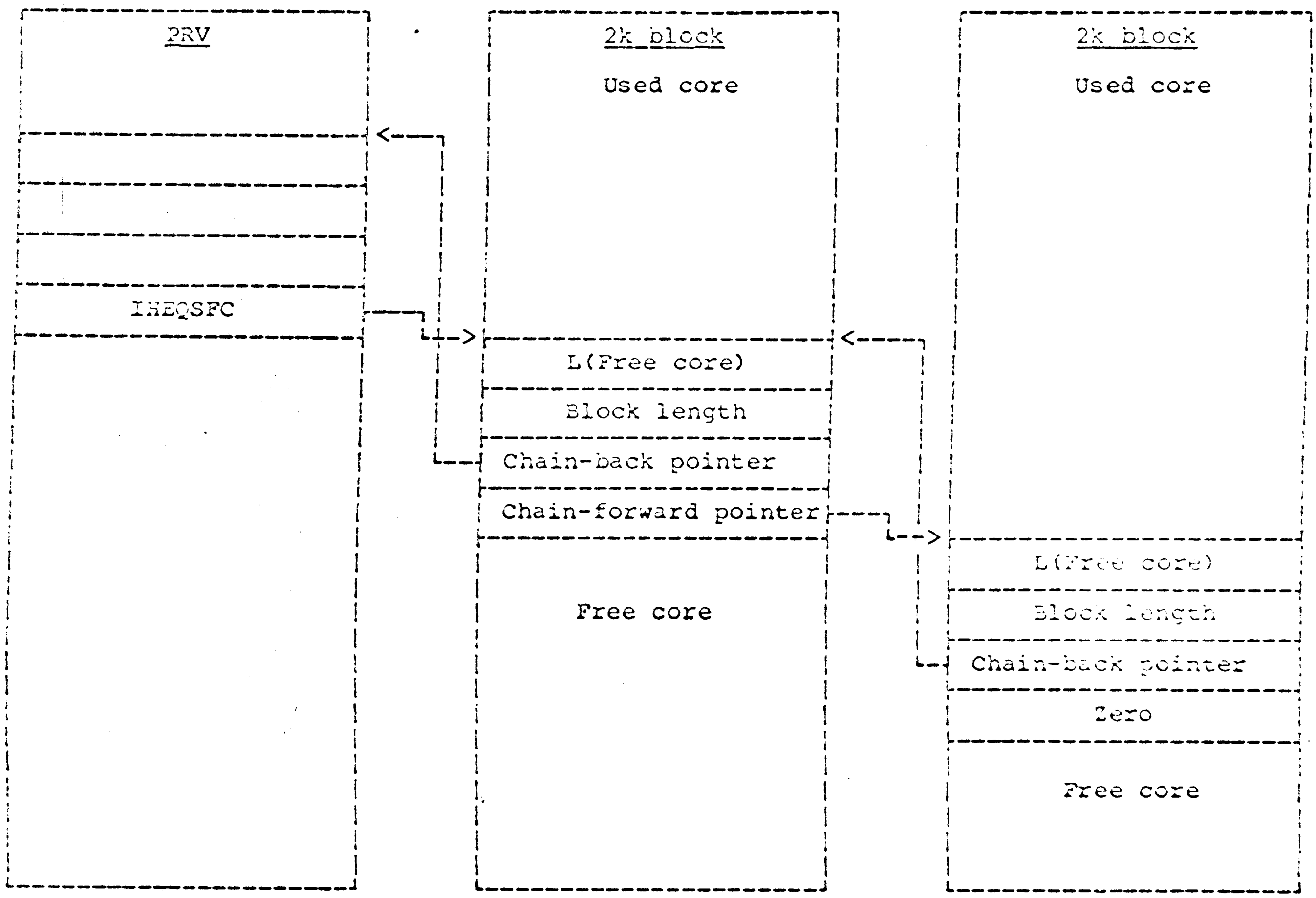


Figure 3 Structure of the Free-Core Chain for Automatic Variables

The management for Dynamic Storage Areas and Variable Data Areas are performed together. Initially 4K or 6K of storage is requested from the supervisor. As this storage is used, the remaining free area in this block is reduced. When a request can no longer be satisfied, an additional 2K block (or 2K multiple) is obtained from the supervisor. The free core areas in these blocks are chained in a doubly-linked list in the order of allocation (fig. 3). When requests for core are made, the free core chain is checked to see if there exists a free core region large enough to satisfy the request. In the event that a 2K area is entirely freed, the free core length will be recognized equal to the block length (both are stored at the beginning of the free core area of the block) and the block returned to the supervisor. Since automatic storage is always freed in the reverse order from which it is allocated, freed storage is always at the bottom of a block and there is at most one free area in any block.

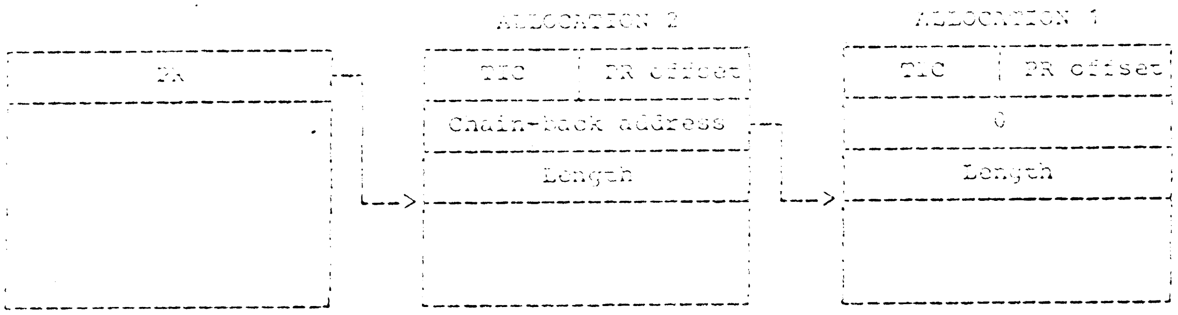


Figure 4 Storage Allocation for a Controlled Variable

Controlled Storage cannot be so handily managed, since variables ean be allocated and freed in arbitrary order. The result of this is that all ALLOCATE statements result in requests to the supervisor for an area large enough for the variable plus associated control information. Every controlled variable is represented by a chain of allocations off a pseudo-register (figure 4). When a controlled variable is first allocated, the pseudo-register (initially zero) is set to the address of this allocation. Each new allocation is chained in immediately following the pseudo-register. The address of the previous allocation is set in the new allocation, forming a chain of the generations of the controlled variable. In this scheme the chain forms a push-down stack, and the execution of a FREE statement results in the restoration of the previous allocation (popping the stack).

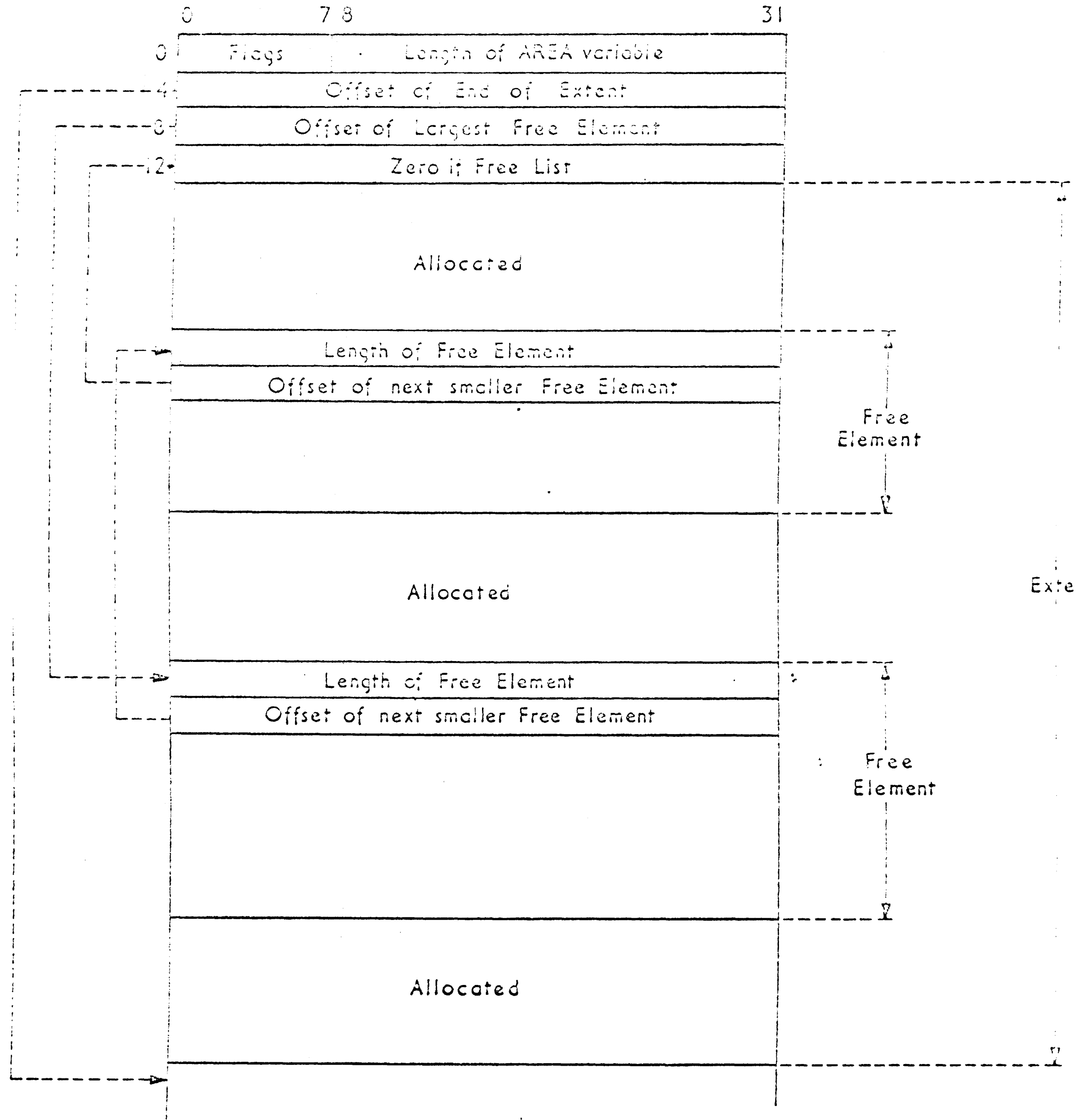
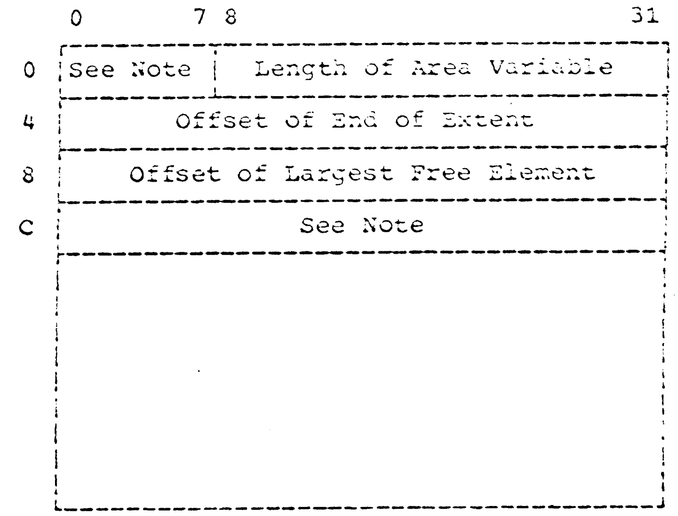


Figure 5 Format of Area Variable



Note: If the area variable contains a free list, bit 0 of the first byte is set to 1, and the fourth word is set to 0.

Figure 6. Format of Area Variable

Based storage is handled in much the same way as controlled storage. All storage requests are passed on to the supervisor, but now without any area for control information. Control information and the control stack is not needed here since a based variable does not have generations associated with it.

Based storage allocated within an area is handled differently. An area contains four words of control information, an area encompassing all allocated storage, and a final free area. Within the storage area encompassing the allocated storage (fig. 5 and 6), there is a free list anchored to the third word of AREA control information. It is organized by the largest free element on the list. The beginning of each free element has the length of the free element and the address of the next smaller free element. The smallest element points to a zero word (which serves as a free element of length zero). When a storage request comes in, the free list is checked first. If there are free elements large enough, the smallest such element is taken off the free list, divided into an element to fit the request, and the amount of storage left over (if any). This excess storage is put back on the free list in the position determined by its size. If no element on the free list is large enough, an allocation is made from the top of the last region of the area, which is unallocated storage. If this cannot satisfy the request, the request fails, resulting in a PL/I AREA condition.

Controlled Storage and Automatic Storage allocation and freeing are done in library module IHESA. Based variables, within and without AREAs are controlled by module IHELSP.

When storage in an AREA is freed, PL/I first attempts to merge it with an adjacent free element (and move the enlarged element to its new position on the free store chain). If it cannot be merged, it is then inserted by itself on the free list. It is not clear why the last free element in the area is kept off the free list. Also, the proposition that it is best to satisfy an allocation from the smallest free element large enough to satisfy the request has been brought into serious question by recent simulation studies.[1]

This provides a key for scanning the standard array, string and structure dope vectors. It consists of one entry for each major structure, minor structure and base element in the original declaration. Each entry consists of one word and can have one of two formats:

1. Structure:

```
 0  1  2           7 8                 15
┌──┬──┬────────────┬───────────────────┐
│F1│F2│     L      │         N         │
└──┴──┴────────────┴───────────────────┘
 16                                  31
┌──────────────────────────────────────┐
│               Offset                 │
└──────────────────────────────────────┘
```

F1 = 0 Structure

F2 = 0

L = Level of structure

N = Dimensionality, including inherited dimensions

Offset = Offset of containing structure from start of DVD
= - 1 for a major structure

2. Base element:

```
 0  1  2           7  8  9  10              15
┌──┬──┬────────────┬──┬──┬──────────────────┐
│F1│F2│     L      │F5│F6│        N         │
└──┴──┴────────────┴──┴──┴──────────────────┘
 16 17 18          23 24                    31
┌──┬──┬────────────┬──┬──┬──────────────────┐
│F3│F4│     A      │  │  │        D         │
└──┴──┴────────────┴──┴──┴──────────────────┘
```

F1 = 1 Base element

F2 = 0 Not end of structure
= 1 End of structure

L = Level of element

F5 = 1 Area variable
= 0 Not area variable

F6 = 1 Event variable
= 0 Not event variable

N = Dimensionality

F3 = 0 Not an aligned bit string
= 1 Aligned bit string

F4 = 0 Not a varying string
= 1 Varying string

A = Alignment in bits (0 to 63)

D = Length, if not a string, in bits
= 0 if a string, in which case the length is in the dope vector

Figure 7. Dope Vector Descriptor (DVD)

[1] Knuth, Donald E., The Art of Computer Programming (Vol. 1)

As in most languages complex data is keyed by dope vectors giving the format of the data. However, in PL/I another level of description is provided in the form of the Dope Vector Descriptor (DVD) (fig. 7). For an array, the Dope Vector Descriptor includes information indicating the number of subscripts in the array. For a string it includes alignment and an indicator as to whether the string is varying. Its major purpose, howeven is to indicate the variable type so that the dope vector may be correctly read.

For structures the Dope Vector Descriptor is more complicated. For each level in the structure, there is a dope vector descriptor giving its level in the structure, its dimensionality (including dimensions of containing sturctures) and the offset of the containing structure from the start of the Dope Vector Descriptor. Following the Dope Vector Descriptor are descriptors for the elements in the structure. Those elements include dimensions from the structure, since in PL/I, the elements inherit the subscripting of the structure. Again, parallel to these Dope Vector Deseriptors are the Dope Vectors for the structure.

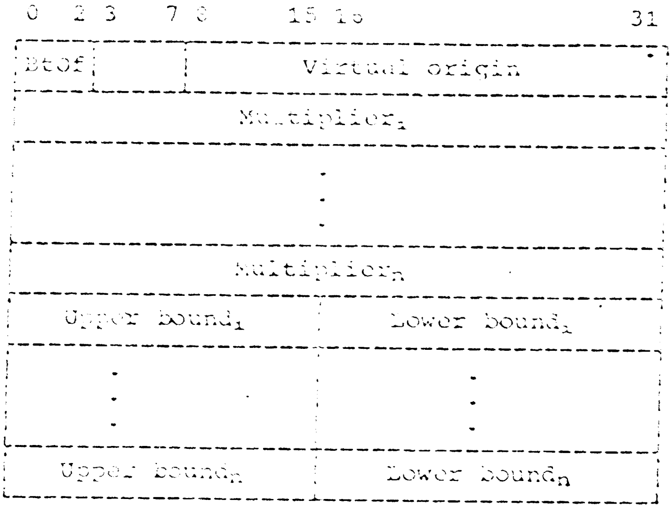


Figure 8 Format of the Array Dope Vector (ADV)

The most straightforward of the dope vectors is the Array Dope Vector (ADV) (fig. 8). It contains a virtual origin (the location where all subscripts are zero -- even if this is outside the array bounds), and for each subscript a multiplier and upper and lower bounds. This generality allows array operations -- since the dope vector contains not only accessing information, but also bounds information. Arrays are accessed by the equation:

$$\text{Address} = \text{virtual origin} + \Sigma S_i * M_i$$

where S_i's are subscripts and M_i's are multipliers

Note that this allows for interleaved arrays -- arrays which have their vectors separated in core (by equal sized areas outside the array). Interleaved arrays permit easy organization of structures in memory.

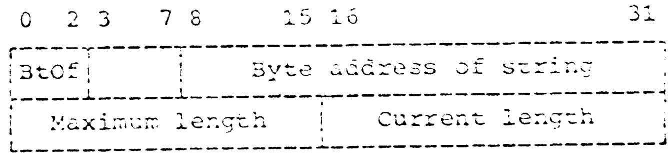


Figure 9 Format of the String Dope Vector (SDV)

Strings also have dope vectors (fig. 9). The String Dope Vector (SDV) contains the address of the string, and the maximum and current length of the string. For fixed length strings these two length are equal. The lengths are given in bits for BIT strings, and in bytes for CHARACTER strings. For bit strings, the bit offset from the beginning of the of the byte is also given. Not much more complicated is the String Array Dope Vector (SADV) (fig. 10). This is used for arrays of strings. It consist of a standard Array Dope Vector with a word appended to give the string length (maximum and current), For fixed length strings these two lengths are equal. For varying length string arrays, the current length is set to zero, and

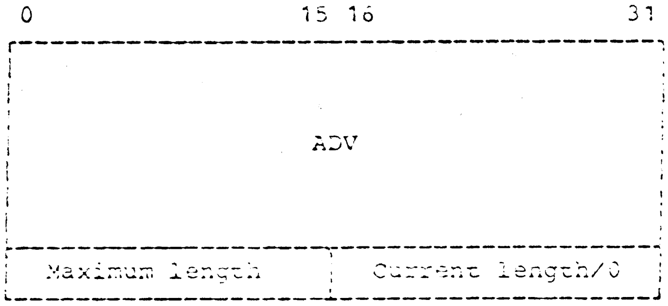


Figure 10 Format of the Primary String Array Dope Vector (SADV)

the array accessing algorithm yields the address of a standard string dope vector for the individual strings. This is necessary since in this case each string in the array has a (potentically) different length.

The final item to consider is the Structure Dope Vector. The structure dope vector is simply the concatenation of dope vectors of the elementary data types in the structure in the order in which they appear. Subscripting is handled by having the subscripts inherited by the component data elements, using the interleaving array notation to skip over the other parts of the structure which intervene in the core memory. This is necessary since the core data layout is exactly that described by the external structure statement and not by the organization of accessing rules (with respect to inherited subscripting). Here it is especially important to note the use of the dope vector descriptor to interpret the dope vector.

Within a routine some of this information may be contained in the code and the full data description may not be read. However, when aggregrates of data are passed among routines, these mechanisms are brought into use. With them PL/I is able to manipulate the highly complex data representations at the heart of all list-processing applications.

References (IBM Manuals):

PL/I Subroutine Library Program Logic Manual
Form GY 28 - 6801

PL/I (F) Language Reference Manual
Form GC 28 - 8201

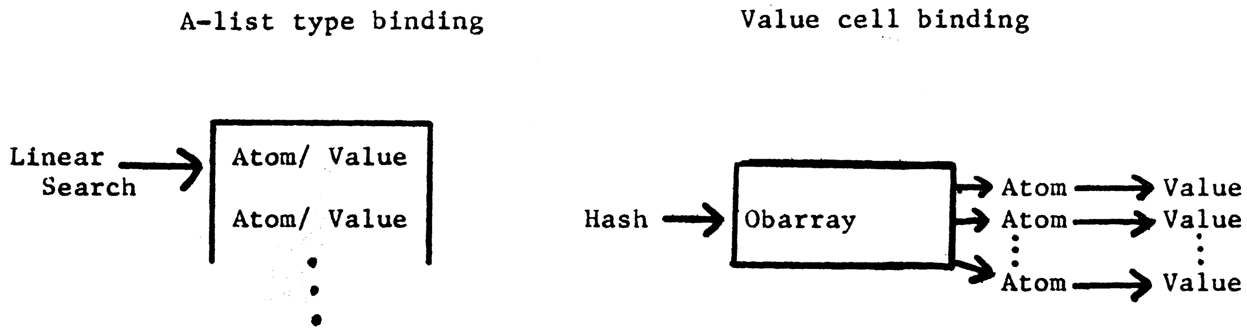
PL/I (F) Programmer's Guide
Form GC 28 - 6594

PL/I (F) Compiler Program Logic Manual
Form GY 28 - 6800

Carole Hafner
June 8, 1972

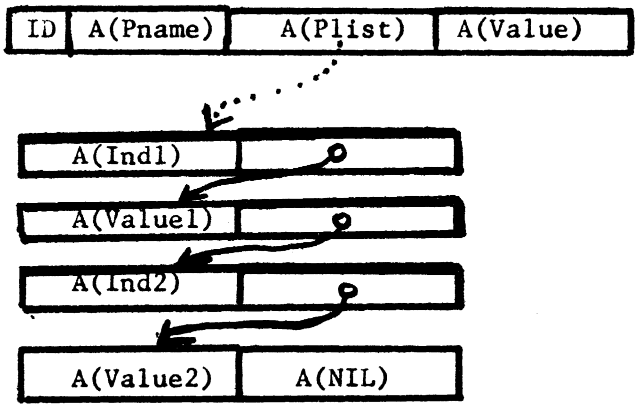
LISP INTERNALS

0. History - LISP 1.5 and MIT LISP



1. Atoms

A. Atom Header

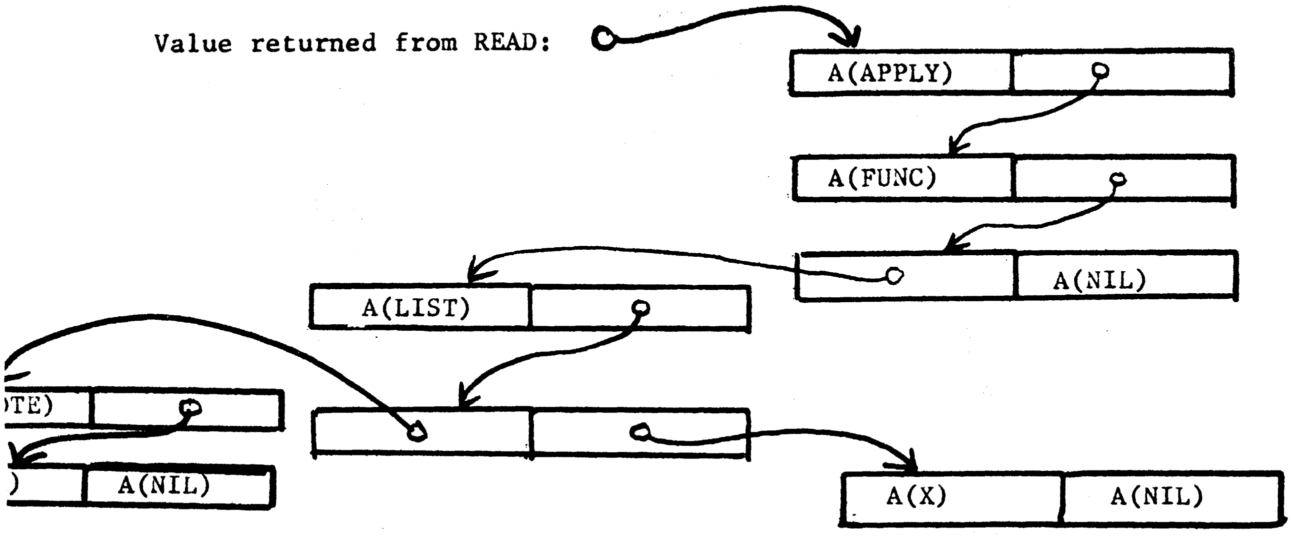


B. Property List

2. Syntax and Semantics

Input to READ: (APPLY FUNC (LIST (QUOTE A) X))

Value returned from READ:



3. Functions in LISP

A. How to invoke LISP functions:
If argument of EVAL is an atom, value of EVAL is the value of that atom. If argument of EVAL is a list (...) then invoke (CAR (...)) as a function where (CDR (...)) is a list of its arguments.

Ex. Input to EVAL: (APPEND L1 L2)
Action of EVAL: Call function APPEND with arguments L1 and L2

(CAR (...)) must be one of the following:

a. A LAMBDA or LABEL expression.
b. An atom with a function definition on its property list.
c. An atom which has a LAMBDA expression bound to it as its value.
d. A LISP expression which will itself EVAL to one of (a-d).

B. Defining and Accessing LISP functions:
An atom is defined as a function by having a function definition on its property list. The following special atoms, when used as indicators, indicate that the value to follow is a function definition.

a. SUBR - the value is a pointer to the entry point of a machine-coded function. Arguments (a fixed number) are EVALed before being passed.
b. FSUBR- the value is a pointer to the entry point of a machine-coded function. (CDR (...)) is passed as a single argument, and is not EVALed.
c. LSUBR- the value is a pointer to the entry point of a machine-coded function. Arguments (any number) are EVALed before being passed.
d. EXPR - the value is a LAMBDA expression. Arguments are EVALed.
   1. LAMBDA-SPREAD - arguments are bound to dummy arguments
   2. LAMBDA-NOSPREAD - arguments are pushed on stack and number of arguments bound to (single) dummy argument
e. FEXPR- the value is a NLAMBDA expression. Arguments are not EVALed.
   1. NLAMBDA-SPREAD - arguments bound to dummy arguments
   2. NLAMBDA-NOSPREAD - list of arguments bound to single dummy argument.
e. MACRO-the value is a NLAMBDA expression. The entire form (...) is bound to a single dummy argument. The value returned from the macro function is itself then EVALed.

If (CAR (...)) is an atom, EVAL looks for one of these indicators on its Plist, and does the appropriate things if it finds one.

The pre-defined LISP function DEFUN is available for defining EXPRs, FEXPRs and MACROs.

```
Ex: (DEFUN CONC EXPR (LA LB)
         (COND
              ((NULL LA) LB)
              ((CONS (CAR LA) (CONC (CDR LA) LB)))
         )
    )

Now: (CONC '(XY Z) '(A B C)) = (XY Z A B C)
```

C. Applying LISP function - recursion

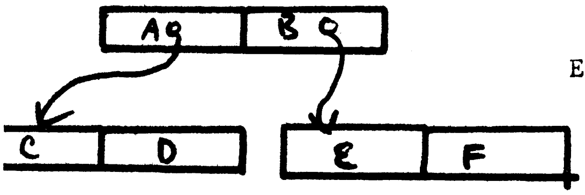
Case 1. A is a SUBR,LSUBR, or EXPR.
(EVAL '(A B C)) ⇒ First EVAL B
Then EVAL C
Use the results as arguments to function A

Case 2. A is FSUBR, FEXPR
(EVAL '(A B C)) ⇒ Just use (B C) as the argument to function A

Case 3. A is a MACRO
(EVAL '(A B C)) ⇒ Use (A B C) as the argument to function A.
When A returns a value, EVAL that as the
value of (A B C).

4. Garbage Collection

A. General outline of method.



0. Current-cell ← POP

Entry point. Current-cell is (A . B):
1. If current-cell is marked go to 0.
   Otherwise:
2. Mark current-cell
3. Push (CDR current-cell) [Address of (E . F)]
4. Current-cell ← (CAR current-cell) [(C . D]
5. Go to 1.

B. Problems
- a. Pname table versus list structure
- b. Atom header space versus value property
- c. Concept of a T.W.A

If time permits . . .

4. Some special implementation details
    A. Special system properties
    B. Atom pointers
    C. Binding conventions
    D. Argument passing

5. Running Compiled code in LISP
    A. Special versus local variables
    B. Stack manipulation

6. The FUNARG problem and free variables in LISP

Example of a LISP macro:

```
(DEFUN CDDR MACRO (X) (LIST 'CDR (LIST 'CDR (LIST 'CDR (LIST 'QUOTE X)))))
```

Now lets EVAL (CDDR X Y Z Q)
    Step 1. Apply the function CDDR as defined to the list (CDDR X Y Z Q).
            This yields the list (CDR (CDR (CDR (QUOTE (CDDR X Y Z Q))))).

    Step 2. EVAL this result and get (Z Q).

III.C.1

Jim Henriksen

. History of Simscript

A. SIMSCRIPT-I, SIMSCRIPT 1.5

1. Developed at Rand Corp. in early 60's
2. Ran on 2nd generation hardware of several manufacturers
3. Featured translation of Simscript into Fortran-II
4. designed for simulation only

B. Simscript-II

1. Developed at Rand in late 60's
2. Runs on IBM 360
3. Features translation into 360 assembly language
4. Designed as a fairly general high level language
5. Implementation incomplete

C. Simscript II Plus

1. Program product of Simulation Associates
2. Costly

D. Simscript 2.5

1. Program product of C.A.C.I.
2. Costly

. General Program Structure

A. PREAMBLE - defines global variables
B. MAIN program
C. Subroutines

I. Variable modes

A. INTEGER
B. REAL
C. TEXT

. Variable Types

A. Scalars
B. Arrays

1. Dimensionality declared at compile time:

```
DEFINE X AS A 2-DIMENSIONAL ARRAY
DEFINE Y AS A 1-DIMENSIONAL ARRAY
```

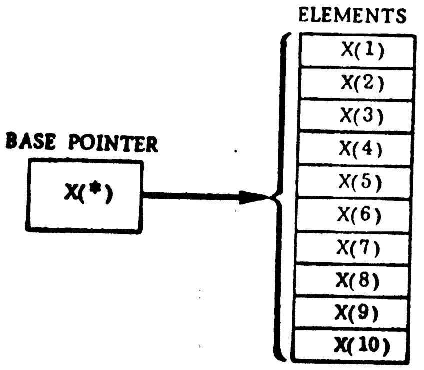
2. Storage allocated, a la PL/I "BASED" variables, at run time:

```
RESERVE X(*,*) AS 10 BY J
RESERVE Y(*) AS N
```

3. Internal storage mode uses pointers, pointer vectors:

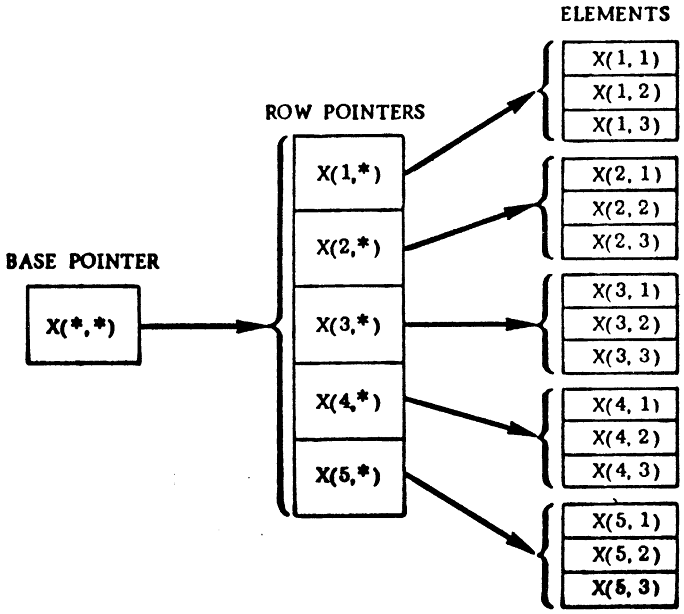
RESERVE X(*) AS 10

Stored as:



RESERVE X(*,*) AS 5 BY 3

stored as:

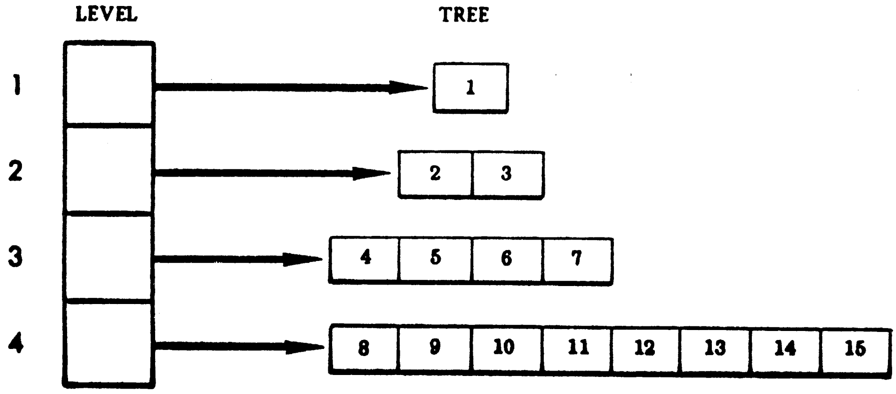


4. Notation, although clumsy, exists to manipulate pointers:

```
PREAMBLE NORMALLY, MODE IS INTEGER
DEFINE LEVEL AND TREE AS 1-DIMENSIONAL ARRAYS
END

READ N   RESERVE LEVEL(*) AS N
FOR I=1 TO N,
DO
    RESERVE TREE(*) AS 2**(I-1)  READ TREE
    LET LEVEL(I)=TREE(*) LET TREE(*)=0
LOOP
END
```

For N=4, the memory structure at the end of program execution looks as follows:

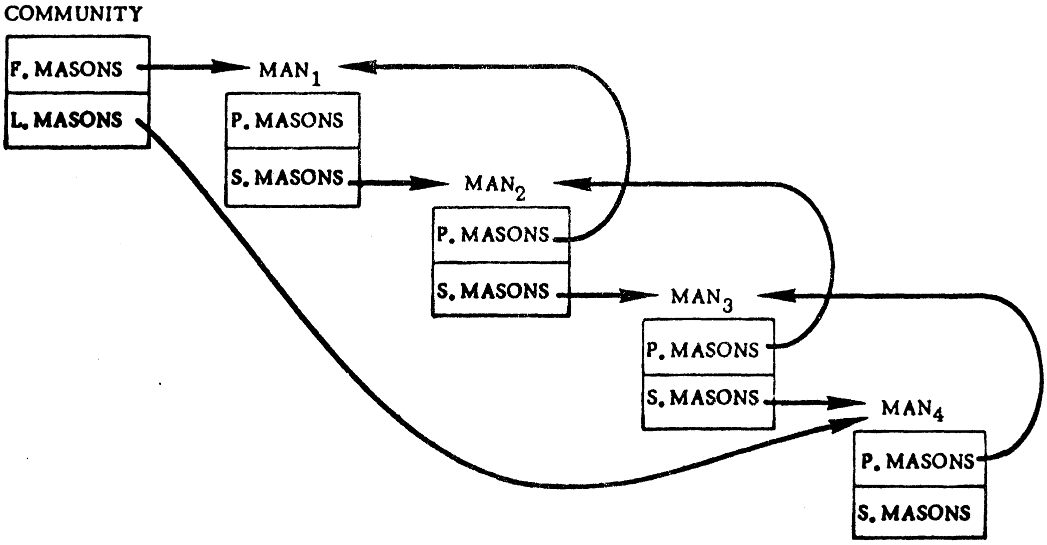


C. Entities and Sets

1. Entities have declared attributes and declared [potential] set memberships and ownerships:

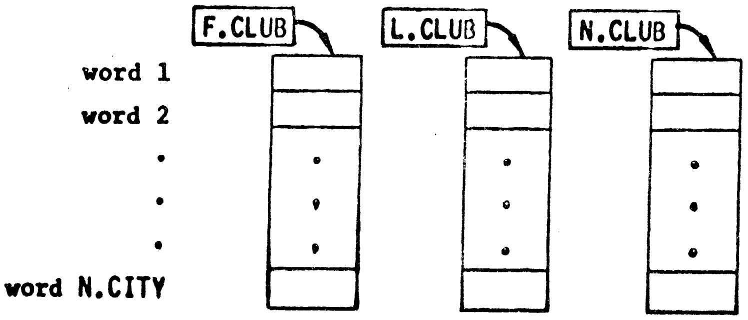
```
EVERY MAN HAS A NAME,AND AN ADDRESS,OWNS SOME
    CHILDREN AND MAY BELONG TO THE MASONS,A CHURCH,
    A FAMILY AND AN ALUMNI.CLUB
EVERY X HAS A P,A Q,A Z AND AN A
EVERY PROGRAM HAS AN ENTRY,OWNS SOME LABELS,
   BELONGS TO A PREAMBLE AND HAS A LENGTH
```

2. Set membership implemented by a straightforward system of implicitly allocated pointers:



3. Entity storage allocation

a. Permanent entities stored as arrays:



b. Temporary entities stored as based structures

4. Entity field and bit packing:

Declaration:

EVERY MAN HAS AN AGE(1/4), A NAME AND A SEX(1/4)

Entity record:

word 1	AGE	
word 2	NAME	
word 3	SEX	

Declaration:

EVERY MAN HAS AN (AGE(1-8) AND NAME(9-32))

Entity record:

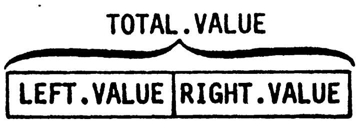
	1	9 32
word 1	AGE	NAME

Declaration:

EVERY PART HAS A (RIGHT.VALUE(2/2),LEFT.VALUE(1/2),TOTAL.VALUE)

Entity record:

word 1



Declaration:

EVERY MAN HAS AN (AGE(1/4),NAME(2/4),WEIGHT(17-32))
AND OWNS A FAMILY

Entity record:

word 1	AGE	NAME	WEIGHT
word 2	F.FAMILY		
word 3	L.FAMILY		

Declaration:

EVERY MAN HAS AN AGE(1/4),OWNS A FAMILY,HAS A (NAME(2/4)
AND WEIGHT(2/2))

Entity record:

word 1	AGE		
word 2	F.FAMILY		
word 3	L.FAMILY		
word 4		NAME	WEIGHT

5. Special purpose subroutines constructed to carry out membership functions of each set

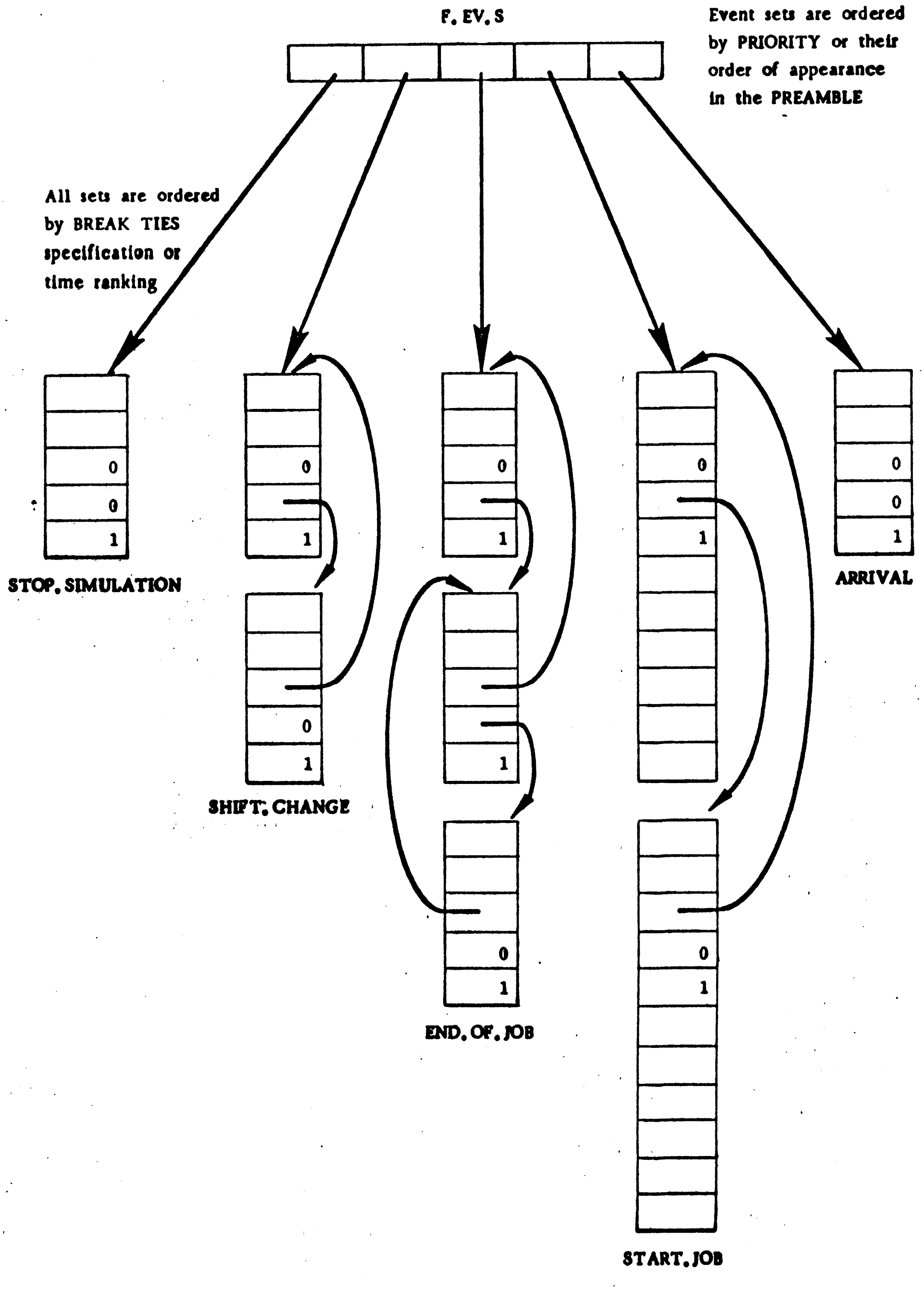
D. Event notices
  1. Two types
    a. Internal
    b. Exogenous - read from a numbered I/O unit
  2. Look like a temporary entity
    a. Has implicit attributes TIME.A, the simulated time of the event, and EUNIT.A, the unit from which the event notice was read
    b. Has implicit membership in the set EV.S

```
PREAMBLE
EVENT NOTICES INCLUDE ARRIVAL, WEEKLY.REPORT AND END.SIM
EVERY JOB.OVER HAS A NEXT.JOB AND OWNS SOME RESOURCES
```

When created, records for these event notices look like

	ARRIVAL	WEEKLY.REPORT	END.SIM	JOB.OVER
word 1	TIME.A	TIME.A	TIME.A	TIME.A
word 2	EUNIT.A	EUNIT.A	EUNIT.A	EUNIT.A
word 3	P.EV.S	P.EV.S	P.EV.S	P.EV.S
word 4	S.EV.S	S.EV.S	S.EV.S	S.EV.S
word 5	M.EV.S	M.EV.S	M.EV.S	M.EV.S
				NEXT.JOB
				F.RESOURCES
				L.RESOURCES
				N.RESOURCES

3. Priority of event notices implemented via index into subscripted set EV.S
    a. Default priority of event notices is in order of occurrence in declarations.
    b. Default overridden by "PRIORITY ORDER IS" statement


F.EV.S
Event sets are ordered by PRIORITY or their order of appearance in the PREAMBLE
All sets are ordered by BREAK TIES specification or time ranking
0
0
1
STOP.SIMULATION
0
1
0
1
SHIFT.CHANGE
0
1
1
0
1
END.OF.JOB
0
1
0
1
START.JOB
0
0
1
ARRIVAL

4. Event notice creation
   a. Created automatically for exogenous events:
      1. At initiallization (START SIMULATION)
      2. At exit from event routine
   b. Created under program control for internal events by use of the "SCHEDULE" statement

7. Control Structure - a recursive calling protocol

```
1    Rname   CSECT
2            USING *,15
3            USING H,7,8,9
4            B     F
5            DC    AL2(L)
6            DC    AL2(0)
7            DC    AL2(X)
8            DC    AL2(T-*+10)
9    F       L     2,0(1)
10           DROP  15
11           BALR  0,2
12   H       EQU   *
```

Fig. 1--Prologue format

BALR instruction (line 11) passes control to XREC, a SIM-
I system routine that establishes a *save area*, and returns
ess of the save area's first byte in general register 6. The
a, illustrated in Fig. 2, is divided into four sections.

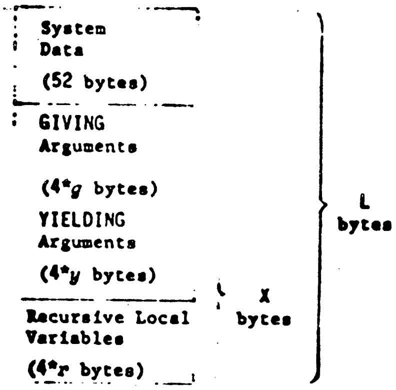


Fig. 2--Save area format

first section, labeled System Data, contains such items as
n point of the calling routine. It should not be used by
ammer. The second section, labeled GIVING Arguments, con-
values of the routine's giving arguments. The third section, labeled YIELDING Arguments, is the area in which yielding argument *values* are stored prior to return to the calling routine. The fourth section, labeled Recursive Local Variables, is the area in which all recursive local variables of the routine are located. When a routine is called, XREC zeroes out the yielding and recursive areas.

THE BODY

The routine body must conform to several conventions concerning:

(a) accessing GIVING argument values
(b) returning YIELDING argument values
(c) accessing recursive local variables
(d) using registers.

GIVING Arguments

General register 6 points to the first byte of the save area. To access the value of the first GIVING argument, the programmer must refer to the first four-byte word after the System Data section of the save area, i.e., the word whose address is 52 bytes greater than the contents of general register 6. The first GIVING argument is at 52(6), the second at 56(6), etc. The i-th giving argument is addressed as [52 + 4*(i - 1)](6).

As an example consider the calling sequence:

CALL GET.DATA GIVEN I,J AND K YIELDING M AND N

The value of I is stored at 52(6), the value of J at 56(6), and the value of K at 60(6).

YIELDING Arguments

The first YIELDING argument follows the last GIVING argument; in general, the address of the i-th YIELDING argument value is [52 + 4*g + 4*(i - 1)](6). In the above example, the value that will be stored in M when control returns to the calling routine is at 64(6) The value that will be stored in N is at 68(6).

I. Storage Allocation

A. Permanent entities, arrays - individual GETMAIN's
B. Temporary entities
   1. Check pool of desired size
   2. GETMAIN if none in pool
   3. Released blocks returned to pools and "branded."
   4. Continues until no more storage available, then FREEMAIN of all free blocks done

String and List Processing Seminar
Thursday, June 15, 1972
G. Lift

Snobol Internals

Data Organization

resident data - system things

allocated data - handled by storage management routines
all data created by source language programs

Data Representation

descriptor (Snobol's unit of storage)

used for all source data objects (often several descriptors together)

structure

T	F	V

T field: data type code (usually)

F field: various flags eg., A flag means V field contains pointer to allocated data region

V field: datum, or pointer to structure for datum

simple examples: integer 5

I	0	5

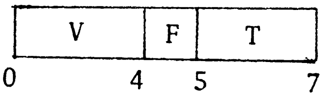
pattern

P	A	

I is the type code for integer, P for pattern

internal objects often have other arrangements

S/360 implementation: 2 words



qualifier (for character strings) - 2 descriptors

T	F	V	0	L

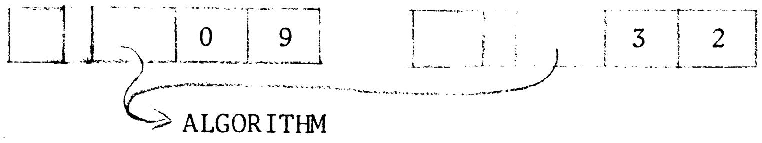
V: "base" pointer to string data

0: displacement of beginning of string from V pointer

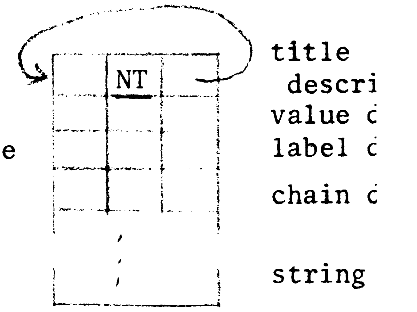
L: string length

T,F as before

example: string ALGORITHM string OR



Data Structures



natural variables - in allocated data region

title descriptor: T field - length variable name

F field: T flag - title descriptor

N flag - natural variable

value descriptor: value of variable, or

pointer to structure

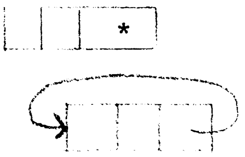
label & chain descriptors (see below)

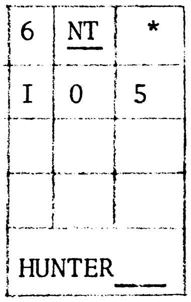
string (var. name) stored in consecutive descriptors following

chain descriptor, padded with blanks

example HUNTER = 5

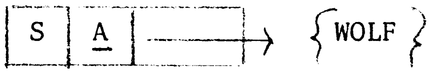
note
means



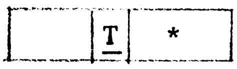


written {HUNTER}

HUNTER = 'WOLF' changes value pointer to



other data types (patterns, arrays ...) - represented with blocks of descriptors, always beginning with a title descriptor:



the T field contains the number of descriptors in the block, not counting the title descriptor

arrays: block contains a descriptor for each array entry, a descriptor for each dimension containing the lower bound and extent of that dimension, and header information (# dimensions, ...)
all entries are allocated at definition

tables: block contains two descriptors for each table entry (since an entry is an ordered pair), and header information

user defined data types: block contains a descriptor for each field, and two descriptors for header information

patterns: see later

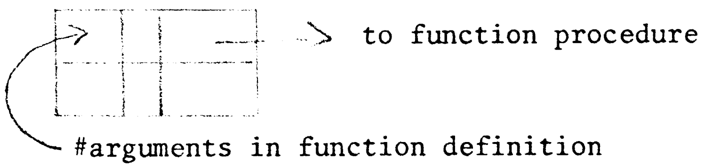
statement labels: natural variable structure with label in 'string field' has pointer to code in label descriptor

code: sequence of function and operand descriptors

function descriptors: T field: #arguments in call
F field: F "function" flag
V field: pointer to link descriptor pair for function

operand descriptors: same form as source-language data

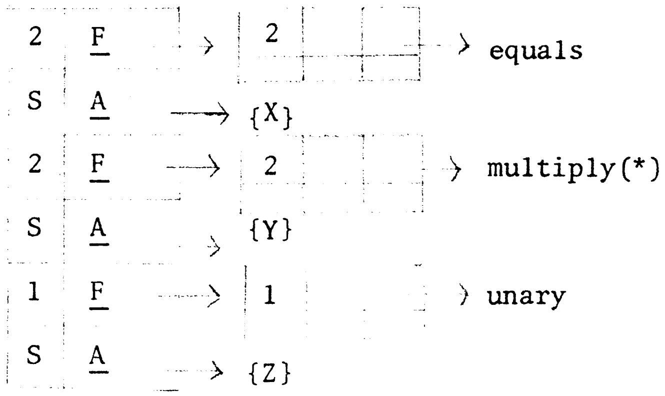
link descriptor pair:



second descriptor used only for special functions e.g., user defined

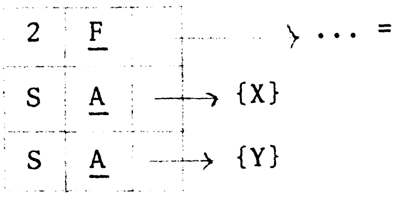
code is arranged in a prefix code format, with each function descriptor followd by its arguments

example: X = Y * -SIZE(Z)

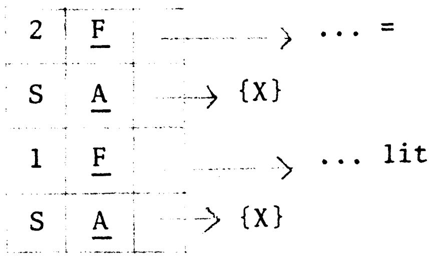


several function descriptors may point to the same link descriptor pair, i.e. there is usually one pair per function. SNOBOL knows the location of all existing link descriptor pairs
executing OPSYN alters the pointer in the link descriptor pair

literals in code are handled by a procedure 'lit' with one argument example X = Y :



X = 'Y':



gotos are handled by a goto function whose argument is a statement label

Strings and Variables

when string created by any source language operation, natural variable structure used to represent it

only one copy of any string in allocated data (no 'sharing' of substrings, though)

example:

after executing

X = 'AB' 'C'

Y = 'A' 'BC'

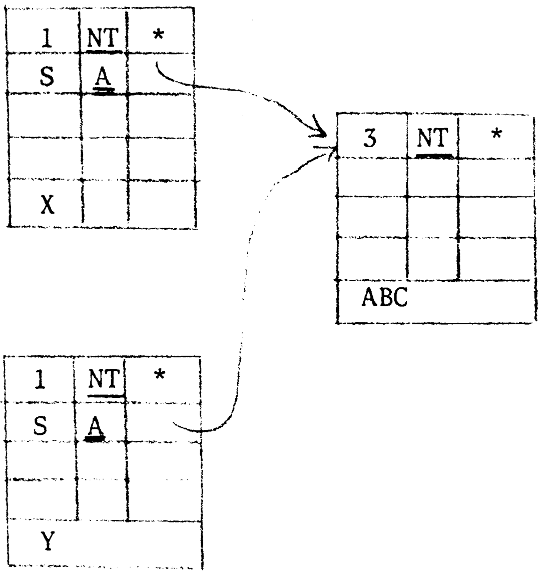


table of natural variables - each string in hash code table

hash coding C (0≤C≤255) hashes into table of 256 'chains'; string put on that chain

hash coding N - chain ordered by increasing N

both hashes are on char. string

V field of chain descriptor has pointer to next on chain (last 0)

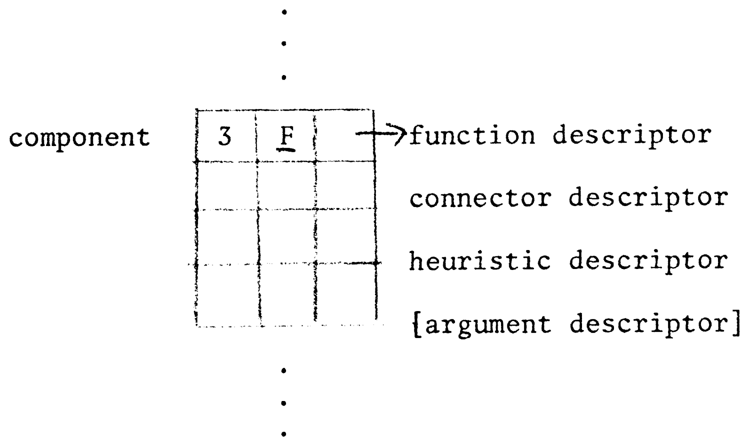
T field of chain descriptor contains N

access for variable (or string) involves search to determine if already present & creation if not

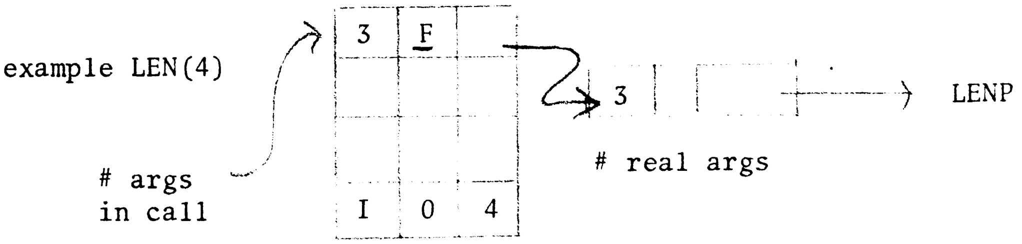
Patterns

construction

each component corresponds to 1 matching operation



if the matching function has no argument, there is no argument descriptor, and the T field of the function descriptor contains 2



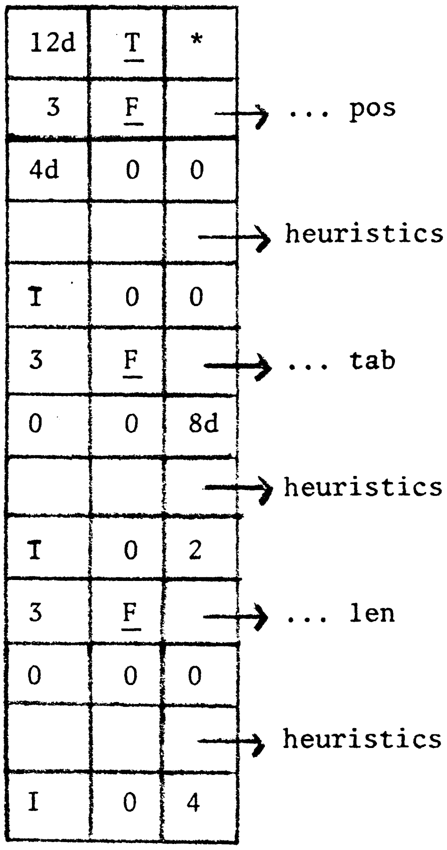
connector descriptor V field: offset of alternative (0:none)

T field: offset of subsequent (0:none)

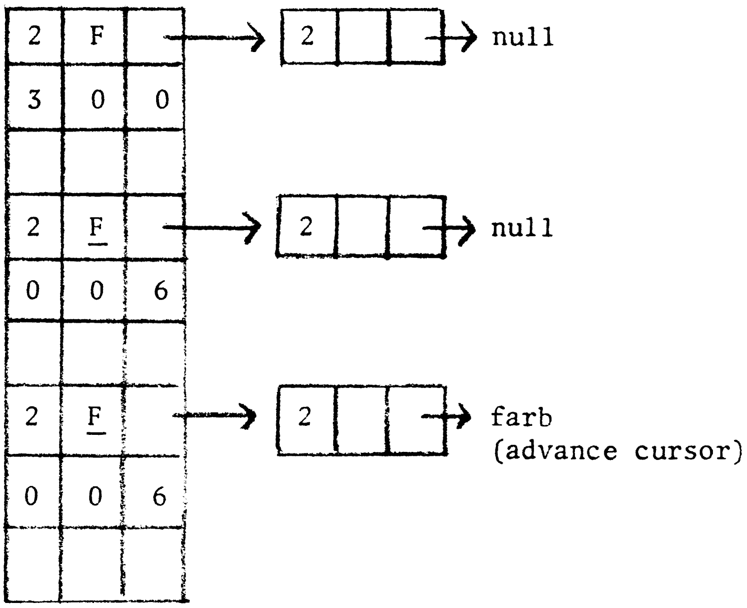
offsets are from beginning of pattern

note the number of arguments includes alternate/subsequent and heuristic example pattern

POS(0)(TAB(2) | LEN(4))



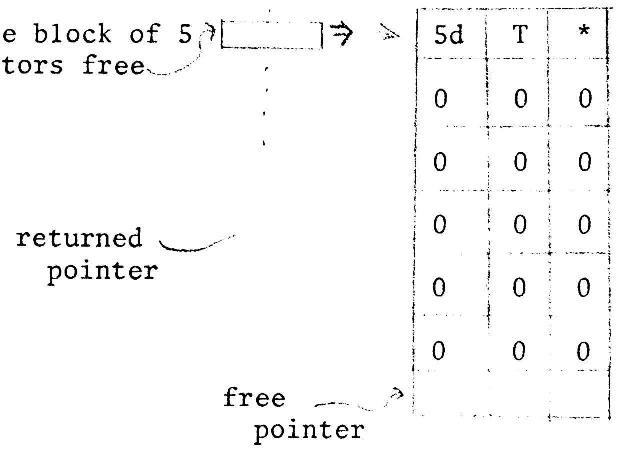
example pattern component for ARB



Storage Management

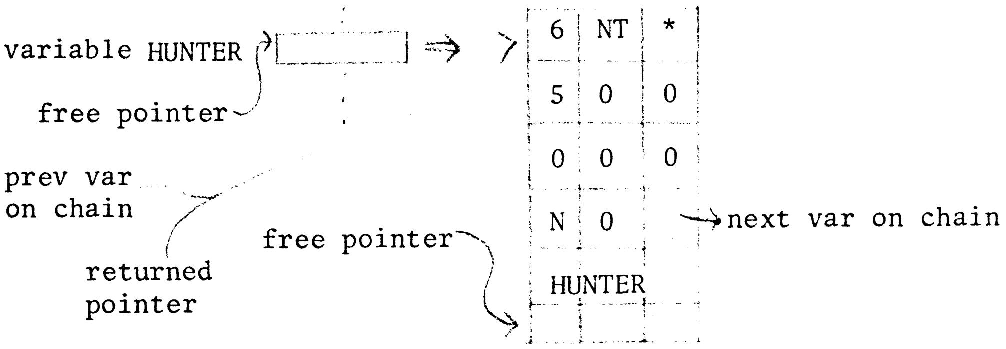
allocation: allocate from free pointer, move free pointer down thru allocated region

ex: allocate block of 5 descriptors free



note that the descriptors in the block are zeroed

ex: allocate natural (new)



note some fields are zeroed, and the variable placed on the appropriate chain garbage collection - useful objects are marked, then compacted into top of allocated area

- mark - M flag turned on in title block of object

- marking algorithm MARK - recursive procedure

  - begin with basic blocks (resident data); at first flag encountered, find title block of object pointed to

    - case 1: no mark on this title - mark this new block; push pointer to old block on SYSSTK; call MARK with new block

    - case 2: marked title - ignore (this does not go thru the chains of natural variables)

  - then mark any natural var. with non null label descriptor or non null value

  - relocation of useful objects -

    - phase 1: linear pass thru allocated area - change V pointers of marked title descriptors to new value

    - phase 2: adjust all pointers (on linear pass) of basic blocks and marked objects (using new title V fields)

    - phase 3: relocate & unmark (on linear pass)
remove useless nat. var. from chains

References

not much available

Griswold, RE. Snobol4 - Structure & Implementation
Share XXXVII; 12 Aug 1971.

__________, Macro Implementation of Snobol4
WH Freeman, in press.

String & List Processing Seminar
Thursday, June 22, 1972
Jim Hamilton

MTS-UMMPS Storage Allocation and Selected Applications

I. General Requirements

A. Must be completely general, i.e. must provide variable size blocks

B. Since storage allocation structures exist for as long as the system is up, storage must never be permanently "lost" due to causes such as fragmentation. Hence most structures are maintained in increasing location order.

C. Must obviously be application independent, hence such things as garbage collection, reclaimation, compaction, etc. are impossible. The only relocation possible is that provided by the DAT hardware, hence the mechanisms will often be page-oriented.

II. Storage Allocation for Supervisor Subroutines

A. Requirements and properties

1. Speed - must be very fast, for commonly used block sizes (e.g. PCBs) because of heavy usage

2. Must never run out of space, since the system will crash if this happens; paging, plus some care in coding, avoid this problem

3. Supervisor code is "dependable", so little error checking need be done.

4. Storage demands are, in a sense, fixed, since the supervisor itself is a closed system (requests from tasks are considered separately in the next section)

B. Pools

1. For very fastest allocation of fixed size (8 byte) entries for

a. CPU Queue
b. WAYT Queue
c. I/O Queue

2. Separate, pre-allocated areas with space for 255 of each type

3. Free space is simple linked list, done with offsets

4. Use of pool index allows "pointers" to fit in one byte

C. The Page Chain

With the exception of the pools, all dynamically allocated storage is taken from, and occasionally returned to, a page chain which is just a simple linked list of available pages. The page chain is constructed at initialization. All other available storage structures are initially empty.

The Supervisor never deals with blocks of real core larger than a page.

D. GRAB-FREE Subroutines

1. For every block size less than SVCASPEC (currently 96 bytes) there is a special chain containing blocks of the corresponding size. Calls to FREE always return small blocks to these chains, which are kept in LIFO order. Calls to GRAB will take a block from the proper chain if it is non-empty; otherwise it is allocated as described in 2.

2. There exist two chains of arbitrary size blocks, maintained in increasing location order to allow recombination. One is for blocks smaller than SVCABIG (currently 1024 bytes) and the other for larger ones.

   All blocks larger than SVCASPEC are returned to these chains, and blocks of any size less than a page are allocated from them, using a first fit algorithm, and splitting blocks when necessary.

3. If a block of the desired size or larger is not available, take a page from the page chain if there are any, and add it to the large or small chain according to size of the request, and try again.

4. If the page chain is empty, begin moving blocks from the special chains for small blocks to the arbitrary size chains, and continue until a large enough block has been found, or until all are moved. The latter case is followed by a superdump.

5. An example storage layout is shown in Figure 1.

III. Storage Allocation at the Task Interface

A. SVC GETBUF, GETSEGX, and FREEBUF

1. Absolute tasks (GETBUF):

   a. Maximum of 4 buffers allowed
   b. Maximum size one page
   c. Does some error checking, then calls GRAB or FREE
   d. The address and length of the allocation are recorded in a 32 byte table pointed to from the job table (4 buffers X 8 bytes = 32 bytes). This is done so that the storage can be recovered if the task goes west.

2. Relocatable Tasks

   a. GETBUF implies segment 4, GETSEGX specifies any of 3 through 8
   b. All requests rounded up to a page boundary
   c. For each task there is a PCB (Page Control Block) chain maintained in increasing virtual address order, one PCB per allocated virtual page.
   d. Supervisor scans PCBs until it finds a large enough hole in the desired segment, then GRABs the required number of PCBs (24 bytes each), initializes them and inserts them in the chain, it never explicitly allocates real core. It implicitly references the first page, however, into which it stores the length of the allocation.
   e. For FREEBUF the PCB chain is scanned for the PCBs describing the freed region. The real core pages, if any, are put back on the page chain, and the PCBs are removed from the task chain, and the Page Out Queue if necessary, and put on the Release Page Queue, where we will leave them for this discussion. Actually the FREEBUF routine is horrendously complex (more so than any other routine described here) but most of its complexity is irrelevant to us.

B. SVC GETSC, FREESC

1. For absolute tasks only.

2. These go directly to GRAB and FREE after checking to be sure there is at least one page on the page chain.

C. SVC GETRP , FREERP

1. Used only by the PDP to get or release pages of real core to be attached to PCBs.

2. GETRP just takes a page from the page chain if there are at least two pages there.

3. Otherwise it scans the small and large chains of supervisor pages and moves any full page blocks it finds over to the page chain. If it found any, it tries again. Otherwise it gives up, and the PDP will try again later.

4. FREERP essentially just adds pages to the page chain, unless they have been reclaimed or released.

IV. Storage Allocation in MTS (GETSPACE/FREESPAC)

A. General Requirements and Properties

1. Since this is the user interface, thorough error checking must be done.
2. It must be possible to recover the storage which has been allocated, in case user programs or various levels of system programs go west, or even when they terminate normally.
3. Storage management structures are maintained in system level storage, separate from the storage being managed; there are two reasons:
    a. It is undesirable to reference VM pages before they are needed.
    b. The user cannot be trusted to confine his references to storage which is allocated to him.
4. The VM integral must be computed, to keep the accounting people happy.

B. Buffer Control Blocks (BCBs)

The storage allocation structure is composed of fixed size (16 byte) buffer control blocks. These too must be allocated and freed. This is done using a conventional LIFO free space list. One page of BCBs is allocated initially (via SVC GETSEGX) and more are allocated if necessary, a page at a time. All BCBs are in segment 4.

C. Storage Management Structures

1. One may wish to look at Figure 2 while reading the following.
2. Primary buffers: Storage requests obtained from the supervisor via SVC GETSEGX are called primary buffers. Each primary buffer is described by two primary BCBs (PBCB1 and PBCB2) and one or more sub-buffer BCBs (SBCBs). VMI accounting is done at the primary buffer level.
3. Each primary buffer may be divided into one or more sub-buffers. Each sub-buffer is described by one SBCB. The SBCB also describes the free block following the allocated block.
4. The PBCB1 blocks are linked in increasing location order, and each points to a PBCB2 block, which points to a list of SBCBs, which are also in increasing location order. There is a separate list of PBCB1 blocks for each segment.

D. Detailed BCB Definitions

1. PBCB1 (all entries are fullwords)
    a. length of longest free block in buffer
    b. location of first byte past end of buffer
    c. link to next PBCB1
    d. link to PBCB2 for this buffer

2. PBCB2

   a. number of sub-buffers (2 bytes)
   b. length of buffer in pages (2 bytes)
   c. location of first byte in buffer (4 b ytes)
   d. number of free bytes before first sub-buffer (4 bytes)
   e. link to first SBCB

3. SBCB

   a. storage index number (1 byte) (read volume 5 to learn about these)
   b. length of allocated block (3 bytes)
   c. location of first byte after block (4 bytes)
   d. length of free block following (4 bytes)
   e. link to next SBCB

E. GETSPACE Algorithm

1. Search PBCB1 list for desired segment, looking for first one which has a free block equal to or longer than desired. If not found, go to step 4.

2. Search the SBCB list (including PBCB2) for the first free block equal to or longer than desired. There must be one, or we blew it. Insert a new SBCB after it to describe the new allocation and any remaining free block.

3. If the free block found in 2 is equal in size to the largest free block recorded in PBCB1, the SBCBs must be searched again to find the new largest free size. Then we are done - return.

4. Issue SVC GETSEGX; if this fails, go to step 5. Get and initialize a PBCB1, a PBCB2, and one SBCB, describing the requested allocation, plus any remaining space in the last page. Search the PBCB1s and insert the new one at the appropriate place. Compute the VMI and new total page count.

5. Must be insufficient space left in this segment. If a specific segment was requested, or if we have already tried segment 8, tell somebody about this problem. Otherwise, increment the segment number by one and go to step 1.

F. FREESPAC Algorithm

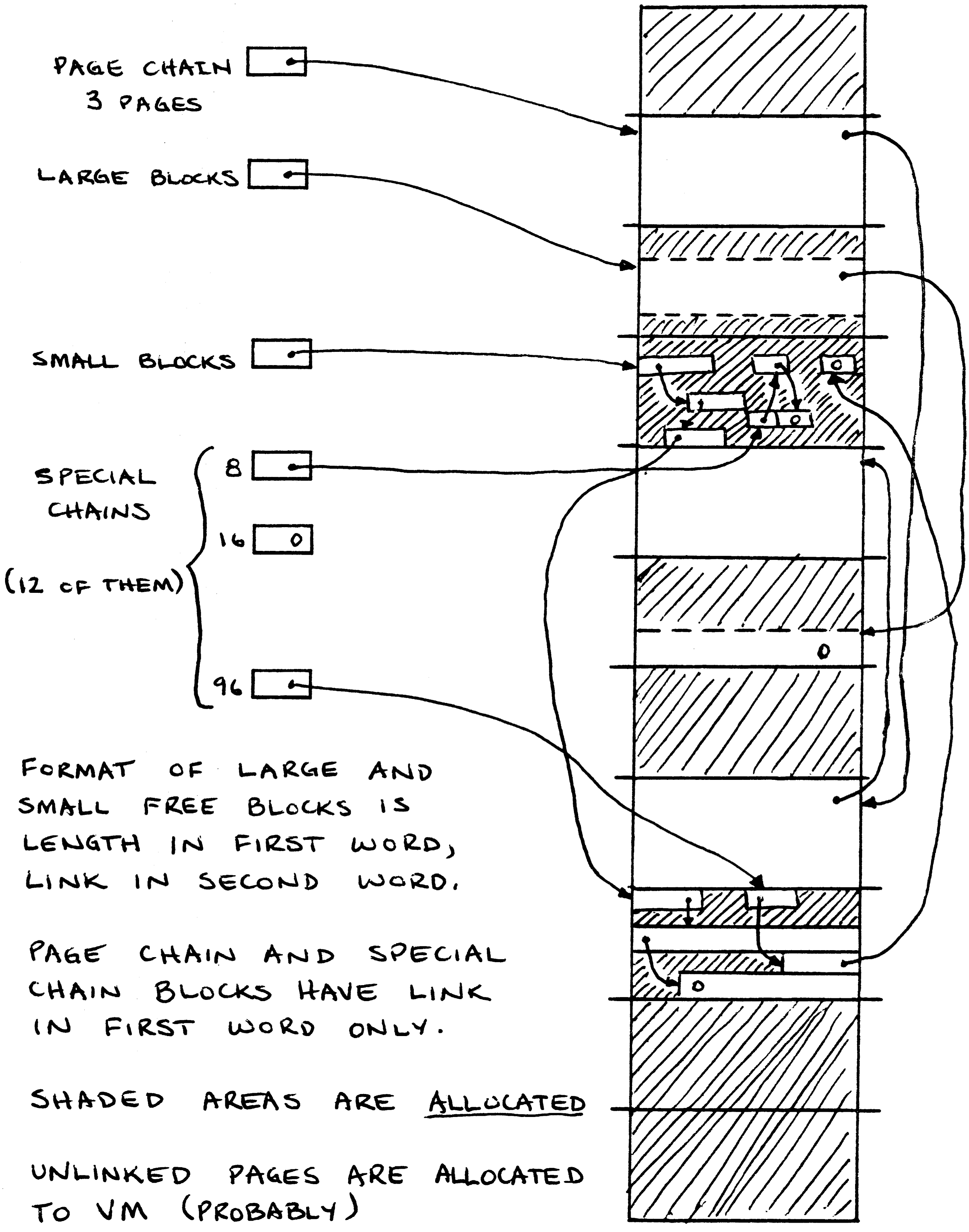
1. Find the SBCB whose allocated block completely contains the one being returned. This is a two step process, going down first the PBCB1s, then the SBCBs. It is a nono if its not there.

2. Compare the freed block to the allocated block. There are 4 cases.

   a. same - this is the normal case. Update field d. of the previous SBCB with the sum of itself and fields b. and d. of this SBCB. Unlink and free this SBCB. Update the maximum free block in PBCB1 if necessary. Decrement the subbuffer count (in PBCB2); if this is zero, the entire buffer is free, so issue SVC FREEBUF, unlink and free the PBCBs, and do VMI accounting. Return.

   b. Starting addresses the same. Update this and previous SBCB accordingly.
   c. Ending addresses the same. Update this SBCB accordingly.
   d. Neither address is the same. Get a new SBCB and insert it before the current SBCB. You should be able to figure out how to diddle the various fields.

3. Update the maximum free length in PBCB1 if necessary, and return.



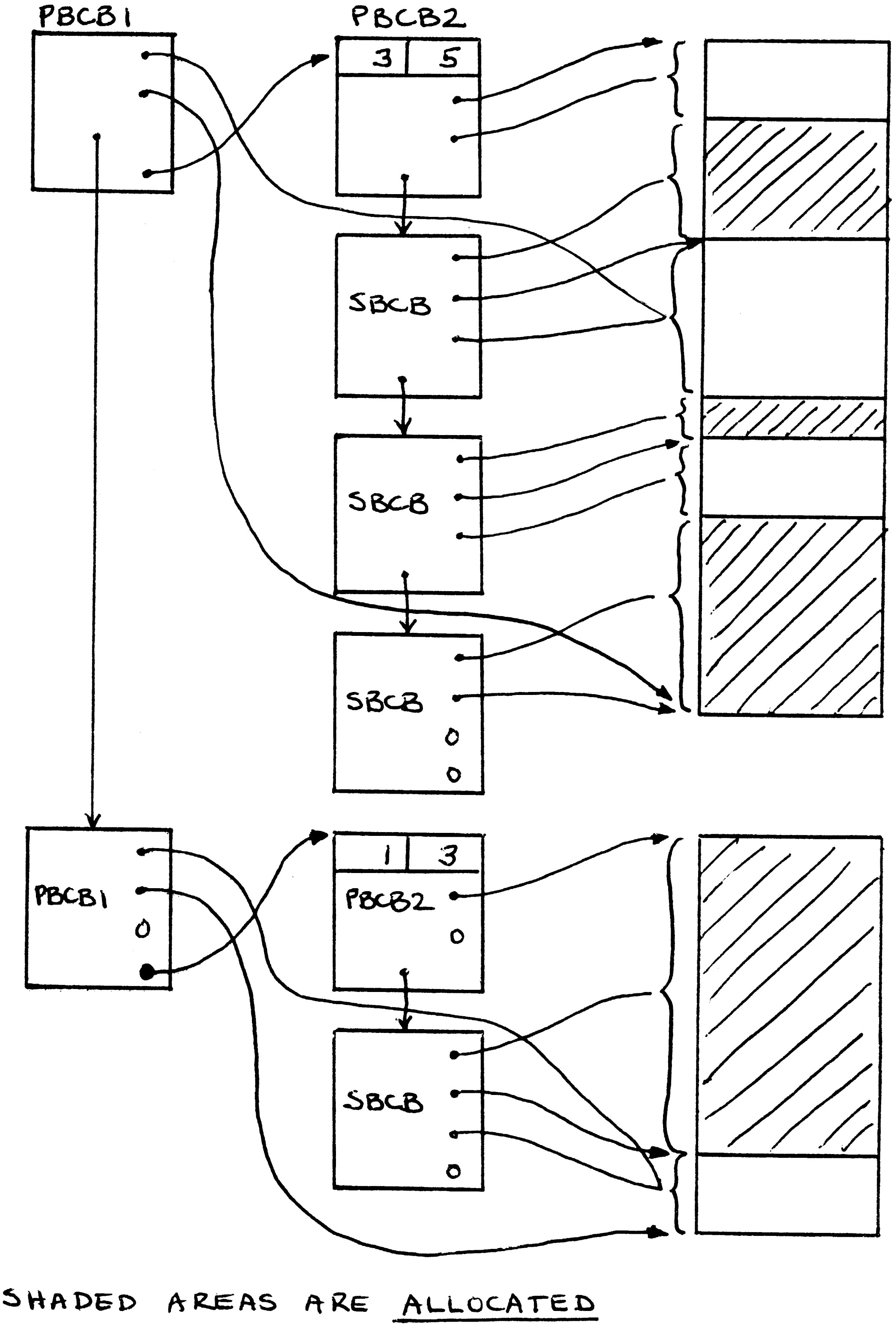
FORMAT OF LARGE AND SMALL FREE BLOCKS IS LENGTH IN FIRST WORD, LINK IN SECOND WORD.

PAGE CHAIN AND SPECIAL CHAIN BLOCKS HAVE LINK IN FIRST WORD ONLY.

SHADED AREAS ARE <u>ALLOCATED</u>

UNLINKED PAGES ARE ALLOCATED TO VM (PROBABLY)

Figure 1 - Supervisor Core Management



SHADED AREAS ARE <u>ALLOCATED</u>

FIGURE 2 - MTS STORAGE ALLOCATION

Jim Hamilton

UMMPS PAGING ALGORITHM

I. Before descending into the details of UMMPS and the PDP, it is probably instructive to say a few words about paging algorithms in general. They may differ in several important ways:

1. Demand paging vs. Anticipatory paging:Under demand paging, a system will move a page to main storage only when it is referenced. On the other hand, a system may attempt to anticipate the need for some pages, and page them in before they are referenced. Almost all current systems use demand paging.

2. The algorithm may be task oriented, or system oriented. That is, the decision as to which pages to move to and from main storage may depend heavily on the status of the task which owns them; this would be a task oriented algorithm. With a system oriented algorithm all pages in the system are treated identically, independent of their owners.

3. The replacement policy, for choosing pages to be removed from main storage, may vary considerably. This is probably the most important factor affecting the performance of paging systems. There are several possibilities discussed in the literature:

   a. FIFO - The oldest page in storage is chosen for removal. This is clearly not a very good choice, but early versions of UMMPS used it.

   b. Least Recently Used (LRU) - The page with the longest time since last reference is chosen for paging out. Note that this algorithm can only be approximated on the 360/67, and most other current processors.

   c. Working Set - The system keeps a record of recent references to pages by a task and attempts to keep a "working set" of pages belonging to a task in core. This is a task oriented policy, which attempts to estimate program behavior. It is said to be a nearly optimal realizable algorithm.

   d. A-Optimal - The page whose next reference is farthest in the future is chosen for removal. This is, in some sense, an optimal algorithm, but is unrealizable without knowledge of future page references. It is mainly a standard for comparison.

UMMPS uses basically a system oriented demand paging algorithm with an LRU replacement policy. The supervisor is totally responsible for these aspects of paging and their implementation is found in the GETWP SVC, which will be described in some detail later. The Paging Drum Processor (PDP) is responsible for the actual transfer of pages to and from the paging devices. In the remainder of these notes we describe 1) the UMMPS - PDP interface, 2) the PDP, 3) UMMPS (mainly GETWP), and 4) a day in the life of the average page.

II. The UMMPS - PDP Interface

A. Data Structures:

The primary data item containing information relevant to paging is the Page Control Block (PCB). A PCB is 24 bytes long and contains the following items:

1. Virtual address
2. Real Address
3. Pointer to owning TCB
4. Task page chain pointer
5. System Queue pointer
6. Reference Bit
7. Change Bit
8. External Address

plus several other items which don't concern us here. PCBs are created by the GETBUF SVC, and are released by the PDP.

There are four queues used by the system in managing paging. These are:

Page-In Queue (PIQ) - Pages to be brought into core from secondary storage

Page-In Complete Queue (PICQ) - Pages which have just been brought into core.

Page-Out Queue (POQ) - Pages which are in core, ordered approximately from least recently used to most recently used.

Release Page Queue (RPQ) - Pages which have been released (via FREEBF SVC).

B. Special SVCs

There are five special SVCs which are used only by the PDP. These are:

GETRP - Get real page - Used to get a real page of core to read into, for a page-in operation.

FREERC - Free real core - Used to release the core allocated to a page after it has been paged out.

GETWP - Get Write Pages - Removes a specified number of pages from the POQ, to be written out.

GETQS - Transfer the contents of the PIQ and RPQ to the PDP.

PDPWAIT - Tells UMMPS the PDP has nothing more to do. It will be restarted by a completion interrupt from any of its devices, or when UMMPS decides there is more for it to do (i.e., PIQ non-empty, or pages need to be written.

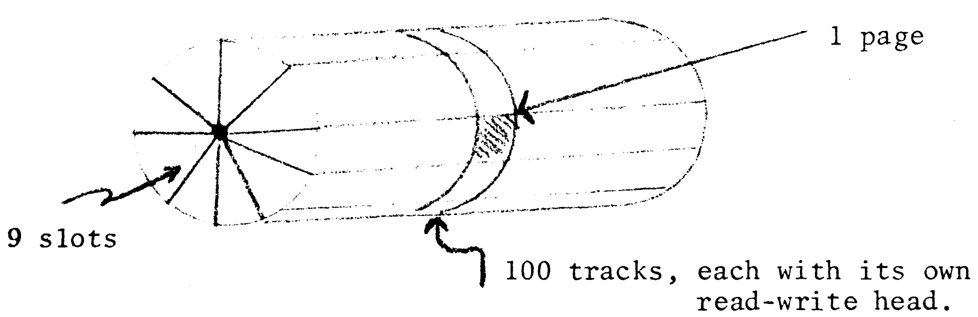
III. The Paging Drum Processor

A. Overview - It is the responsibility of the PDP to manage the paging devices, which currently include two drums and one disk. Each drum holds 900 pages, and the disk holds 6400 pages, for a total of 8200 pages. The worst case observed to date has been something over 4000 pages, and a typical heavy load is between 2000 and 2500 pages.

The disk is used only when the drums are nearly full, and at this time the PDP chooses pages on an LRU basis and moves them from drum to disk. This is called page migration.

The PDP consists of two asynchronous parts: the first builds channel programs and starts them via SVC STIO, and the second handles the completion interrupts and posts the completion of the paging operations.

B. In order to understand the operation of the PDP, it is necessary to understand the workings of the paging drum.



The picture shows the logical structure of the paging drum. Physically there are 200 tracks, with 4 1/2 pages per track, but the PDP treats it as shown, with the difference obscured by a trick in the channel programs.

The PDP constructs channel programs for all nine slots at a time. It will then chain these together if possible. The PDP is so designed that the drums can be kept running for an indefinite time with only one SIO, with reads in the appropriate slots, and with writes filling in the rest as necessary. Using this method, writes (i.e., page-outs) are essentially free.

C. PDP Data Structures - The PDP maintains a huge data block for each drum. Each such block contains, among other things:

1. 9 Local Page-in queues, one for each slot.
2. 3 channel program buffers
3. Spaces for 27 PCB pointers for the PCBs associated with the 3 possible channel programs.
4. A bit table describing available space on the drum; organized by slot.
5. 9 migration lists, one for each slot of PCBs ordered from least to most recently used. ("used" in this context means paged-in or paged-out.)

6. A local page-in complete queue

There is also a data block for the disk. Since the disk is managed on a page at a time basis, with no attempt at optimization, this data block contains only one local page-in queue, one channel program, one current PCB pointer, and the bit table.

There is also a "global" local page-in complete queue, on which the local PICQs from each device are collected, and whose contents are occasionally transferred to the supervisor's PICQ.

D. The algorithm

1. Get the PIQ and RPQ via the GETQS SVC.

   1.1 For each PCB on the RPQ, release its external address, free its real core page via SVC FREERC, if there is one, and free the PCB, via FREESC SVC (Free Supervisor Core).

   1.2 For each PCB on the PIQ, add it to the end of the local page-in queue for the appropriate slot on the appropriate drum, or to the LPIQ for the disk. This process is called slot sorting. If the PCB has no external address, put it on the local PICQ now, since it must be a new page.

2. For each drum do the following:

   2.1 Allocate a new channel program buffer if possible. If not, go try the next drum.

   2.2 For each slot: if the LPIQ for the slot is non-empty, remove the top PCB, get a real page via SVC GETRP if possible, and construct a CCW to read the page in. If no core is available, go to step 2.3 immediately.

   2.3 If there are slots available which don't contain reads, check drum availability; if there are less than MIGTH pages left on all drums, and if no migration is currently in progress, take a page from the top of the migration list for one of the available slots, and construct a CCW to read it into a page of supervisor core. Remember that a migration read has been started. MIGTH, the migration threshold, is currently set to 50 pages per drum, or a total of 100 pages, with 2 drums. This will probably be reduced in the future.

   2.4 If there are slots available which have no reads, and which have unused tracks for writing, issue SVC GETWP, requesting as many pages as there are available slots.

   2.5 For each PCB returned by GETWP, see if it has been changed. If not, and it's on the drum, just issue SVC FREERC. If it has been changed, or is on the disk, free its existing external copy, and construct a CCW to write it into one of the available slots.

   2.6 If there are still some available slots because of unchanged pages, issue another GETWP, and go to step 2.5.

2.7 If any reads or writes were set up in 2.2 through 2.6 above, package up the channel program and either issue SVC STIO if no channel program is currently running, or chain it to the end of the current channel program, with a TIC command.

This completes the setting up of channel programs for the drums.

3. For the disk, do the following:

3.1 If the disk is already running, do nothing.

3.2 If the local PIQ for the disk is non-empty, issue GETRP for a real page, and construct a CCW to read it in. Go to step 3.4.

3.3 If a migration read was set up in 2.3, then allocate a disk page and construct a CCW to write it out.

3.4 If anything was done in 3.2 or 3.3, complete the channel program but modify it so only the seek is done, then issue SVC SIO. This way the channel is not busy during the seek.

3.5 When the seek terminates, restart the channel program, doing the whole thing this time, unless we are doing a migration write, in which case we may have to wait for the completion of the read.

This completes the setting up of disk channel programs.

4. Collect the local page-in complete queues from the several devices and add them to the "global" local PICQ. If there are any new pages on this queue, issue a GETRP for them. If GETRP fails, keep these pages on this local PICQ, but put all completed pages on the supervisors PICQ. The supervisor will eventually find them there and restart the waiting tasks. If any channel programs were started above, go to step 1. Otherwise PDPWAIT. This completes part one of the PDP.

The remainder of the PDP consists of device-end and PCI (program controlled interrupt) interrupt routines. The PDP arranges to receive a task interrupt at the completion of any of its channel program buffers, i.e., once every logical drum revolution, from each drum. At such times it does the following steps:

5. For each PCB in the channel program just completed, do the following:

5.1 Add it to the bottom of the migration list for the appropriate slot, if this is a drum.

5.2 If it is a read operation, add the PCB to the local PICQ for this device. Go to step 5.4.

5.3 If it is a write operation, free the real core page, via FREERC.

5.4 Free the channel program buffer

5.5 If this was a device end, mark the status of the device as stopped.

5.6 Return and re-enable the interrupt.

This completes the description of the PDP algorithm. Many details of the actual implementation have been omitted for simplicity, but most of the important ideas are there.

IV. The Supervisor

In this section we discuss the algorithms for the five PDP SVCs, plus the subroutine TRANS, which is called to handle paging exceptions, among other things. All but TRANS and GETWP are, at least conceptually, simple, but all are mentioned briefly, for completeness.

PDPWAIT - Save the TCB pointer for the PDP (this is the only way the supervisor knows which task is the PDP). Remove it from the CPU queue. Save the restart address (a parameter in GRO)

GETQS - Lock the PAGQ lock, pass the PIQ and RPQ pointers to the PDP, set these pointers to zero, unlock, and return. This must be an SVC because only the supervisor can do the required locking.

There are several variables which control the page replacement policy, as implemented in the GETRP, FREERC, and GETWP SVCs. These are:

1. NFRPGS - Number of free pages available
2. MINFRPGS - Minimum number of free pages which must be maintained, currently = 1
3. NWRTPGS - Number of pages being written out
4. WRTFRPGS - The threshold for deciding when to write pages, currently = 15

GETRP - If the number of free pages is greater than or equal to MINFRPGS, remove a page from the free page chain and decrement NFRPGS. Otherwise indicate that no page is available.

FREERC - Add the page to the free page chain and increment NFRPGS.

GETWP - A little more complicated

1. If NFRPGS + NWRTPGS > WRTFRPGS, return zero pages.
2. For several reasons, the proper operation of GETWP requires that no CPU be relocatable. Therefore, if the other CPU is relocatable, a WRD instruction must be executed at this point, which causes an external interrupt to the other CPU. A flag is then set which will hold up the other CPU until GETWP finishes its work and resets the flag.
3. Starting at the top of the POQ, do the following for each PCB encountered, until either getting enough pages to fill the request, or until reaching the end of the POQ.

   3.1 Update the reference and change bits in the PCB with those in the storage keys for the real page.

   3.2 Set these bits in the storage keys to zero.

   3.3 If the page has not been referenced, add it to the list of those to be paged out. If it belongs to a non-privileged task, page it out anyway. If it has been referenced, reset the reference bit and move it to the end of the POQ.

3.4 If a page which has not been referenced belongs to a task which is running on the other CPU, don't page it out; instead leave it on the POQ.

3.5 Each page to be paged out is removed from its page table.

4. Update NWRTPGS, and return.

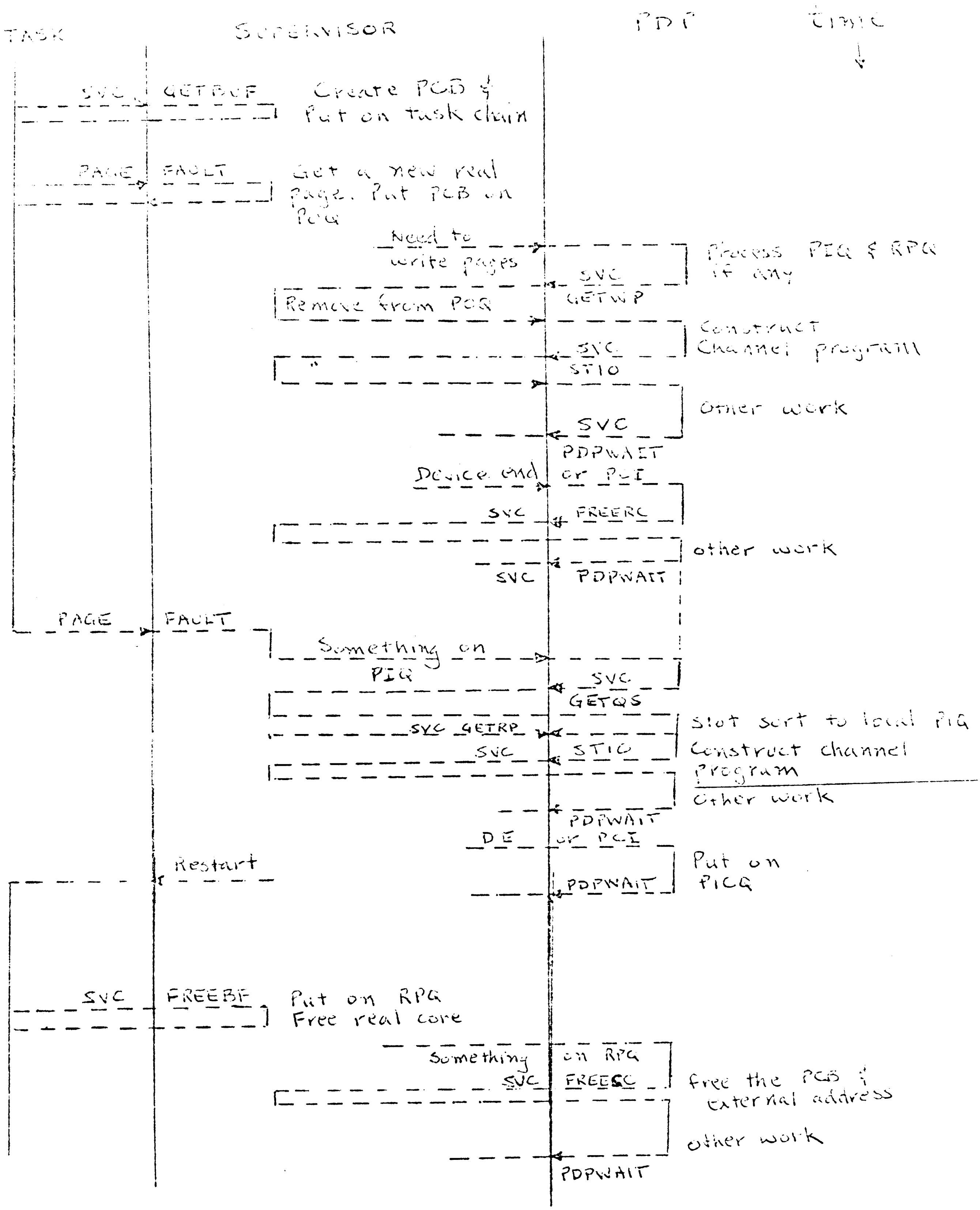
TRANS - A supervisor subroutine called by anything which needs to reference a virtual address, which means mainly paging exceptions, but includes other parts of the supervisor as well.

The algorithm for TRANS is:

1. Try an LRA instruction, if this works, return.
2. Search the task PCB chain, or shared PCB chain if segment 2. If not found, simulate program interrupt 5.
3. If the page is being paged in already, chain this request onto the previous and go schedule another task.
4. If the page is being paged-out, mark it as reclaimed, update the page-table, and return.
5. If the page has an external address, put the PCB on the PIQ, and schedule another task.
6. If the page has no external address, it must be new; try to get a real page for it. If successful, return, with the page table updated. Otherwise put it on the PIQ, and the PDP will retry.

V. A Day in the Life of the Average Page

The chart on the next page illustrates the various transitions which may be encountered by a page during its lifetime. Note: only those SVCs are shown which deal with the particular page we are watching.


TASK
SUPERVISOR
PDP
time
SVC GETBUF
Create PCB & Put on task chain
PAGE FAULT
Get a new real page. Put PCB on PCQ
Need to write pages
Process PIQ & RPQ if any
SVC GETWP
Remove from PCQ
Construct channel program
SVC STIO
"
Other work
SVC PDPWAIT
Device end or PCI
SVC FREERC
other work
SVC PDPWAIT
PAGE FAULT
Something on PIQ
SVC GETQS
Slot sort to local PIQ
SVC GETRP
SVC STIO
Construct channel program
Other work
PDPWAIT
DE or PCI
Put on PICQ
PDPWAIT
Restart
SVC FREEBF
Put on RPQ Free real core
Something on RPQ
SVC FREESC
Free the PCB & external address
other work
PDPWAIT

Jon Bauer
CCS 500
June 28, '72

IV.C.1

The Michigan Graphics Interpreter

In the few pages that follow, I'll describe the major features of the MGI routines and how they're implemented. A large portion of the paper will be devoted to describing the MGI routines setting the ground-work for explaining the implementation which is actually quite straightforward.

The MGI system consists of a set of Fortran-callable routines. They were written to be device independent and to that end, they generate no hardware graphics commands directly. Rather, they return graphical data which, in turn, can be used to drive hardware-dependent graphic routines to draw the images.

The MGI system was written to manipulate graphical objects called 'elements'. Routines perform three basic operations. Elements may be created, transformed, and recovered by calls to the appropriate routines.

The current MGI system handles three types of elements: 'parts', 'transforms', and 'assemblies'. A part is an element which defines a three-dimensional graphical entity in terms of points and lines. A transform is an element which can cause a part to be scaled, rotated, viewed in a perspective, and/or translated (moved). An assembly is an element which defines a heirarchical relationship between parts and subassemblies and transforms. A transform can be applied to parts and assemblies.

Parts

A part is completely defined by the following data...

1. A legal MGI name,
2. The number of points,
3. The homogeneous coordinates of each point, and
4. An attribute number for each point.

A call to the appropriate routine with the above information causes the part to be created.

A 'legal MGI name' is defined to be a string of non-blank characters, left justified to a full word boundary followed by a blank. Up to six non-blank characters may be specified. (MGI names are also used to name transforms and assemblies.)

Four homogeneous coordinates are used to define each point of a part. They can be written. . .

(HX,HY,HZ,H)

where the normal coordinates can be recovered by. . .

$$X=\frac{HX}{H} \quad Y=\frac{HY}{H} \quad Z=\frac{HZ}{H}$$

If we wish to express the normal coordinates (X,Y,Z) as homogeneous coordinates, we could of course write (X,Y,Z,1)

The reasons for representing points using homogeneous coordinates are. . .

1. Three dimensional transformations may be represented and performed very simply in this coordinate system, and
2. Points at infinity can be described using finite numbers.

The first and more important reason should be easier to understand after reading the next section on transforms.

Attribute numbers (one per point) are generally used to specify the type of lines to be drawn to the corresponding points. For example, zero might be used to indicate invisible lines and one to three may indicate different line intensities.

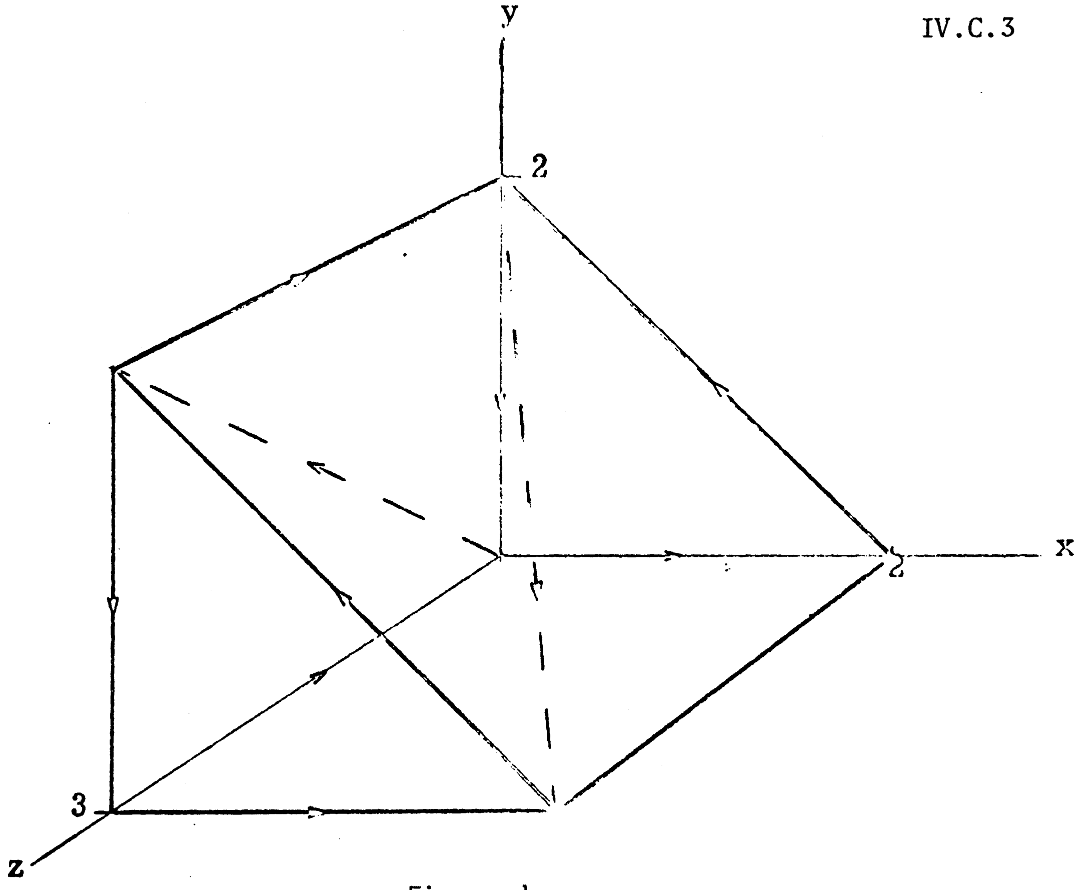


Figure 1

Using this convention, the shape shown in Fig. 1 could be defined by:

Point	X	Y	Z	Attribute
1	0	0	3	0
2	2	0	3	1
3	0	2	3	1
4	0	0	3	1
5	0	0	0	1
6	2	0	0	1
7	0	2	0	1
8	0	0	0	1
9	0	2	3	0
10	0	2	0	1
11	2	0	3	0
12	2	0	0	1

```
      REAL X(12)/0., 2., 3*0., 2., 4*0., 2*2./,
     &     Y(12)/2*0., 2., 3*0.,2., 0., 2*2., 2*0./,
     &     Z(12)/4*3., 4*0., 3., 0., 3., 0./
      INTEGER IAT(12)/0, 7*1, 0, 1, 0, 1/
C
      CALL MGINIT(50)
      CALL MGPT2('WEDGE ', 12, X, Y, Z, IAT, &800)
```

This example creates a Part named WEDGE which may now be manipulated by the MGI routines.

Transforms

A transform is defined by the following data. . .

1. A legal MGI name, and
2. A 4 x 4 transformation matrix.

We can also represent a set of points, for example a part, as a matrix. A set on n points can be represented by the matrix. . .

$$\begin{bmatrix} HX_1 & HY_1 & HZ_1 & H_1 \\ HX_2 & HY_2 & HZ_2 & H_2 \\ \cdot & & & \\ \cdot & & & \\ \cdot & & & \\ HX_n & HY_n & HZ_n & H_n \end{bmatrix}$$

Multiplying such a matrix (set of points) by a transformation yields a new matrix which represents the transformed set of points, eg. . .

$$\underset{\text{original part}}{\begin{bmatrix} HX_1 & HY_1 & HZ_1 & H_1 \\ HX_2 & HY_2 & HZ_2 & H_2 \\ \cdot & & & \\ \cdot & & & \\ \cdot & & & \\ HX_n & HY_n & HZ_n & H_n \end{bmatrix}} \times \underset{\text{transform}}{\begin{bmatrix} A & B & C & D \\ E & F & G & H \\ I & J & K & L \\ M & N & O & P \end{bmatrix}} \rightarrow \underset{\text{transformed part}}{\begin{bmatrix} HX'_1 & HY'_1 & HZ'_1 & H'_1 \\ HX'_2 & HY'_2 & HZ'_2 & H'_2 \\ \cdot & & & \\ \cdot & & & \\ \cdot & & & \\ HX'_n & HY'_n & HZ'_n & H'_n \end{bmatrix}}$$

Let's look at some particular matrices and what they do.

Identity

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Causes no change.

Independent Scale Changes

$$\begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Causes the x axis to be scaled by a factor of a.
Causes the y axis to be scaled by a factor of b.
Causes the z axis to be scaled by a factor of c.

Uniform Scale Change

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1/d \end{bmatrix}$$

Causes the part to be scaled by a factor of d.

Translation

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ e & f & g & 1 \end{bmatrix}$$

Causes the part to be shifted e units along the x axis, f units along the y axis, and g units along the z axis.

Rotation About x Axis

$$\begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Causes a part to be rotated about the x axis by an angle of θ.

Rotation About y Axis

$$\begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation About z Axis

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Projection

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1/v \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Causes a part to be distorted as if it were being viewed from a point along the z axis, v units from the origin.

Probably the nicest feature of this matrix representation of transformations (and the motivation behind using the homogeneous coordinate system) is that the above basic transformations can be combined in any amount and any order and the resulting more complex transformation can also be represented as one 4 x 4 matrix.

It's easy to show this. Suppose we wish to apply n transformations to a part (for example, we might want to move (translate) a part to the origin, rotate it about the x axis, rotate it about the y axis, enlarge (scale) it, and move it back to where we found it.) We could multiply each point in the part by each transformation matrix (in the proper order) to transform the part ie. . .

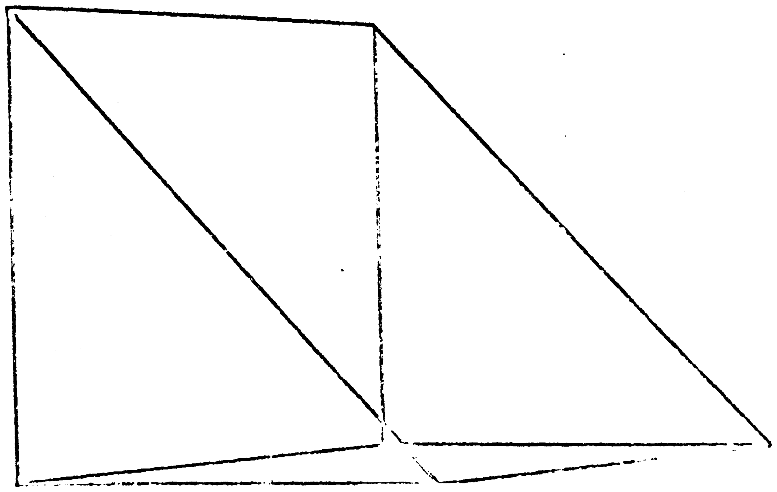
$$\left(\cdots\left(\left(\begin{bmatrix}\text{PART}\\ \text{MATRIX}\end{bmatrix}\times\begin{bmatrix}T_1\end{bmatrix}\right)\times\begin{bmatrix}T_2\end{bmatrix}\right)\times\cdots\times\begin{bmatrix}T_n\end{bmatrix}\right)\rightarrow\begin{bmatrix}\text{TRANSFORMED}\\ \text{PART}\\ \text{MATRIX}\end{bmatrix}$$

Or we could multiply all the transformation matrices first and apply the resulting transformation matrix to the part to be transformed, ie. . .

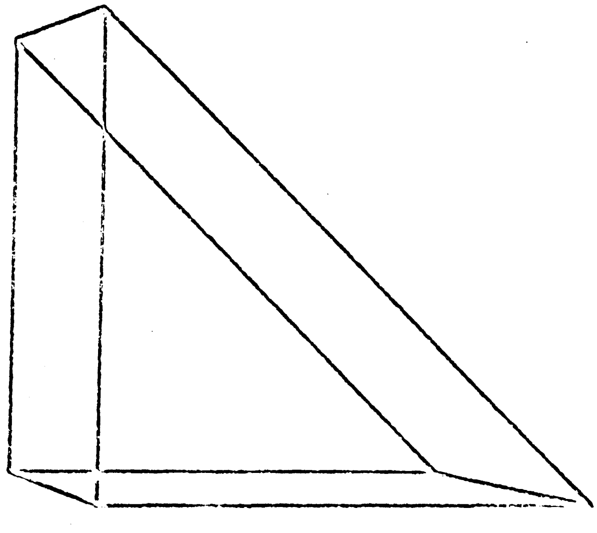
$$\begin{bmatrix}\text{PART}\\ \text{MATRIX}\end{bmatrix}\times\left(\cdots\left(\begin{bmatrix}T_1\end{bmatrix}\times\begin{bmatrix}T_2\end{bmatrix}\right)\times\cdots\times\begin{bmatrix}T_n\end{bmatrix}\right)\rightarrow\begin{bmatrix}\text{TRANSFORMED}\\ \text{PART}\\ \text{MATRIX}\end{bmatrix}$$

This buys us two things. a) Once we've computed the composite transformation matrix, the computation costs for transforming a part are reduced by a factor of n. b) We can represent any combination of the above simple transformations, no matter how complicated, as one 4 x 4 matrix saving storage costs and simplifying our implementation of assemblies which will be described in the next section.

```
C          EXAMPLE PROGRAM--PARTS & TRANSFØRMS.
C          TØ BE RUN ON A COMPUTEK GRAPHICS TERMINAL.
C
       REAL X(12)/0., [illegible]., 3*[illegible]., [illegible]., 4*0., 2*[illegible]./,
     &      Y(12)/2*0., [illegible]., [illegible]*0., [illegible]., 0., [illegible]*[illegible]., 2*0./,
     &      Z(12)/4*3., 4*0., [illegible]., [illegible]., 3., 0./,
     &      PI/3.14159/
       INTEGER IAT(12)/0, [illegible]*1, 0, 1, 0, 1/
C
C          INITIALIZE AND CREATE WEDGE.
C
       CALL MGINIT(50)                                  initialize the MGI routines
       CALL MGPT2('WEDGE ',12,X,Y,Z,IAT,&800)
C
C          ERASE SCREEN AND READ ANGLE.
C
 10    CALL CKDER                                       computek routine to erase the screen
       PRINT 900
 900   FØRMAT('ENTER ANGLE IN DEGREES:')
       READ 905, ANG
 905   FØRMAT(F6.1)
C
C          DEFINE AND BUILD UP T1.
C
C           1)  SCALE UP BY 100.
C
       CALL MGTR('T1 ','SCA ',100.,&800)               define a transform
C
C           2)  TRANSLATE SØ THAT CENTER IS AT ØRGIN.
C
       CALL MGBTR('T1 ','TRA ',-1[illegible].,-100.,-150.,&800)      build up the
C                                                                      transform
C           3)  RØTATE ABØUT Y-AXIS.
C
       CALL MGBTR('T1 ','Y[illegible] ',ANG*PI/180.,&800)
C
C           4)  [illegible] A PERSPECTIVE.
C
       CALL MGBTR('T1 ','ZPE ',[illegible].,&800)
C
C           [illegible])  TRANSLATE TØ [illegible] ØF SCREEN.
C
       CALL MGBTR('T1 ','TRA ',[illegible].,[illegible].,[illegible].,&800)
C
C          GET [illegible]
C            (NØTE THAT [illegible])
C
       CALL MGPT[illegible]('WEDGE ','T1 ',[illegible])
C
C          CALL CØMPUTEK [illegible] LINE [illegible].
C
       CALL CKVAR[illegible](X,Y,NP,IAT,1)                  this computek routine draws lines
       READ 905, DUM                                      (visible and invisible) to the
       GØ TØ 10                                           coordinates specified by the X
C                                                         and Y vectors.
C          ERRØR.
C
 800   PAUSE 'ERRØR'
       GØ TØ 10
C
       END
```



WEDGE, Angle = -30°



WEDGE, Angle = 10°

Assemblies

An assembly is defined by the following data. . .

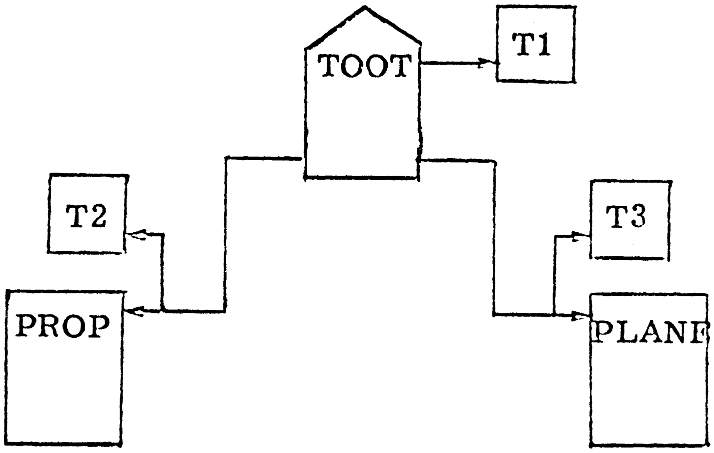
1. A legal MGI name,
2. The address of a primary transform (optional), and
3. The number of subelements (one or more)
4. One or more of the following pair:
    a. The address of a subelement, and
    b. The address of a secondary transform to be associated with this subelement (optional).

A call to the appropriate routine with the above information causes the assembly to be created.

Assemblies are used to relate parts and other subassemblies with transformations. An assembly can be manipulated in much the same way as a part.

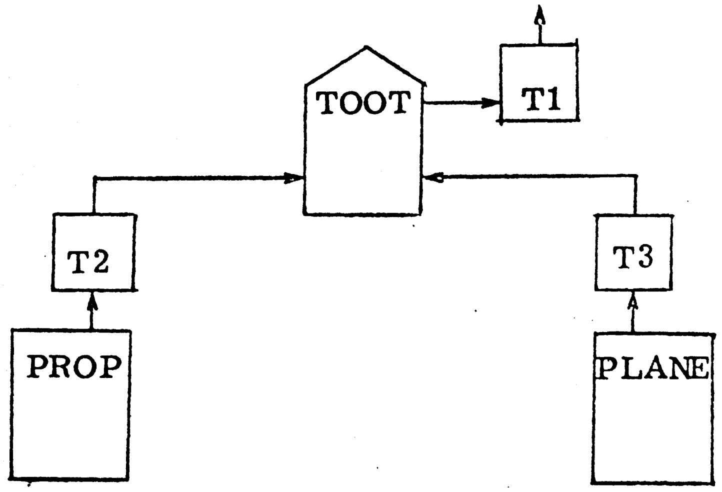
As an example, consider an assembly named TOOT which consists of two parts named PROP (a propeller) and PLANE (the rest of the plane). The points of TOOT may be transformed and recovered by calling the appropriate routine and referring to 'TOOT' by name. It's also possible to associate a transformation with TOOT's subelements PROP and PLANE (for example, to rotate PROP about its x axis).

A simplified graphical representation of the data structure for TOOT is. . .



In the above diagram, Tl is the transform associated with the assembly TOOT. T2 is a transform associated with PROP and T3 is a transform associated with PLANE.

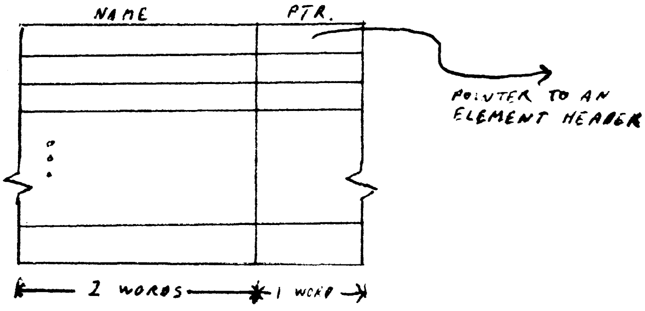
When the points of TOOT are to be recovered (to display the plane for example) the data can be thought to "flow" upward (see next diagram)



from PROP and PLANE through T2 and T3 respectively. PROP and PLANE are called the 'subelements' of this assembly. (In this example, the subelements are both parts, but they could be other assemblies.) Associated with each subelement of an assembly is a 'secondary transform', in this case T2 and T3. When an assembly is recovered, the points of each subelement are transformed by the appropriate secondary transforms. Associated with each assembly is a 'primary transform', in this case Tl. After being transformed by secondary transforms, all the points of an assembyy are transformed by the primary transform. It's thus possible, in effect, to manipulate the entire assembly by transforming the primary transform. (If no transformation is desired, the identity transform can be specified.)

List-Structure Implementation of MGI System

The MGI routines maintain a dictionary of element (part, transform, and assembly) names along with a pointer from each name to the element's header.



Headers are eight words long and of the form. . .

NAME (8 BYTES)	
BACK PTR	TYPE CODE
AMOUNT OF STORAGE	# OF ITEMS
PRIM TRANSFORM PTR	0

(FOLLOWED CONTIGUOUSLY BY THE FIRST NODE OF THE ELEMENT)

BACK PTR represents a back pointer into the dictionary. TYPE CODE indicates whether the header is for a part, transform, or assembly (1, 2, or 3). AMOUNT OF STORAGE holds the number of words of storage being used by the element. Depending on whether the element is a part or assembly, # OF ITEMS holds the number of points or number of subelements respectively. (If the element is a transform, the # OF ITEMS field is unused.) The PRIM TRANSFORM PTR field is only used when the element is an assembly in which case it points to the primary transform of the assembly if it exists.

Nodes are six words long.

PART-NODE

HX_i	HY_i
HZ_i	H_i
IAT_i	CP

TRANSFORM-NODE

T_{11}	T_{21}
T_{31}	T_{41}
T_{12}	T_{22}

(FOLLOWED CONTIGUOUSLY BY 2 MORE NODES CONTAINING T_{23}-T_{44})

ASSEMBLY-NODE

SECONDARY TRANSFORM PTR	SUBELEMENT PTR
SECONDARY TRANSFORM PTR	SUBELEMENT PTR
	CP

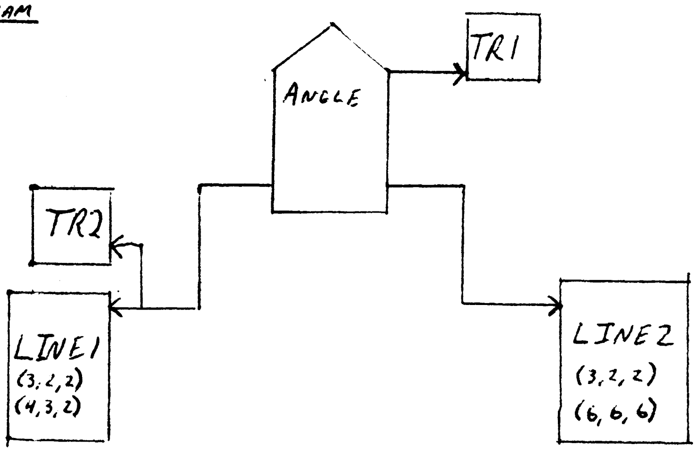
In the part-node, HX_i, HY_i, HZ_i, and H_i represent the homogeneous coordinates of a point, and IAT_i represents the attribute associated with the line drawn to that point (eg visible or invisible). CP represents a continuation pointer which either points to the next part-node or is 0 indicating that the next part-node follows contiguously.

Three contiguous transform nodes are used to hold the 16 elements of the transformation matrix.

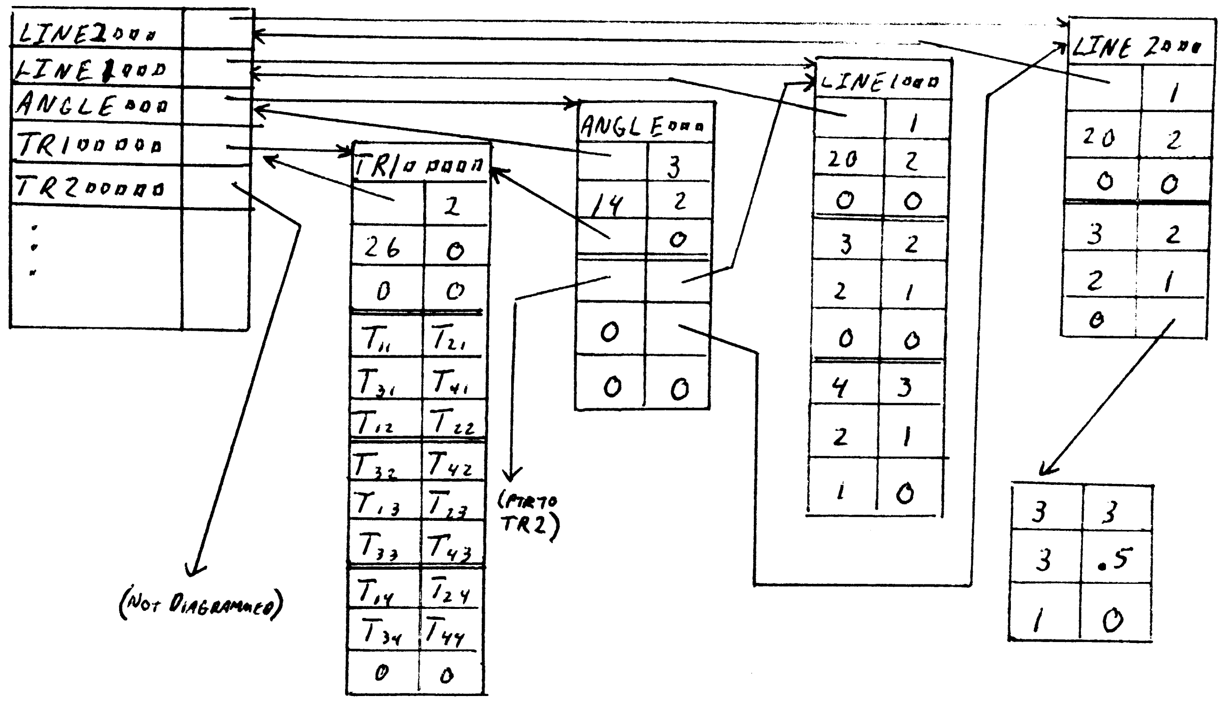
An assembly-node can contain two sets of pointers (in addition to the continuation pointer which has the same function as in the part-node). Each set of pointers points to an assembly subelement and the secondary transform which modifies it.

Consider the following example's simple and then more complete diagrams to get an overview of how the dictionary, headers, and nodes interwork.

SIMPLE DIAGRAM



More Complete Diagram



References:

1. MGI Users' Guide, John Van Roekel (Available at Aero. Eng. Office)

2. Engineering Summer Conference Notes on Graphics, 1971 (On homogeneous coordinates)

3. Drawl: Concomp Report #30, AD-715952

S. T. D. S. - System Basics

Beverly Katz
CCS 500
26 June 1972

Introduction

In the process of presenting the basics of Set Theoretic Data Structures (STDS), several areas will be covered. First, the system's history and underlying philosophy will be discussed. This will be followed by a general discussion of information retrieval systems. Then the machine environment will be contrasted to the information environment and the use of set theory in information systems will be explained. This will be followed by a discussion of system internals, implementation and limitations.

History and Underlying Philosophy

At the beginning of the ConComp Project at the University of Michigan in 1967, Project Director Franklin H. Westervelt initiated an investigation of data structures. He felt that present efforts in data structure development were too machine-oriented and that future needs would not be met. His contention was that users of large data bases should not be burdened with the intricacies of complicated data representation. Instead, data representation should be transparent to the user. Thus the need for a generalized data structure arose. This structure was not to be machine-oriented but information-oriented (where design emphasis was placed on information content instead of machine representation). Historically the development of a data structure was based on a particular problem for a specific machine and then generalized. Another approach would be to start with a general means of expressing any problem and then adapt the general representation for a particular machine. This approach eventually resulted in the development of Set Theoretic Data Structures (STDS) by Set Theoretic Information Systems, Inc. (STIS) of Ann Arbor.

It took over three years for the concept of STDS to become a usable interface. The preliminary work in this area began with Timothy E. Johnson and Jerome A. Feldman with Associated Data Structures. From this work, Associated Data Structures introduced the functional notation

$$A(0) = V$$

Its significance is that something done in the information environment can have an arbitrarty implementation in the machine. Functional notation has the limitation that its arguments have to be single valued. For example, the square root function $f(4) = 2$ by convention. But $f(4) = -2$ would be equally correct. David Childs, formerly of the ConComp Project and now with STIS, felt that set theory could be used instead of functional notation. This doesn't pose the problem presented above since the arguments and parameters are sets, hence $f(4) = \{2,-2\}$.

Information Retrieval Systems

In understanding why there is a need to create a more general system, one must understand the limitations of standard information retrieval methods and data structures. The traditional way to implement a retrieval system has been to determine what questions are to be asked of the data, then to write a program and choose an appropriate data structure to answer those questions. Therefore one is limited to that class of questions. Data organization methods are utilized for storing, updating and retrieving data. Each task demands a different data representation to be efficient. These different data organizations are used to bridge the gap between logical user requirements and the physical realities of storage media and computer hardware. Common data organization falls into four classifications - sequential, random, list and networks. Each one has specific assets and limitations.

Sequential organization offers fast accessing of sequential records but has difficulties in random accessing, inserting and deleting records.

Random organization provides efficiency for queries and updates but not for insertions and deletions. In addition, one is required to have indices or keys which cause redundancy in data representation. Sometimes dictionaries are required which increase the storage overhead.

List organization includes rings and simple and inverted list structures. In this type of organization, pointers are used to link the elements. These pointers increase the amount of storage used even though they increase retrieval efficiency. It is not uncommon for the pointers to take up more space than the data.

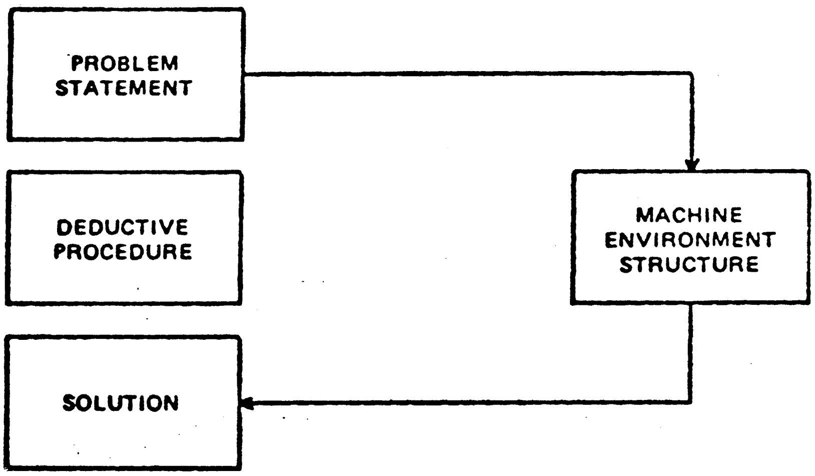
The general network organization is one in which several levels of pointes are used (i.e., tree structures). This structure is a collection of access paths. Each path is used to answer a specific type of questions. The problem with this type of structure is that it is almost impossible to tell when an existing path is obsolete since the user and his program become dependent on the paths. If all the paths are kept until the final reference then there exists an excessive number of paths using an excessive amount of storage.

Since no one representation is best or even adequate for all retrievals because data is usually bound to a machine representation, there exists a need for a new type of system. E. F. Codd suggests that the relational view of data is much superior to the existing noninferential, formated data systems. "It provides a means of describing data with its natural structure only — that is, without superimposing any additional structure for machine representaion purposes."[1] STDS epitomizes the relational model of data. It provides an efficient method (minimal cost) of making data transformation (i.e., changing data from one storage mode to another) without effecting the form of the retrieval program. Then any retrieval requirement could be facilitated by the selection from machine representations availabl

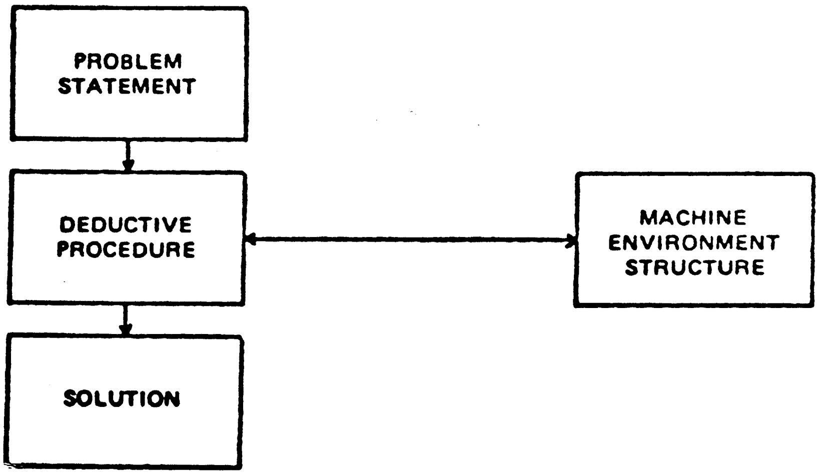
Any transformation can be made subject only to the gained or lost retrieval speed versus the increase or decrease in storage requirements. Therefore, the information is not limited by the restrictions imposed by machine representation.

In order to accomplish this, the characteristics of the information had to be isolated from those of the machine. Many of the problems in data management arise as a result of failing to recognize this distinction.

The former is the one that pervades current computer installations. It is represented by:



Schematically, the relationship of the IE, ME, and MES should look like this:



Diagrams from STIS Corporation.[2]

Machine Environment versus Information Environment

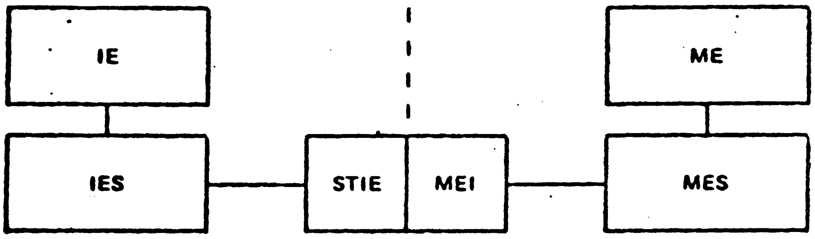
In order to understand the distinction between the information environment and the machine environment, it is necessary to see which characteristics and properties are attributed to each.

The information environment can be loosely characterized by items, data, relationships, queries, questions and answers while the machine environment can be described in terms of bits, bytes, words, addresses, contents of addresses, registers, sorts, searches and pointers. Problem statements and solutions are inherently a property of an information environment as is the deduction procedure used for obtaining the solution from the problem statement. Problem execution is a capability of a machine environment. The transformation operations are in this category. There exists a machine environment structure (MES) which specifies the operational particulars of a given machine environment. These include word size, memory size, languages available, etc. and all the other considerations that allow a user to interface with the machine environment. The information environment structure (IES) is a deductive procedure which accomodates the information environment.

In most systems, the IES must be a deductive procedure accomodating the information environment and the additional time and storage constraints imposed by the MES. In this case the

IES is the same as the MES. In order to achieve separation, two requirements must be met. First, a general deductive procedure language that's potentially expressive of any information environment. The second requirement is the construction and implementation of a language that accomodates the storage and time constraints of the MES. Set theory with the appropriate machine environment interface provides these properties that are necessary for isolating the two environements. STDS provides the interface between the environments.

Since set theory provides the properties necessary for isolating the information enciron-ment from the machine environment, any implementation of set theory with the appropriate machine environment interface will suffice.



SET THEORETIC DATA STRUCTURE

The implementation of a *set theoretic information environment* (STIE) and a *machine environment interface* (MEI) is called a *set theoretic data structure* (STDS).

The feasibility of an STDS has been shown. The viability of an STDS depends completely on the quality and caliber of the particular implementation.

SET THEORETIC DATA STRUCTURE AS AN INTERFACE BETWEEN THE INFORMATION ENVIRONMENT AND THE MACHINE ENVIRONMENT[3]

Set Theory in Information Systems

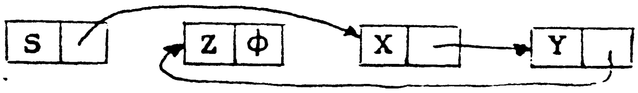
"Any data structure is actually an isomorphism between the machine environment and the information environment preserving the functional aspects of each."[4] Hence STDS is a data structure that can map the myriad relationships of the information environment into the algorithmic world of the computer. This data structure is machine independent since the user can access his data without needing to know which machine is used (subject to the limitations of certain machines). The second condition that is necessary for isolation is satisfied since general information requests can be abstracted to set operations. Set operations are defined in terms of results, not in terms of the means for obtaining results. Thus any procedure giving the correct result is legitimate.

For example, if an information request is expressed as a set operation , $(+)$, given data as sets A and B with the retrieved result as set C, then the abstraction can be stated:

$$A \; (+) \; B = C$$

In order to define this abstraction and therby prove that $(+)$ is a valid set theoretic operation, it is necessary than an element X is a member of C if and only if there exists a truth function relating A, B and X (i.e., $C = \{X: \psi(a,B,X)\}$). This is an existance criteria for C, not a procedure for constructing C. It is the fact that the function is decidable that makes $(+)$ a set theoretic operation and assures that C is defined. Nothing has constrained how A, B or C should be structured nor what procedure should be employed to construct C from A and B. For instance, let (+) be the same as U (union) then A U B = C. One doesn't care how this union is constructed or how the machine denotes C as A U B. Therefore, any convenient and/or economic representation of data (e.g., the sets A and B) fulfills the necessary and sufficient condition of isolation of the information environment from the machine environment. Hence the STDS routines are essentially machine independent.

Set operations are abstractions on sets yet the term "set" has a vague meaning when applied to an information retrieval system. All data can be represented as sets. A set has to be well ordered since there are n! ways of ordering n records. Thus, the lexicographical ordering was chosen. These sets are a collection of n-tuples (a record) represented by blocks of contiguous storage. It is important to prove the assertion that any data can be represented by a set. The method of proof will be by example. In these, S' denotes the address of the location that contains S and ϕ defines a null pointer. A list structure is one of the most commonly used data structures. Consider this linear list:



The only information that this structure contains is that S comes before X which is followed by Y which comes before Z. This information can be denoted by a set of three ordered pairs - $\{\langle S,X\rangle,\langle X,Y\rangle,\langle Y,Z\rangle\}$. If someone wanted to know the structural interpretation of the list, it could be represented by the following set:

$$\{\langle S',S,X'\rangle \; , \; \langle X',X,Y'\rangle \; , \; \langle Y',Y,Z'\rangle \; , \; \langle Z',Z,\phi\rangle\}$$

A set can be used to describe more complex list structures. They can describe the individual cells of a list structure. Let A be a set which contains only one 4-tuple:

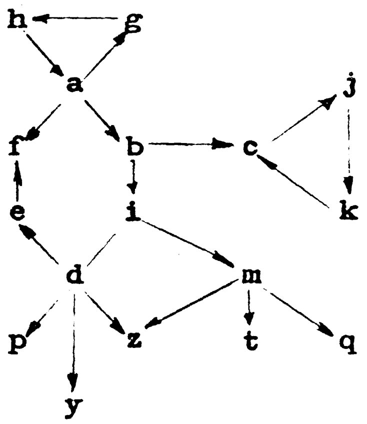
$$A = \{\langle 2, 6, 4, 4\rangle\}$$

and let B be a set which also contains only one 4-tuple:

B = {⟨type,information,forward pointer,backward pointer⟩}

If a user wants to know what a cell looks like, all he has to do is list A + B. He will find that the type of cell is found in the first two bytes, the information in the cell is in the next six bytes, the pointer to the next cell is in the next four bytes and the pointer to the previous cell is is the last four bytes. If C is the list structure being described then C is a collection of n-tuples where an n-tuple is sixteen bytes (in this case). This scheme can be used to describe any list structure.

It might seem hard to transform a network structure into a set. This composite structure can be represented by the following set of twenty ordered pairs:



$$\{\langle g,h\rangle, \langle h,a\rangle, \langle a,f\rangle, \langle a,b\rangle, \langle a,g\rangle, \langle b,i\rangle, \langle b,c\rangle, \langle c,j\rangle, \langle j,k\rangle, \langle k,c\rangle, \langle i,m\rangle, \langle i,d\rangle, \langle m,q\rangle, \langle m,t\rangle, \langle m,z\rangle, \langle d,z\rangle, \langle d,y\rangle, \langle d,p\rangle, \langle d,e\rangle, \langle e,f\rangle\}$$

Hence, every data structure and all types of data can be represented as a set.

Sets can be represented internally by many different data structures. Each different representation is referred to as a "mode". These data structures were developed by STIS Corp. which take advantage of the capabilities and limitations of a specific MES. It is through a set's transformation into these different modes that efficiency is achieved. STIS Corp. believes that for any retrieval request there exists a best data structure (mode). Since sets can be transformed easily from one mode to another, STDS can provide the best data structure for each retrieval request, if the best mode is known.

To understand the basic internal scheme of STDS, one must understand some basic properties of sets. Every stored representation of a set must preserve all of the properties of the set. Every representation of a particular set must behave identically under set operations. For example, let A be a set of students and B be a set of Michigan residents. Then to find the set C consisting of all students who are Michigan residents, C would equal the intersection of A and B. Let A have mode(6) which may be assumed to have slow retrieval and small storage properties. Assign mode(3) to B and assume that it has fast retrieval characteristics but requires a large amount of storage. Consider the set theoretic expression $A \cap B = C$; C after the intersection operation will be of the default mode. If the mode of A were changed from 6 to 3 or if the mode of B were changed from

3 to 6 or if both modes were changed, the resulting set C would be exactly the same, only the time to execute the intersection operation and the resulting default mode for C might vary.

System Internals

STDS is constructed in five structurally independent parts:

(1) a collection of set operations, S

(2) a set of datum names, B, which point to the ith generator set's header

(3) the data which is a collection of datum definitions, one for each datum item

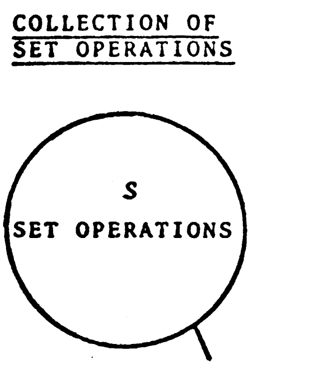
(4) a collection of set names, N

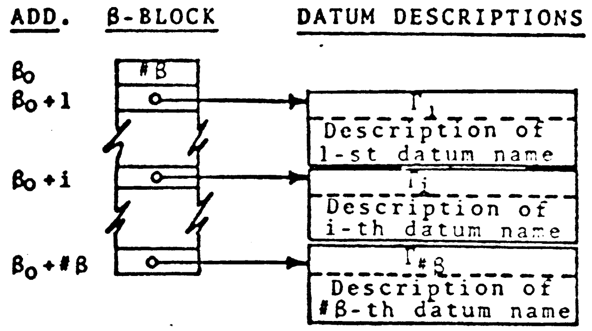
(5) a collection of set representations each with a name in N

STDS consists of a series of modules, most of which are set operations, written in FORTRAN which access sets through the pointers in N. All information between sets can be expressed as a set theoretical expression generated from an operation in S. Here is one of the limitations of STDS. Since all retrieval requests have to be expressed in set theoretical expressions, there may exist a request which cannot be transformed into those terms. The set theoretical expressions determine what sets are to be accessed and which operations are to be performed so that all storage requirements are known prior to the execution of the operations. If a request requires more storage than can be obtained from the system, then the operation is not performed at all; there is no partial completion. There are no pointers between sets so the collection of set operations acts as the only structural tie between sets.

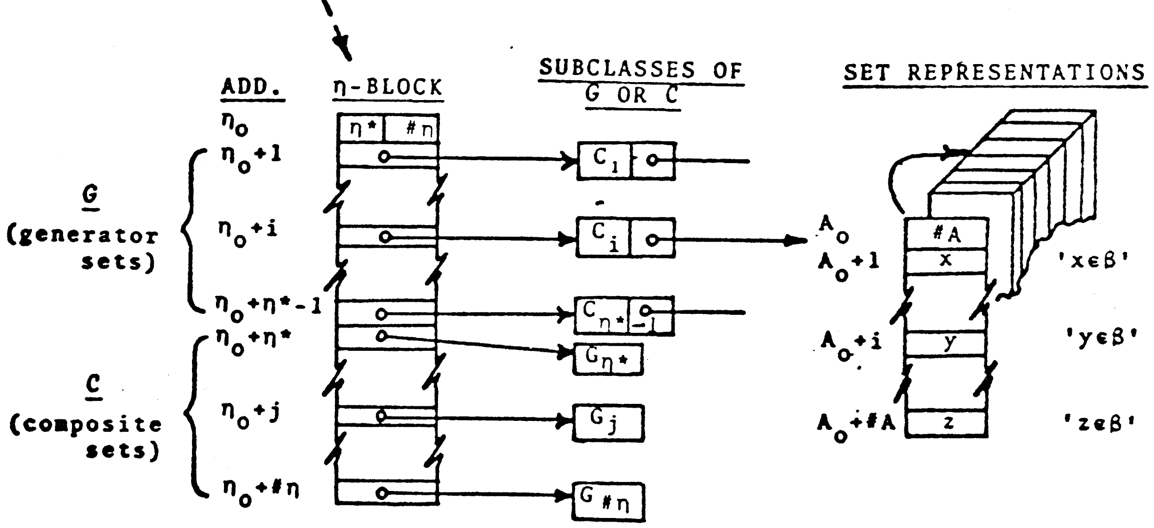
B is the set of datum names which is considered a block in set) which locates a section of core that list the generator set names that are used by the composite set. Suppose that a new set A is created so that two new elements are added to the N block. Let N(a) denote the generator set and N(a*) denote the composite set. The content pointer in N(a) locates a section of core which contains a*, the address of the data set in core. N(a*) points to a location of storage which only contains a. If more information, A', has to be added to A and more contiguous space is available, the N block has to be updated. A new generator set element a' is created. N(a') points to core which contains a', the address of A'. Nothing happens t_o the section of core pointed to by N(a). The location to which N(a*) points is now changed to denote that A is really A U A' which contains a and a'.

There are no pointers between sets. The N block and B block are the only pointers in STDS. Each set has basically only one pointer in N associated with it so that it can be easily updated and relocated. Deletions can be easily accomplished. The element that is to be deleted is replaced by the least element of the set. The "new" set is well ordered and takes up one n-tuple less storage so that an insertion can be easily completed. To add a new record to a set, it is put in the next contiguous location, if space is available, and then the set is re-ordered. If there is no more contiguous space, a new set is formed (unknown to the user) and the N block is updated as previously described.




ADD.
β-BLOCK
DATUM DESCRIPTIONS
β₀
β₀+1
β₀+i
β₀+#β
#β
Γ₁
Description of 1-st datum name
Γᵢ
Description of i-th datum name
Γ#β
Description of #β-th datum name

1. β is the set of datum names: $\beta = \{1,2,\ldots,\#\beta\}$
2. $\beta_o, \beta_o+1, \ldots, \beta_o+\#\beta$ are addresses of β-block.
3. β-block contains pointers to datum descriptions and lists of generator sets, Γ, using the datum name.
4. For each datum name 'i' in β, Γ_i is a subclass of G.



5. η is the class of set names: $\eta = \{1,2,\ldots,\#\eta\}$.
6. $\eta_o, \eta_o+1, \ldots, \eta_o+\#\eta$ are addresses of η-block.
7. $A_o, A_o+1, \ldots, A_o+\#A$ are addresses of elements of β contained in set A.
8. G is the class of generator sets: $G = \{1,\ldots,\eta^*-1\}$.
9. C is the class of composite sets: $C = \{\eta^*,\ldots,\#\eta\}$.
10. $\eta = G \cup C$
11. The C_i are subclasses of C.
12. The G_j are subclasses of G.

Storage Schema of an STDS.[5]

System Implementation

STDS is implemented in four sections:

1. A series of efficient sorts
2. Modular set operations
3. Garbage collection routines
4. A series of linkages.

The implementation discussed herein pertains to the demonstration version - STDS-DEMO.

There is a series of efficient algorithms which sort the whole n-tuple lexicographically. Hence a set is composed of records in lexical order. All of the set operations are modular routines and new operations may be added at any time without disturbing those previously implemented.

The system is modular for an eventual adaptation to hardwired integrated circuits. STIS is currently trying to market a "hardware data management processor" which they call Set Theoretic Information Concentrator. This "concentrator" is a hardware device operationally situated between the central processing unit and the peripheral storage. The information concentrator interprets an I/O access as an information access and reads from the peripheral storage device as much information as will fit into memory. This process permits fewer I/O accesses to retrieve equivalent amounts of information, thus converting many I/O constrined sytems into CPU bound systems. The STIC offers many data management capabilities:

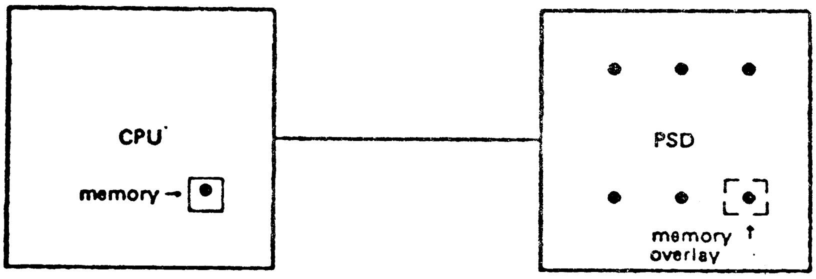
GENERAL DESCRIPTION OF A SET THEORETIC INFORMATION CONCENTRATOR 6

A *set theoretic information concentrator* (STIC) is a hardware device operationally situated between a central processing unit (CPU) and peripheral storage devices (PSD's) providing the following data management capabilities:

- minimization of input/output (I/O) accesses with maximization of information content
- unanticipated query.capability
- multi-user access and concurrent updating
- utilization of any or all data access methods
- complete data management, processing, querying, and updating
- separate control over optimization of storage size and retrieval times (S/T)
- restricted access and manipulation protection from unauthorized users
- data base management to optimize physical placement of data for different job loads or mixes
- frees user from concern with machine representation of data
- no *a priori* restrictions on achieving "best times", "least storage", and "general interrogation" that a given CPU and PSD will allow
- data can be stored in its theoretically most compact form and then transformed into its theoretically best retrieval form
- provides information expansion through simultaneous access of multiple data bases
- provides separation of information representation from storage representation

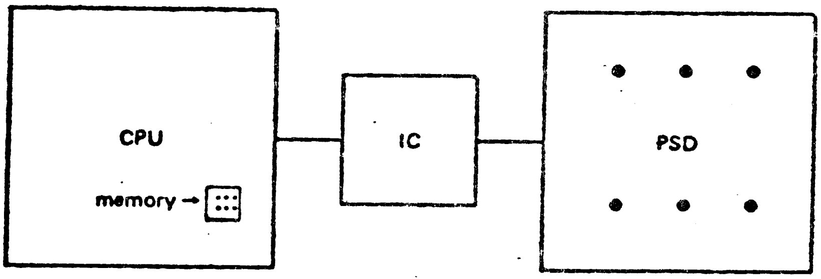
PRINCIPAL OF INFORMATION CONCENTRATION[7]

USUAL COMPUTER INSTALLATION CONFIGURATION



When the CPU needs new information in memory, an I/O access loads the memory from the PSD. The structure of the information in memory and the structure of the information in the PSD *are identical.* If the information in the PSD is structured so that only a small portion of usable information can fit in memory at any one time, then the host program can become I/O bound.

COMPUTER INSTALLATION WITH AN INFORMATION CONCENTRATOR



The IC interprets an I/O access as an information access and "gathers" as much information from the PSD as will fit into memory. This process permits fewer I/O accesses to retrieve equivalent amounts of information, thus converting many I/O bound problems to CPU bound problems.

These capabilities must be taken in light of the fact that they are only promises as STIC is not yet an operational reality.

When using STDS the user must keep track of the amount of storage that is being used. The user must explicitly call for the system's garbage collection routines or eventually run out of storage.. Set theoretic operations can be called by FORTRAN or PL/1 as subroutines. To accomplish this, STDS provides the necessary linkages. A set is represented as a sequential file with a maximum record size of 32,768 bytes.

The input to STDS should be a sequential file; later versions will take a line file and do the conversion for the users. After the data is read in by STDS, it must be converted into a set. To be a set the N block, B block and accompanying regions must be updated and a set must be created. The header contains the maximum length of the n-tuples in the set and the set's cardinality, the number of n-tuples in the set.

System Limitations

Most of the current limitations are due to the vast storage requirements of set theoretic operations. The current input limit of 255 pages causes difficulties for users with large data bases. As stated previously, the amount of storage needed for a set theoretic operation is determined before its execution. Sometimes the system determines that much more storage is necessary for the completion of the operation than is needed to contain the end result. Thus the storage limitation will not allow the operation to execute, even though there might be enough room. A similar problem is caused by the inability to get enough storage during peak time-sharing periods.

Currently, STDS requires a virtual memory machine, particularly the IBM System 360/67. Later versions are supposed to be free of this limitation, however.

One common problem is the user's inability to understand and use set theory with respect to information retrieval systems. Some users have trouble translating their ideas into set theoretic expressions.

Conclusions

Set theoretic data structures are a fresh approach to the current state of information retrieval systems. Further development in this area by STIS Corporation and others might lead to a universal, generalized information retrieval system.

APPENDIX A[8]

Description of the Demonstration Files for STDS used in Appendix B

Universe: March 1967 Current Population Survey with Income and Work Experience Supplements Included (Bureau of Labor Statistics). Married, Spouse Present, Head and Wife of Family or Subfamily, Age of Wife 21 or over, Living in 96 of largest 104 Standard Metropolitan Statistical Areas (SMSA).

Item #	Unit	Characteristics	Coding
Family Files			
1.	Family	Family Code Number	1-11629
2.	Family	Primary Family	0
		Each Subfamily or Secondary Family[b]	1-6
3.	Family	Type of Family	
		Primary Family	1
		Subfamily	2
		Secondary Family	3
4.	Family	Weight[a]	0-999999
5.	Family	Presence of Own Children	
		None	0
		All 6-17	1
		None Under 3, Some 3-5, Some 6-17	2
		All 3-5	3
		Some Under 3	4
6.	Family	Relation of Wife to Head of Household	
		Wife	2
		Child	3
		Other Relative	4
		Non-Relative	5,6
7.	SMSA	CPS Unemployment Rate	
		0-9.7%	0-97
		Over 9.7%	98
8.	SMSA	BLS Employment Change, 1966-67	
		Under .1%	0
		0.1-9.7%	1-97
		Over 9.7 %	98

Item #	Unit	Characteristics	Coding
9.	SMSA	Relative Opportunities	
		.001-.997	1-997
		.998 or more	998
Wife Files			
1.	Family	Family Code Number	1-11629
2.	Family	Primary Family	0
		Each Subfamily or Secondary Family[b]	1-6
3.	Family	Race of Wife	
		White	1
		Negro	2
		Other	3
4.	Wife	Age	
		Age at last birthday	14-98
		Age 99 or over	99
5.	Wife	Labor Force Status, March	
		Not in Labor Force	1
		In Labor Force	2
6.	Wife	Employment Status	
		Employed:	
		At work full-time	01
		At work part-time	02
		With a job,not at work	03
		Unemployed:	
		Looking for work	04
		Temporary lay-off	05
		New Job	06
		New Job, School	07
		Out of Labor Force:	
		House	08
		School	09
		Unable	10
		Unpaid,less than 15 hrs.	11
		Other	12
7.	Wife	Recoded-Intermediate Hours	
		1-34 Hours	
		Usually full-time, Economic	1
		Usually full-time, Other	2

Item #	Unit	Characteristic	Coding
		Usually part-time, Economic	3
		Usually part-time, Other	4
		35-39 Hours	5
		40 Hours	6
		41-47 Hours	7
		48+ Hours	8
		Intermediate Duration of Unemployment not coded 1 or 2	9
8.	Wife	Intermediate Duration of Unemployment	
		Under 4 weeks	1
		4 weeks	2
		5-6 weeks	3
		7-10 weeks	4
		11-14 weeks	5
		15-26 weeks	6
		Over 26 weeks	7
		Not unemployed	9
9.	Wife	Years of School Completed	
		None	1
		1-4 elementary	2
		5-7 elementary	3
		8 elementary	4
		1-3 high school	5
		4 high school	6
		1-3 college	7
		4 college	8
		5 or more college	0
10.	Wife	FILOW	
		Negative or none	0
		Amount	1-24999
		$25,000 or over	25000
Husband Files			
1.	Family	Family Code Number	1-11629
2.	Family	Primary Family	0
		Each Subfamily or Secondary Family	1-6
3.	Husband	Age	
		Age at last birthday	14-98
		Age 99 or over	99

Item #	Unit	Characteristic	Coding
4.	Husband	Labor Force Status, March	
		Not in Labor Force	1
		In Labor Force	2
		Armed Forces	9
5.	Husband	Employment Status	
		Employed:	
		At work full-time	01
		At work part-time	02
		With a job,not at work	03
		Unemployed:	
		Looking for work	04
		Temporary lay-off	05
		New Job	06
		New Job, School	07
		Out of Labor Force:	
		House	08
		School	09
		Unable	10
		Unpaid,less than 15 hrs.	11
		Other	12
		Armed Forces	99
6.	Husband	Recoded-Intermediate Hours	
		1-34 Hours:	
		Usually full-time, Economic	1
		Usually full-time, Other	2
		Usually part-time, Economic	3
		Usually part-time, Other	4
		35-39 Hours:	5
		40 Hours	6
		41-47 Hours	7
		48+ Hours	8
		Intermediate Duration of Unemployment not coded 1 or 2	9
7.	Husband	Intermediate Duration of Unemployment	
		Under 4 weeks	1
		4 weeks	2
		5-6 weeks	3
		7-10 weeks	4
		11-14 weeks	5
		15-26 weeks	6
		Over 26 weeks	7
		Not unemployed	9

Item #	Unit	Characteristic	Coding
8.	Husband	Years of School Completed	
		None	1
		1-4 elementary	2
		5-7 elementary	3
		8 elementary	4
		1-3 high school	5
		4 high school	6
		1-3 college	7
		4 college	8
		5 or more college	0
9.	Husband	FILOW	
		Negative or none	0
		Amount	1-24999
		$25,000 or over	25000

[a] Weight in file is 100 times the true weight.

[b] Each subfamily or secondary family within a primary family unit has a separate number. Subfamilies and secondary families are contiguous on the file to their respective primary family.

APPENDIX B[9]

Example Use of STDS

```
#R IC47:STDS*
#EXECUTION BEGINS

 ** SET-THEORETIC DATA STRUCTURE:  INTERACTIVE DEMONSTRATION **
     (2/18/70)

 FOR AN EXPLANATION ENTER "1":
```

(See Appendix A)

```
?GET(H)
        FILE = CPSH1
  ENTER PRINT FORMAT: (9I8)
 DONE! L(*)= 9 C(*)= 5812   ( 0.2912 SEC)
```

H is the set of husbands (heads of households) (See Appendix C)

```
?GET(W)
        FILE = CPSW1
  ENTER PRINT FORMAT:  (10I7)
 DONE! L(*)= 10 C(*)= 5812   ( 0.3250 SEC)
```

W is the set of wives. See Appendix C.

```
?SET(UNEM,4,6,5,7)
   * DONE!  L(*)=   1  C(*)=    4   ( 0.0036 SEC)
```

UNEM is the set of codes for "UNEMPLOYED"

```
?RS(5,H,UNEM,UH)
 DONE!  L(*)=    9  C(*)=  95   ( 0.4715 SEC)
```

UH is set of unemployed husbands

```
?XPAN(2,W,UH,WUH,1)
 DONE!  L(*)=    17  C(*)=  95  (0.5356 SEC)
```

WUH combines (matches) the wives of unemployed husbands in a set of 17-tuples (wife-husband relationships)

```
?IGTJ(WUH,4,11,OLDR)
DONE! L(*)=  17  C(*)=    13 ( 0.0023 SEC)
```

OLDR is the set of wives of unemployed husband's who are older than their husbands.

```
?RS(6,OLDR,UNEM,UNW)
DONE!  L(*)=   17   C(*)=   1 ( 0.0023 SEC)
```

UNW is the set of unemployed wives of unemployed husbands who are older than their husbands.

```
?LIST(UNW,1,1,-1)
    ENTER OUTPUT FORMAT:    (5I10)
      2006         0         1        57         2
         5         9         6         3      6520
        53         2         5         9         6
         6      6321
```

By examining this with the codes in Appendix A, it may be seen that we have located a 57 year old wife of a 53 old husband such that the FILOW (family income less own wages) for each is in excess of 6000 dollars. This may be due to pensions, interest, dividends, capital gains or other sources, explaining the basis of this data sample instance. Observe the CPU times used to find this result from an initial set of 5812.

```
                                              Total time to do STDS
?MTS                                          operations = 1.644 sec.
 #SIGNOFF
 #OFF AT 18:11.26
 #ELAPSED TIME       1400.7      SEC.← due largely to typed comments!
 #CPU TIME USED        10.471    SEC.← due largely to program "loading
 #STORAGE USED        929.573    PAGE-SEC.          and "relocation"!
 #DRUM READS          107
 #APPROX. COST OF THIS RUN     $2.42
 #FILE STORAGE        841 PG-HR      $.12
```

The above is an actual STDS session retyped for greater readability using italics to distinguish the comments.

OPERATIONS EXTENDED TO N-TUPLES

A = {<p,q,r,s,t>,
<k,z,m,n,o>,
<a,b,c,d,e>,
<u,v,w,x,y>,
<f,g,h,i,j> }

I-TH DOMAIN of A

DM(1,A,C)	C = {a,f,k,p,u}
DM(3,A,C)	C = {c,h,m,r,w}

I-TH RESTRICTION of A to B

Let B = {z,s} then for
RS(2,A,B,C)

C = {<k,z,m,n,o>}

and

RS(4,A,B,C)

C = {<p,q,r,s,t>}

LET A = {<name,mother,father,spouse,sex>}

GIVEN x ,find all sisters of x.

let X = {x} and W = {female}

RS(1,A,X,B)	B = {n-tuples with x in position 1}
DM(2,B,M)	M = {x's mother, m}
DM(3,B,F)	F = {x's father, f}
RS(2,A,M,C)	C = {n-tuples with m in position 2}
RS(3,A,F,D)	D = {n-tuples with f in position 3}
IN(C,D,G)	G = {intersection of D and C}
RS(5,G,W,H)	H = {n-tuples of G, female in 5 pos.}
RL(DM(1,H),X,S)	S = {sisters of x }

Footnotes

[1] E. F. Codd, "A Relational Model of Data For Large Shared Data Bank," *CACM*, June 1970, pp.377-387.

[2] S.T.I.S. Corp., "Set Theoretic Information Concentrator," 1972.

[3] Ibid.

[4] D. L. Childs, *STIS Technical Report 1 - Development of a Set Theoretic Data Structure*, p. 3.

[5] D. L. Childs,"Description of a Set Theoretic Data Structure," ConComp Technical Report No. 3, University of Michigan, March, 1968, p. 5.

[6] STIS Corp., Op. Cit.

[7] Ibid.

[8] D. L. Childs, STIS Technical Report 1, ,pp.C1-C5.

[9] Ibid., pp. D1-D2.

REFERENCES

Childs,D. L.,*Description of a Set-Theoretical Data Structure*, Technical Report No. 3, Concomp Project,University of Michigan, March 1968.

Childs,D. L.,*Feasibility of a Set-theoretic Data Structure*, Techinal Report No.6, Concomp Project,University of MIchigam, August 1968.

Childs,D. L.,S.T.I.S. Technical Report No. 1, *Developement of a Set-Theoretical Data Structure-Basis for Machine Independent Information Management System*,S.T.I.S. Corp.,Ann Arbor, Mi..

Codd,E. F.,*A Relational Model of Data for Large Shared Data Bank*, Communications of A.C.M.,Vol. 13, No. 6, June 1970, pp377-87.

Dodd,George G.,*Elements of Data Management Systems*,Computing Surveys, Vol. 1, No. 2, June 1969, pp 132-177.

Hershey,W. R. and Easthope,C. H., *A Set-Theoretical Data Structure and Retrieval Language*,L.M.I.S. Project,University of Michigan, April 1972.

Ray, Paul H., *Users Guide to STDS-Demo. System*, Ann Arbor,Mi., March 22,1971.

S.T.I.S.,*Operations Manual for STDS-1*,S.T.I.S. Corp., Ann Arbor, Mi., December 1971.

SAMPLE PROGRAMS

To provide some comparison between the various languages presented in the seminar, a sample problem was devised and the seminar participants provided solutions to this problem in several languages. The problem description is on the following page; the sample programs constitute the remainder of this section. We offer them with no further comment, other than the comments of the authors.

Simple Standard Problem for String and List Processing Seminar Participants

Those persons who presented source language descriptions in the seminar are requested to write a small program (whose function is described later) for inclusion in the final seminar report. The obvious intent is to have one problem common to all reports, so as to gain some feeling for the differences in the various languages. The program should function as follows:

1. the input is a variable-size set of alphabetic strings (A,B,...,Z), each string being one to four characters in length
2. the strings should be placed in alphabetic order (standard dictionary sequence) by the program
3. the internal ordering mechanism should be the construction, as a function of the input, of an ordered binary tree (this will be discussed in class)
4. the program should then produce as output the alphabetized list of strings.

Since there is a great deal of variation in the languages, we leave to each of you the design of the input formats and output format. Your program should read multiple data sets (as described in 1. and 2.); hence you will need one or two terminator symbols. For the final report, we would like:

1. a brief, at most one page, prose description of the program
2. a listing of the program
3. program results (if you are able to run the program in MTS) for the following input orderings:

   a. ordered

   b. reverse ordered

   c. randomly ordered

   each data set should contain

   a. C CC CCC CCCC

   b. at least one string < C

   c. at least one string > CCCC

   d. any other strings.

SLIP (Dean Lucier)

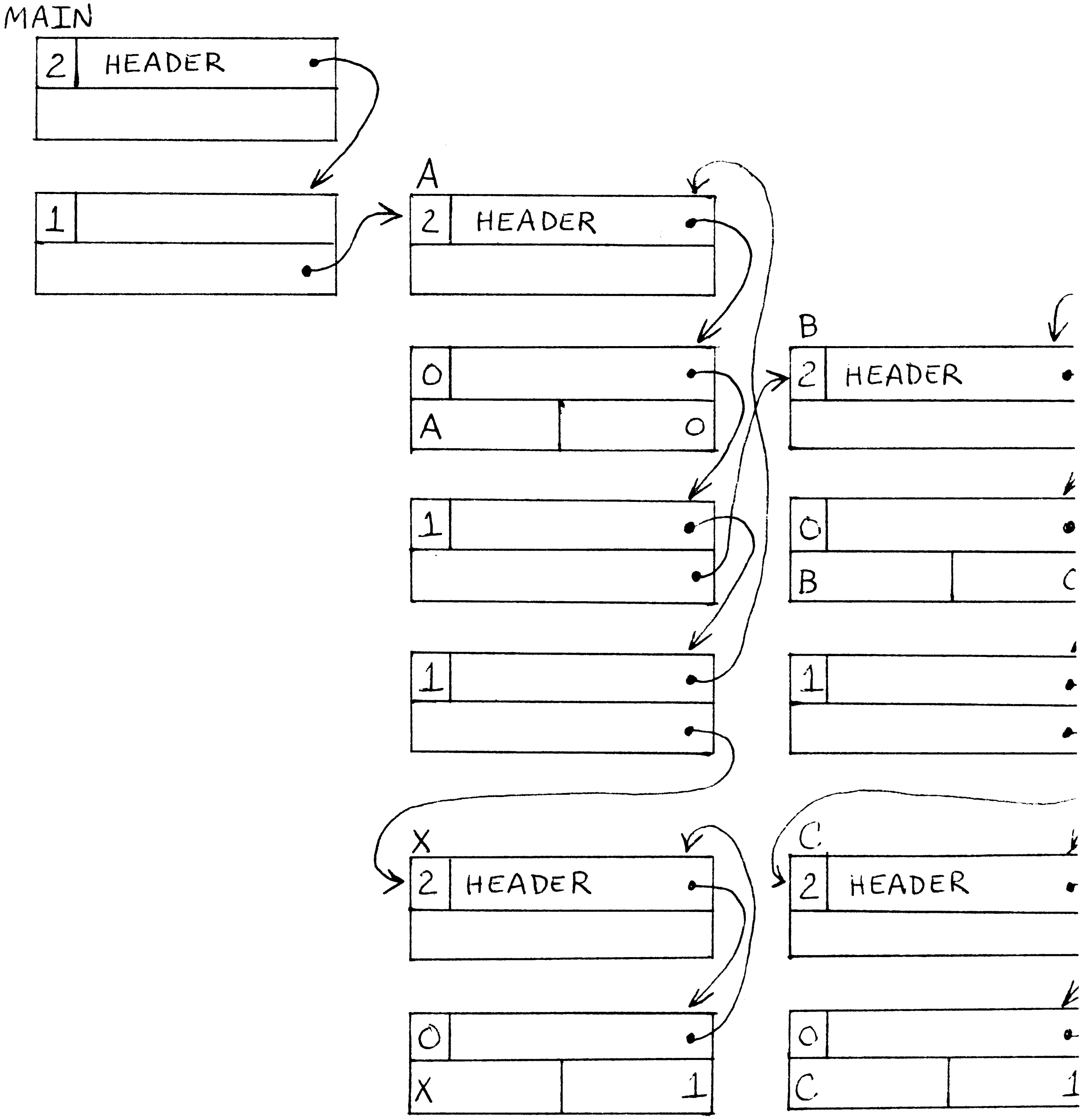
The sample program takes as input a variable number of one to four character strings and proceeds to build a binary tree to represent their alphabetic ordering. The input format is one string per card, left justified with a blank used as a delimeter for the length of the character string. A period in column one denotes the end of the set of strings which need to be alphabetized. An asterisk in column one denotes the end of the file.

Each string is examined to see if its duplicate is in the tree already. A header cell is created for each new letter at each level required. At the same time a type 0 cell is placed at the top of the list, which contains in the datum the letter it represents and a count of the number of times this letter, associated with previous letters by being linked as a sublist, was the last character in the string. On each of these lists are name cells which point to succeeding lists (letters in strings) which followed the letter representing the node.

The routine to print out the tree structure is then quite simple. Starting with the main list and using the structural advance, it prints out the datum portion of all type 0 cells which have a termination count greater than zero.

The function LCNTR(READER) provides the level of the sublist so the string can be rebuilt for printing. An example of the list structures for the strings ABC and AX are shown on the next page.

The MTS SLIP as shown in the program listing does have a bug in the advance linear operations using the predecessor (left) linkage. Therefore, when the data strings are not in order, my routine which is used to insert a cell in front of another cell fails.


MAIN
2
HEADER
1
A
2
HEADER
0
A
0
1
1
X
2
HEADER
0
X
1
B
2
HEADER
0
B
C
1
C
2
HEADER
0
C
1

```
C           360 / SLIP SAMPLE PROGRAM
C                        SLIP IS EMBEDDED IN FORTRAN. FOR EASIER
C IDENTIFICATION
C                 EACH SLIP STATEMENT IS FOLLOWED BY A LINE OF ASTERISKS.
C                                                    *****************
C
C           THIS PROGRAM WAS WRITTEN AS PART OF A THE COURSE REQUIREMENT
C                 IN A STRING AND LIST PROCESSING SEMINAR AT THE UNIVERSITY
C                 OF MICHIGAN IN MAY-JUNE 1972.
C                                         AUTHOR.   DEAN LUCIER
            COMMON    AVSL,W(100)
            DOUBLE PRECISION SPACE(1000),INITAS,MAIN,READER,FLAG,STUFF,
     X            ASTUFF,INSARG,NEWLST,LST,NAME,LIST,LRDROV,LVLRVT,ADVSER,
     X            ADVLWL,ADVLNR,LPNTR,SUBST,LVLRV1,NEWTOP,NEWBOT,NXTLFT,
     X            LOFRDR,INITRD
            INTEGER   A(4),LVL,PARENT,PCOUNT,I,B,C(2),STAR,PERIOD,
     X            BLANK,TRALST,TRARDR,LCNTR
            DATA STAR/4H*   /, PERIOD/4H.   /, BLANK/4H    /
            EQUIVALENCE  (STUFF,PARENT,C(1)),(PCOUNT,C(2))
  900       FORMAT (4A1,75X,A1)
  910       FORMAT (/4A1)
  920       FORMAT (/4A4)
  940       FORMAT  (24H HOW ABOUT THESE RESULTS)
  960       FORMAT (/27H1HERE ARE THE INPUT STRINGS)
  980       FORMAT (/27H1HERE IS THE GENERATED TREE)
  990       FORMAT (/42H1HERE IS THE FINAL INTERNAL LIST STRUCTURE)
  995       FORMAT (/22H WE ARE ABOUT TO CRASH)
            M = 150
C       THIS WILL PROVIDE 75 SLIPCELLS ON LAVS (MTS USES 2 DOUBLEWORDS)
            CALL INITAS(SPACE,M)
C                 ********************      CREATE A LAVS AND PUBLIC STORAGE
            CALL SETRAC(SPACE,M)
C                 ***************      SET UP SLIP DUMP IF LAVS EXHAUSTED
   10       CALL LIST(MAIN)
C                 **********                GET MAIN LIST HEADER CELL
            READER = LRDROV(MAIN)
C                       ************        ACQUIRE A READER FOR THE LIST
            WRITE (6,960)
   20       CALL INITRD(LVLRVT(READER))
C                 **********************    RETURN TO HEADER OF MAIN LIST
   25       READ (5,900)  (A(I),I=1,4),B
            WRITE (6,910) (A(I),I=1,4)
            IF (A(1).EQ.STAR)    GO TO  800
            IF (A(1).EQ.PERIOD) GO TO  500
C       AT THIS POINT WE HAVE A STRING IN THE A ARRAY - LEFT JUSTIFIED
            LVL = 1
   50       STUFF = ADVSER(READER,FLAG)
C                       ******************  LOOK FOR A SUBLIST
            IF (FLAG.NE.000)    GO TO  450
            IF (A(LVL).LT.PARENT)     GO TO 200
C       SINCE CHARACTERS ARE LEFT-JUSTIFIED, THE HIGH-ORDER BIT IS ON
C           SIGNIFYING NEGATIVE
            IF (A(LVL).GT.PARENT)     GO TO 300
C       THE CURRENT STRING MATCHES THIS LEVEL OF THE TREE
   88       IF (LVL - 4)  90,95,95
```

```
 0031          90          LVL = LVL + 1                                                      59.000
 0032                      IF (A(LVL).NE.BLANK)      GO TO 400                                60.000
 0033          95          PCOUNT = PCOUNT + 1                                                61.000
 0034                      ASTUFF = LPNTR(READER)                                             62.000
               C                   *************           GET ADDRESS OF CURRENT CELL        63.000
 0035                      CALL SUBST(STUFF,ASTUFF)                                           64.000
               C                *******************        REPLACE INCREMENTED COUNTER        65.000
               C      AT THIS POINT - DONE WITH CURRENT STRING - GO GET ANOTHER               66.000
 0036                      GO TO 20                                                           67.000
 0037          200         CALL LVLRV1(READER)                                                68.000
               C                **************       RETURN TO POINT OF DESCENSION            69.000
 0038                      INSARG = LPNTR(READER)                                             70.000
               C                   *************     RETURNS ADDR OF CURRENT CELL             71.000
 0039                      PARENT = A(LVL)                                                    72.000
 0040                      PCOUNT = 0                                                         73.000
 0041                      NEWLST = LIST(9)                                                   74.000
               C                   *********               GET NEW SUBLIST-REF COUNT = 0      75.000
 0042                      CALL NEWTOP(STUFF,NEWLST)                                          76.000
               C                ********************       INSERT PARENT ID ON SUBLIST        77.000
 0043                      CALL NXTLFT(NEWLST,INSARG)                                         78.000
               C                *********************      LINK NEW SUBLIST                   79.000
 0044                      WRITE (6,995)                                                      80.000
               C      THE LEFT POINTER FUNCTIONS HAVE A BUG IN THEM - WATCH ME CRASH          81.000
 0045                      STUFF = ADVLWL(READER,FLAG)                                        82.000
               C                  *******************      POSITION ON NEW NAME CELL          83.000
 0046                      WRITE (6,995)                                                      84.000
               C      WE WILL NEVER GET THIS FAR - I TOLD YOU SO                              85.000
 0047                      GO TO 50                                                           86.000
 0048           300        CALL  LVLRV1(READER)                                               87.000
               C                 *************             RETURN TO POINT OF DESCENSION      88.000
 0049                      STUFF = ADVLNR(READER,FLAG)                                        89.000
               C                  *******************      LOOK FOR NEXT NAME CELL            90.000
 0050                      IF (FLAG.EQ.ODD)     GO TO 50                                      91.000
               C      AT THIS POINT - NEED TO ADD CELL                                        92.000
 0051                      GO TO 450                                                          93.000
 0052          400         STUFF = ADVLNR(READER,FLAG)                                        94.000
               C                  *******************          LOOK FOR NAME CELLS            95.000
 0053                      IF (FLAG.NE.ODD)     GO TO 450                                     96.000
 0054                      GO TO 50                                                           97.000
 0055          450         PARENT = A(LVL)                                                    98.000
 0056                      PCOUNT = 0                                                         99.000
 0057                      NAME = LOFRDR(READER)                                             100.000
               C                 *************             RETURNS NAME OF LIST SCANNED      101.000
 0058                      NEWLST = LIST(9)                                                  102.000
               C                   *********               GET NEW SUBLIST - REF COUNT = 0   103.000
 0059                      CALL NEWTOP(STUFF,NEWLST)                                         104.000
               C                ********************       INSERT PARENT ID ON SUBLIST       105.000
 0060                      CALL NEWBOT(NEWLST,NAME)                                          106.000
               C                *******************        LINK LISTS                        107.000
 0061                      STUFF = ADVLNR(READER,FLAG)                                       108.000
               C                  *******************      POSITION ON NEW NAME CELL         109.000
 0062                      GO TO 50                                                          110.000
 0063          500         WRITE (6,990)                                                     111.000
 0064                      CALL SLPDMP(SPACE,M)                                              112.000
               C                 ***************           DUMP LIST SRUCTURE                113.000
```

```
MICHIGAN TERMINAL SYSTEM FORTRAN G(41336)              MAIN           06-22-72        18:43.51        PAGE P003

                C     TIME TO DISPLAY CONTENTS OF TREE                                    114.000
  0065                    WRITE (6,980)                                                   115.000
  0066          520       STUFF = ADVSER(READER,FLAG)                                     116.000
                C                 ******************     SCAN FOR SUBLIST IDENTIFIERS     117.000
  0067                    IF (FLAG.NE.ODD)     GO TO  700                                 118.000
  0068                    LVL     = LCNTR(READER)                                         119.000
                C                   *************        GET LEVEL OF STRUCTURE           120.000
  0069                    A(LVL)     = PARENT                                             121.000
  0070                    IF (LVL.EQ.4)        GO TO 550                                  122.000
  0071                    J = LVL + 1                                                     123.000
  0072                    DO 540  I=J,4                                                   124.000
  0073          540       A(I) = BLANK                                                    125.000
  0074          550       IF (PCOUNT - 0)  520,520,575                                    126.000
  0075          575       WRITE (6,920)  (A(I),I=1,4)                                     127.000
  0076                    GO TO 520                                                       128.000
  0077          700       CALL IPARDR(READER)                                             129.000
                C               *************            ERASE THE READER                 130.000
  0078                    CALL IRALST(MAIN)                                               131.000
                C               ************             ERASE THE LIST                   132.000
  0079                    GO TO 10                                                        133.000
  0080          800       WRITE (6,940)                                                   134.000
  0081                    STOP                                                            135.000
  0082                    END                                                             136.000
   *OPTIONS IN EFFECT*  ID,EBCDIC,SOURCE,NOLIST,NODECK,LOAD,NOMAP
   *OPTIONS IN EFFECT*  NAME = MAIN     , LINECNT =       57
   *STATISTICS*    SOURCE STATEMENTS =        82,PROGRAM SIZE =    10302
   *STATISTICS*  NO DIAGNOSTICS GENERATED
 NO ERRORS IN MAIN

 NO STATEMENTS FLAGGED IN THE ABOVE COMPILATIONS.
```

```
HERE ARE THE INPUT STRINGS

A

ABD

ASRT

B

C

CC

CCC

CCCB

CCCC

CCCD

DEAN

DX

ZZZZ

•
```

```
****** SLIP DUMP ******


AVAILABLE SLIP STORAGE


FIRST AVAILABLE CELL IS 005036E8    LAST AVAILABLE CELL IS 005034B8

ADDRS   TYPE  LINK WORD               DATUM

              ID LNKL    M LNKR       HEXADECIMAL        EBCDIC    REAL                           INTEGERS              ID LNKL    M LNKR

5036F8 RDR     3 503568 0 503728      000A065500000000                0.3381847532686294D-78        656981              0 0 5032A8 0 000000
503728 RDR     3 503668 0 503648      000A065500000000                0.3381847532686294D-78        656981              0 0 5032A8 0 000000
503648 DAT     0 503638 0 503358      E940404000000000 Z             -0.5868931715387703D 49   -381665216              0 7 020200 0 000000
503358 RDR     3 503658 0 5035D8      000AC6AB00000003                0.3382290222638992D-78        657067              3 0 503558 0 000018
5035D8 RDR     3 503578 0 503528      000A069F00000002                0.3382228451947917D-78        657055              2 0 5034F8 0 000010
503528 RDR     3 5035B8 0 5034B8      000A06D900000001    R           0.3382527010288105D-78        657113              1 0 5036C8 0 000008
5034B8 RDR    '3 503418 0 000000      000A065500000000                0.3381847532686294D-78        656981              0 0 5032A8 0 000000

READERS

ADDRS   TYPE  LINK WORD               DATUM

              ID LNKL    M LNKR       HEXADECIMAL        EBCDIC    REAL                           INTEGERS              ID LNKL    M LNKR

5032B8 RDR     3 5032A8 0 000000      000A065500000000                0.3381847532686294D-78        656981              0 0 5032A8 0 000000

LISTS

ADDRS   TYPE  LINK WORD               DATUM

              ID LNKL    M LNKR       HEXADECIMAL        EBCDIC    REAL                           INTEGERS              ID LNKL    M LNKR

5032A8 HDR     2 503418 0 5032F8      0000000000000001                0.1198509146801202D-93             0            1 0 000000 0 000008
5032F8 NAM     1 5032A8 0 5034E8      200A0659C00A0659                0.1150791403465799D-39    537527897-1073084839 1 5032C8 6 5032C8
5034E8 NAM     1 5032F8 0 503568      200A0699C00A0699    R      R    0.1150903507342971D-39    537527961-1073084775 1 5034C8 6 5034C8
503568 NAM     1 5034E8 0 503668      200AC6A9C00AC6A9    Z      Z    0.1150931533312264D-39    537527977-1073084759 1 503548 6 503548
503668 NAM     1 503568 0 503418      200A06D1C00A06D1    J      J    0.1151001598235496D-39    537528017-1073084719 1 503688 6 503688
503418 NAM     1 503668 0 5032A8      200A06D9C00A06D9    R      R    0.1151015611220143D-39    537528025-1073084711 1 5036C8 6 5036C8

5032C8 HDR     2 5033F8 0 503308      0000000000000001                0.1198509146801202D-93             0            1 0 000000 0 000008
503308 DAT     0 5032C8 0 503348      C140404000000001 A             -0.4015686035156249D 01  -1052753856            1 6 020200 0 000008
503348 NAM     1 503308 0 5033F8      200A0665C00A0665                0.1150812422942769D-39    537527909-1073084827 1 503328 6 503328
5033F8 NAM     1 503348 0 5032C8      200A067BC00A067B    #      #    0.1150850958650546D-39    537527931-1073084805 1 5033D8 6 5033D8
```

```
5032F8 HDR   2 503318 0 503378     0000000000000001         0.1198509146801202D-93            0           1 0 000000 0 000008
503378 DAT   0 5032F8 0 503318     C140404000000000 A      -0.4015686035156249D 01 -1052753856            0 6 020200 0 000000
503318 NAM   1 503378 0 5032F8     200A068FC00A068F         0.1150885991112163D-39   537527951-1073084785 1 503478 6 503478

503328 HDR   2 503388 0 503338     0000000000000001         0.1198509146801202D-93            0           1 0 000000 0 000008
503338 DAT   0 503328 0 503388     C240404000000000 B      -0.6425097656250000D 02 -1035976640            0 6 020200 0 000000
503388 NAM   1 503338 0 503328     200A066DC00A066D    -  - 0.1150826435927415D-39   537527917-1073084819 1 503368 6 503368

503368 HDR   2 5033A8 0 5033A8     0000000000000001         0.1198509146801202D-93            0           1 0 000000 0 000008
5033A8 DAT   0 503368 0 503368     C440404000000001 D      -0.1644824999999999D 05 -1002422208            1 6 020200 0 000008

5033D8 HDR   2 503448 0 5033E8     0000000000000001         0.1198509146801202D-93            0           1 0 000000 0 000008
5033E8 DAT   0 5033D8 0 503448     E240404000000000 S      -0.2186347438166925D 41  -499105728            0 7 020200 0 000000
503448 NAM   1 5033E8 0 5033D8     200A0685C00A0685    E  E 0.1150868474881354D-39   537527941-1073084795 1 503428 6 503428

503408 HDR   2 5033B8 0 5033B8     0000000000000001         0.1198509146801202D-93            0           1 0 000000 0 000008
5033B8 DAT   0 503408 0 503408     C340404000000001 C      -0.1028015624999999D 04 -1019199424            1 6 020200 0 000008

503428 HDR   2 503488 0 503438     0000000000000001         0.1198509146801202D-93            0           1 0 000000 0 000008
503438 DAT   0 503428 0 503488     D940404000000000 R      -0.3181554257996212D 30  -650100672            0 6 020200 0 000000
503488 NAM   1 503438 0 503428     200A068DC00A068D         0.1150882487866001D-39   537527949-1073084787 1 503468 6 503468

503468 HDR   2 5034A8 0 5034A8     0000000000000001         0.1198509146801202D-93            0           1 0 000000 0 000008
5034A8 DAT   0 503468 0 503468     E340404000000001 T      -0.3498155901067080D 42  -482328512            1 7 020200 0 000008

503478 HDR   2 503398 0 503398     0000000000000001         0.1198509146801202D-93            0           1 0 000000 0 000008
503398 DAT   0 503478 0 503478     D540404000000001 N      -0.4854666531366289D 25  -717209536            1 6 020200 0 000008

5034C8 HDR   2 503518 0 503518     0000000000000001         0.1198509146801202D-93            0           1 0 000000 0 000008
503518 DAT   0 5034C8 0 5034C8     C240404000000001 B      -0.6425097656250000D 02 -1035976640            1 6 020200 0 000008

5034F8 HDR   2 503578 0 503538     0000000000000001         0.1198509146801202D-93            0           1 0 000000 0 000008
503538 DAT   0 5034F8 0 503578     E940404000000000 Z      -0.5868931715387703D 49  -381665216            0 7 020200 0 000000
503578 NAM   1 503538 0 5034F8     200A06ABC00A06AB         0.1150935036558425D-39   537527979-1073084757 1 503558 6 503558

503548 HDR   2 503608 0 5035A8     0000000000000001         0.1198509146801202D-93            0           1 0 000000 0 000008
5035A8 DAT   0 503548 0 503608     C340404000000001 C      -0.1028015624999999D 04 -1019199424            1 6 020200 0 000008
503608 NAM   1 5035A8 0 503548     200A06BDC00A06BD         0.1150966565773880D-39   537527997-1073084739 1 5035E8 6 5035E8

503558 HDR   2 503658 0 503508     0000000000000001         0.1198509146801202D-93            0           1 0 000000 0 000008
503508 DAT   0 503558 0 503658     E940404000000000 Z      -0.5868931715387703D 49  -381665216            0 7 020200 0 000000
503658 NAM   1 503508 0 503558     200A06C7C00A06C7    G  G 0.1150984082004688D-39   537528007-1073084729 1 503638 6 503638

503588 HDR   2 503618 0 503618     0000000000000001         0.1198509146801202D-93            0           1 0 000000 0 000008
503618 DAT   0 503588 0 503588     C440404000000001 D      -0.1644824999999999D 05 -1002422208            1 6 020200 0 000008
```

```
[illegible]                                      [illegible]                        -0.2292[illegible]D 41  -415219648               1 7 02020C 0 000008

5035F8 HDR   2 503698 0 503628     0000000000000001               0.11985091468012020-93            0            1 0 000000 0 000008
503628 DAT   0 5035E8 0 503698     C340404000000001 C            -0.10280156249999990 04 -1019199424            1 6 02020C 0 C00008
503698 NAM   1 503628 0 5035E8     200A06CFCC0A06CF               0.11509980949893350-39   537528015-1073084721 1 503678 6 503678

503638 HDR   2 503458 0 503458     0000000000000001               0.11985091468012020-93            0            1 0 000000 0 000008
503458 DAT   0 503638 0 503638     F940404000000001 Z            -0.58689317153877030 49   -381665216            1 7 02020C 0 C00008

503678 HDR   2 503598 0 503688     0000000000000001               0.11985091468012020-93            0            1 0 000000 0 000008
503688 DAT   0 503678 0 503738     C340404000000001 C            -0.10280156249999990 04 -1019199424            1 6 020200 0 000008
503738 NAM   1 503688 0 503498     200A06E3C00AC6F3      T    T   0.11510331274509510-39   537528035-1073084701 1 503718 6 503718
503498 NAM   1 503738 0 503598     200A0681C00A0681      A    A   0.11508614683890310-39   537527937-1073084799 1 503408 6 503408
503598 NAM   1 503498 0 503678     200A06B1C00A06B1      1    1   0.11509455462969100-39   537527985-1073084751 1 503588 6 503588

503688 HDR   2 5035F8 0 5036A8     0000000000000001               0.11985091468012020-93            0            1 0 000000 0 000008
5036A8 DAT   0 503688 0 5036F8     C440404000000000 D            -0.16448249999999990 05 -1002422208            0 6 020200 0 000000
5036F8 NAM   1 5036A8 0 5035F8     200A06E9C00A06E9      Z    Z   0.11510436371894360-39   537528041-1073084695 1 503748 6 503748
5035F8 NAM   1 5036F8 0 503688     200A06B9C00A06B9      9    9   0.11509595592815570-39   537527993-1073084743 1 5035C8 6 5035C8

5036C8 HDR   2 5035B8 0 5036D8     0000000000000001               0.11985091468012020-93            0            1 0 000000 0 000008
5036D8 DAT   0 5036C8 0 5035B8     F940404000000000 Z            -0.58689317153877030 49   -381665216            0 7 02020C 0 000000
5035B8 NAM   1 5036D8 0 5036C8     200A069FC00A069F               0.11509140170814560-39   537527967-1073084769 1 5034F8 6 5034F8

503718 HDR   2 5032D8 0 5032D8     0000000000000001               0.11985091468012020-93            0            1 0 000000 0 000008
5032D8 DAT   0 503718 0 503718     C240404000000001 B            -0.64250976562500000 02 -1035976640            1 6 02020C 0 C00008

503748 HDR   2 5033C8 0 503708     0000000000000001               0.11985091468012020-93            0            1 0 000000 0 000008
503708 DAT   0 503748 0 5033C8     C540404000000000 E            -0.26317199999999990 06   -985644992            0 6 020200 0 000000
5033C8 NAM   1 503708 0 503748     200A065FC00A065F      ¬    ¬   0.11508019132042830-39   537527903-1073084833 1 5032F8 6 5032F8


PUBLIC LISTS

ADDRS  TYPE  LINK WORD             DATUM

             ID LNKL    M LNKR     HEXADECIMAL       EBCDIC   REAL                              INTEGERS             ID LNKL    M LNKR

506388 HDR   2 506388 0 506388     0000000000000FFF               0.49078949561509230-90            0        4095 0 000000 0 007FF8

506398 HDR   2 506398 0 506398     0000000000000FFF               0.49078949561509230-90            0        4095 0 000000 0 007FF8

5063A8 HDR   2 5063A8 0 5063A8     0000000000000FFF               0.49078949561509230-90            0        4095 0 000000 0 007FF8

5063B8 HDR   2 5063B8 0 5063B8     0000000000000FFF               0.49078949561509230-90            0        4095 0 000000 0 007FF8
```

```
5063C8 HDR   2 5063C8 0 5063C8    0000000000000FFF    0.4907894956150923D-90    0    4095 0 000000 0 007FF8
5063D8 HDR   2 5063D8 0 5063D8    0000000000000FFF    0.4907894956150923D-90    0    4095 0 000000 0 007FF8
5063E8 HDR   2 5063E8 0 5063E8    0000000000000FFF    0.4907894956150923D-90    0    4095 0 000000 0 007FF8
5063F8 HDR   2 5063F8 0 5063F8    0000000000000FFF    0.4907894956150923D-90    0    4095 0 000000 0 007FF8
506408 HDR   2 506408 0 506408    0000000000000FFF    0.4907894956150923D-90    0    4095 0 000000 0 007FF8
506418 HDR   2 506418 0 506418    0000000000000FFF    0.4907894956150923D-90    0    4095 0 000000 0 007FF8
506428 HDR   2 506428 0 506428    0000000000000FFF    0.4907894956150923D-90    0    4095 0 000000 0 007FF8
506438 HDR   2 506438 0 506438    0000000000000FFF    0.4907894956150923D-90    0    4095 0 000000 0 007FF8
506448 HDR   2 506448 0 506448    0000000000000FFF    0.4907894956150923D-90    0    4095 0 000000 0 007FF8
506458 HDR   2 506458 0 506458    0000000000000FFF    0.4907894956150923D-90    0    4095 0 000000 0 007FF8
506468 HDR   2 506468 0 506468    0000000000000FFF    0.4907894956150923D-90    0    4095 0 000000 0 007FF8
506478 HDR   2 506478 0 506478    0000000000000FFF    0.4907894956150923D-90    0    4095 0 000000 0 007FF8
506488 HDR   2 506488 0 506488    0000000000000FFF    0.4907894956150923D-90    0    4095 0 000000 0 007FF8
506498 HDR   2 506498 0 506498    0000000000000FFF    0.4907894956150923D-90    0    4095 0 000000 0 007FF8
5064A8 HDR   2 5064A8 0 5064A8    0000000000000FFF    0.4907894956150923D-90    0    4095 0 000000 0 007FF8
5064B8 HDR   2 5064B8 0 5064B8    0000000000000FFF    0.4907894956150923D-90    0    4095 0 000000 0 007FF8
5064C8 HDR   2 5064C8 0 5064C8    0000000000000FFF    0.4907894956150923D-90    0    4095 0 000000 0 007FF8
5064D8 HDR   2 5064D8 0 5064D8    0000000000000FFF    0.4907894956150923D-90    0    4095 0 000000 0 007FF8
5064F8 HDR   2 5064E8 0 5064E8    0000000000000FFF    0.4907894956150923D-90    0    4095 0 000000 0 007FF8
5064F8 HDR   2 5064F8 0 5064F8    0000000000000FFF    0.4907894956150923D-90    0    4095 0 000000 0 007FF8
```

```
506518 HDR    2 506518 0 506518     0000000000000FFF     0.4907894956150923D-90     0
506528 HDR    2 506528 0 506528     0000000000000FFF     0.4907894956150923D-90     0     4095 0 000000 0 007FF8
506538 HDR    2 506538 0 506538     0000000000000FFF     0.4907894956150923D-90     0     4095 0 000000 0 007FF8
506548 HDR    2 506548 0 506548     0000000000000FFF     0.4907894956150923D-90     0     4095 0 000000 0 007FF8
506558 HDR    2 506558 0 506558     0000000000000FFF     0.4907894956150923D-90     0     4095 0 000000 0 007FF8
506568 HDR    2 506568 0 506568     0000000000000FFF     0.4907894956150923D-90     0     4095 0 000000 0 007FF8
506578 HDR    2 506578 0 506578     0000000000000FFF     0.4907894956150923D-90     0     4095 0 000000 0 007FF8
506588 HDR    2 506588 0 506588     0000000000000FFF     0.4907894956150923D-90     0     4095 0 000000 0 007FF8
506598 HDR    2 506598 0 506598     0000000000000FFF     0.4907894956150923D-90     0     4095 0 000000 0 007FF8
5065A8 HDR    2 5065A8 0 5065A8     0000000000000FFF     0.4907894956150923D-90     0     4095 0 000000 0 007FF8
5065B8 HDR    2 5065B8 0 5065B8     0000000000000FFF     0.4907894956150923D-90     0     4095 0 000000 0 007FF8
5065C8 HDR    2 5065C8 0 5065C8     0000000000000FFF     0.4907894956150923D-90     0     4095 0 000000 0 007FF8
5065D8 HDR    2 5065D8 0 5065D8     0000000000000FFF     0.4907894956150923D-90     0     4095 0 000000 0 007FF8
5065F8 HDR    2 5065E8 0 5065E8     0000000000000FFF     0.4907894956150923D-90     0     4095 0 000000 0 007FF8
5065F8 HDR    2 5065F8 0 5065F8     0000000000000FFF     0.4907894956150923D-90     0     4095 0 000000 0 007FF8
506608 HDR    2 506608 0 506608     0000000000000FFF     0.4907894956150923D-90     0     4095 0 000000 0 007FF8
506618 HDR    2 506618 0 506618     0000000000000FFF     0.4907894956150923D-90     0     4095 0 000000 0 007FF8
506628 HDR    2 506628 0 506628     0000000000000FFF     0.4907894956150923D-90     0     4095 0 000000 0 007FF
506638 HDR    2 506638 0 506638     0000000000000FFF     0.4907894956150923D-90     0     4095 0 000000 0 007F
```

```
506648 HDR   2 506648 0 506648      0000000000000FFF         0.4907894956150923D-90        0      4095 0 000000 0 007FF8

506658 HDR   2 506658 0 506658      0000000000000FFF         0.4907894956150923D-90        0      4095 0 000000 0 007FF8

506668 HDR   2 506668 0 506668      0000000000000FFF         0.4907894956150923D-90        0      4095 0 000000 0 007FF8

506678 HDR   2 506678 0 506678      0000000000000FFF         0.4907894956150923D-90        0      4095 0 000000 0 007FF8

506688 HDR   2 506688 0 506688      0000000000000FFF         0.4907894956150923D-90        0      4095 0 000000 0 007FF8

506698 HDR   2 506698 0 506698      0000000000000FFF         0.4907894956150923D-90        0      4095 0 000000 0 007FF8

5066A8 HDR   2 5066A8 0 5066A8      0000000000000FFF         0.4907894956150923D-90        0      4095 0 000000 0 007FF8

5066B8 HDR   2 5066B8 0 5066B8      0000000000000FFF         0.4907894956150923D-90        0      4095 0 000000 0 007FF8

5066C8 HDR   2 5066C8 0 5066C8      0000000000000FFF         0.4907894956150923D-90        0      4095 0 000000 0 007FF8

5066D8 HDR   2 5066D8 0 5066D8      0000000000000FFF         0.4907894956150923D-90        0      4095 0 000000 0 007FF8

5066E8 HDR   2 5066E8 0 5066E8      0000000000000FFF         0.4907894956150923D-90        0      4095 0 000000 0 007FF8

5066F8 HDR   2 5066F8 0 5066F8      0000000000000FFF         0.4907894956150923D-90        0      4095 0 000000 0 007FF8

506708 HDR   2 506708 0 506708      0000000000000FFF         0.4907894956150923D-90        0      4095 0 000000 0 007FF8

506718 HDR   2 506718 0 506718      0000000000000FFF         0.4907894956150923D-90        0      4095 0 000000 0 007FF8

506728 HDR   2 506728 0 506728      0000000000000FFF         0.4907894956150923D-90        0      4095 0 000000 0 007FF8

506738 HDR   2 506738 0 506738      0000000000000FFF         0.4907894956150923D-90        0      4095 0 000000 0 007FF8

506748 HDR   2 506748 0 506748      0000000000000FFF         0.4907894956150923D-90        0      4095 0 000000 0 007FF8

506758 HDR   2 506758 0 506758      0000000000000FFF         0.4907894956150923D-90        0      4095 0 000000 0 007FF8

506768 HDR   2 506768 0 506768      0000000000000FFF         0.4907894956150923D-90        0      4095 0 000000 0 007FF8

506778 HDR   2 506778 0 506778      0000000000000FFF         0.4907894956150923D-90        0      4095 0 000000 0 007FF8
```

```
506788 HDR    2 506788 0 506788        0000000000000FFF
506798 HDR    2 506798 0 506798        0000000000000FFF        0.4907894956150923D-90        0        4095 0 000000 0 007FF8
5067A8 HDR    2 5067A8 0 5067A8        0000000000000FFF        0.4907894956150923D-90        0        4095 0 000000 0 007FF8
5067B8 HDR    2 5067B8 0 5067B8        0000000000000FFF        0.4907894956150923D-90        0        4095 0 000000 0 007FF8
5067C8 HDR    2 5067C8 0 5067C8        0000000000000FFF        0.4907894956150923D-90        0        4095 0 000000 0 007FF8
5067D8 HDR    2 5067D8 0 5067D8        0000000000000FFF        0.4907894956150923D-90        0        4095 0 000000 0 007FF8
5067E8 HDR    2 5067E8 0 5067E8        0000000000000FFF        0.4907894956150923D-90        0        4095 0 000000 0 007FF8
5067F8 HDR    2 5067F8 0 5067F8        0000000000000FFF        0.4907894956150923D-90        0        4095 0 000000 0 007FF8
506808 HDR    2 506808 0 506808        0000000000000FFF        0.4907894956150923D-90        0        4095 0 000000 0 007FF8
506818 HDR    2 506818 0 506818        0000000000000FFF        0.4907894956150923D-90        0        4095 0 000000 0 007FF8
506828 HDR    2 506828 0 506828        0000000000000FFF        0.4907894956150923D-90        0        4095 0 000000 0 007FF8
506838 HDR    2 506838 0 506838        0000000000000FFF        0.4907894956150923D-90        0        4095 0 000000 0 007FF8
506848 HDR    2 506848 0 506848        0000000000000FFF        0.4907894956150923D-90        0        4095 0 000000 0 007FF8
506858 HDR    2 506858 0 506858        0000000000000FFF        0.4907894956150923D-90        0        4095 0 000000 0 007FF8
506868 HDR    2 506868 0 506868        0000000000000FFF        0.4907894956150923D-90        0        4095 0 000000 0 007FF8
506878 HDR    2 506878 0 506878        0000000000000FFF        0.4907894956150923D-90        0        4095 0 000000 0 007FF8
506888 HDR    2 506888 0 506888        0000000000000FFF        0.4907894956150923D-90        0        4095 0 000000 0 007FF8
506898 HDR    2 506898 0 506898        0000000000000FFF        0.4907894956150923D-90        0        4095 0 000000 0 007FF8
5068A8 HDR    2 5068A8 0 5068A8        0000000000000FFF        0.4907894956150923D-90        0        4095 0 000000 0 007FF8
5068B8 HDR    2 5068B8 0 5068B8        0000000000000FFF        0.4907894956150923D-90        0        4095 0 000000 0 007FF8
```

```
5068C8 HDR   2 5068C8 0 5068C8      000000000000FFF      0.4907894956150923D-90       0      4095 0 000000 0 007FF8

5068D8 HDR   2 5068D8 0 5068D8      000000000000FFF      0.4907894956150923D-90       0      4095 0 000000 0 007FF8

5068E8 HDR   2 5068E8 0 5068E8      000000000000FFF      0.4907894956150923D-90       0      4095 0 000000 0 007FF8

5068F8 HDR   2 5068F8 0 5068F8      000000000000FFF      0.4907894956150923D-90       0      4095 0 000000 0 007FF8

506908 HDR   2 506908 0 506908      000000000000FFF      0.4907894956150923D-90       0      4095 0 000000 0 007FF8

506918 HDR   2 506918 0 506918      000000000000FFF      0.4907894956150923D-90       0      4095 0 000000 0 007FF8

506928 HDR   2 506928 0 506928      000000000000FFF      0.4907894956150923D-90       0      4095 0 000000 0 007FF8

506938 HDR   2 506938 0 506938      000000000000FFF      0.4907894956150923D-90       0      4095 0 000000 0 007FF8

506948 HDR   2 506948 0 506948      000000000000FFF      0.4907894956150923D-90       0      4095 0 000000 0 007FF8

506958 HDR   2 506958 0 506958      000000000000FFF      0.4907894956150923D-90       0      4095 0 000000 0 007FF8

506968 HDR   2 506968 0 506968      000000000000FFF      0.4907894956150923D-90       0      4095 0 000000 0 007FF8

506978 HDR   2 506978 0 506978      000000000000FFF      0.4907894956150923D-90       0      4095 0 000000 0 007FF8

506988 HDR   2 506988 0 506988      000000000000FFF      0.4907894956150923D-90       0      4095 0 000000 0 007FF8

506998 HDR   2 506998 0 506998      000000000000FFF      0.4907894956150923D-90       0      4095 0 000000 0 007FF8

5069A8 HDR   2 5069A8 0 5069A8      000000000000FFF      0.4907894956150923D-90       0      4095 0 000000 0 007FF8

5069B8 HDR   2 5069B8 0 5069B8      000000000000FFF      0.4907894956150923D-90       0      4095 0 000000 0 007FF8


FREE CELLS

**NO FREE CELLS IN USE**
```

```
HERE IS THE GENERATED TREE

A

A   B   D

A   S   R   T

B

C

C   C

C   C   C

C   C   C   B

C   C   C   C

C   C   C   D

D   F   A   N

D   X

Z   Z   Z   Z
```

```
HERE ARE THE INPUT STRINGS

ZZZZ

OX

WE ARE ABOUT TO CRASH
USER PROGRAM INTERRUPT.   PSW = 07150006 7050036A

 PSW     =  07150006 7050036A

         GENERAL REGISTERS 0...15
                 200A06B5    00500B1C    70500B72    00500BAE    00503030    00000000    00000000    005050
                 00000000    00000000    00500A90    005051E8    005051E8    00500A90    70500BBC    005003

         FLOATING POINT REGISTERS
                 40404040 40404040      400A06D3 000A06C5      00C00000 00000000      00000000 00000000

NEXT CARD IS
```

PL/I (J. Unger)

The following program reads sets of input character strings, sequences them by constructing a binary tree based on their relative alphabetic order, and then prints the ordered sets of input strings.

The input data is read using stream I/O from SCARDS (default system input). A variable number of blanks are used to separate individual character strings within a set, while a set delimiter of '*' denotes the end of a set of strings. The end of all the input sets is recognized as the PL/1 ENDFILE condition.

Each string within an input set is extracted using the SUBSTR function after the position of the first character and the number of characters within the string have been determined using DO...WHILE... and the INDEX function. A node representing the extracted string is then allocated and initialized.

A new node is added to the tree using the following algorithm:

1. If the TREE_BASE is NULL, this node becomes the top of a new tree. Otherwise, the base pointer TREE is set to point to the top of the tree.
2. The string associated with the new node is compared to the string in the tree node.
3. If the new value is greater, the P_RIGHT pointer is followed.
   a. If P_RIGHT is NULL, i.e., there is no other node with a value greater than the tree node's value, the new node is inserted at this point.
   b. If P_RIGHT is not NULL, then there is at least one node with a higher value, TREE is set to point to the next node to the right, and steps 2 and 3 or 4 are repeated.
4. If the new node value is less than the tree node value, then the P_LEFT pointer is followed.
   a. If P_LEFT is NULL, i.e., there is no tree node with a a value less than the current tree node's value, the new node is added at this point

b. If P_LEFT is not NULL, there is at least one tree node with a value less than the current tree node's value, and TREE is set to point to the next node to the left and steps 2 and 3 or 4 are repeated.

Once all strings have been extracted and added to the tree, the tree node values are printed out in alphabetical order. This is accomplished by starting at the top tree node and using the following algorithm:

1. Follow the P_LEFT pointers until a node with P_LEFT = NULL is encountered. This contains the (next) lowest sequenced value and is to be printed. A flag is set to '1' to indicate that this node's contents have already been printed.

2. The P_RIGHT pointer associated with the node just printed is examined. If it is not NULL, TREE is set to point to the next node to the right, and the process starting with 1 is begun again. If P_RIGHT is NULL, step 3 occurs.

3. The parent of the current tree node is examined. If a back trace of a left sub-tree is occurring, the parent node has not yet been printed (PRINT=0), so it is printed and flagged and the process starting with 2 is begun again. If a back trace of a right sub-tree is occurring, the parent node has already been printed (PRINT=1), so the process beginning with step 3 is repeated. The top node of the tree is recognized by its NULL P_PARENT pointer.

```
STMT LEVEL NEST
  1              SAMPLE_PROG:  PROCEDURE OPTIONS (MAIN);
  2     1          DECLARE
                     INPUT CHAR (80),
                     1 NODE BASED (TREE),
                       2 P_LEFT   POINTER,
                       2 P_RIGHT  POINTER,
                       2 P_PARENT POINTER,
                       2 VALUE    CHAR (4),
                       2 PRINT    CHAR(1),
                     I  FIXED BINARY (15,0),
                     J  FIXED BINARY (15,0),
                   M  FIXED BINARY (15,0),
                     TREE_BASE POINTER,
                     NEW       POINTER;
                 /*
  3     1        ON ENDFILE GO TO END_OF_JOB;
                 /*
  5     1        GET_NEXT:
                   I = 1;
  6     1          J = 0;
  7     1          TREE_BASE = NULL;
  8     1          GET EDIT (INPUT) (A(80));
                 /*
  9     1        EXTRACT_STRING:
                   M = I + J;
 10     1          DO I = M TO 80 WHILE (SUBSTR(INPUT,I,1) = ' ');  END;
 12     1          IF SUBSTR(INPUT,I,1) = '*' THEN GO TO PRINT_ROUTINE;
 14     1          J = INDEX (SUBSTR(INPUT,I+1),' ');
 15     1          IF J = 0 THEN J = INDEX (SUBSTR(INPUT,I+1),'*');
                 /*
 17     1        BUILD_NODE:
                   ALLOCATE NODE SET (NEW);
 18     1          NEW -> P_LEFT = NULL;
 19     1          NEW -> P_RIGHT = NULL;
 20     1          NEW -> P_PARENT = NULL;
 21     1          NEW -> PRINT = '0';
 22     1          NEW -> VALUE = SUBSTR(INPUT,I,J);
                 /*
 23     1        ADD_TO_TREE:
 24     1          IF TREE_BASE = NULL THEN DO;
 25     1   1        TREE_BASE = NEW;
 26     1   1        GO TO EXTRACT_STRING;
 27     1   1        END;
 28     1          TREE = TREE_BASE;
 29     1        COMPARE_VALUE:
 30     1          IF NEW -> VALUE = VALUE THEN GO TO EXTRACT_STRING;
 31     1          IF NEW -> VALUE > VALUE
 32     1            THEN DO;
 33     1   1          IF P_RIGHT = NULL
 34     1   1            THEN DO;
 35     1   2              P_RIGHT = NEW;
 36     1   2              NEW -> P_PARENT = TREE;
 37     1   2              GO TO EXTRACT_STRING;
 38     1   2              END;
 39     1   1            ELSE DO;
 40     1   2              TREE = P_RIGHT;
```

```
STMT LEVEL NEST

 41     1    2                GO TO COMPARE_VALUE;
 42     1    2                END;
 43     1    1            END;
 44     1               ELSE DO;
 45     1    1            IF P_LEFT = NULL
 46     1    1              THEN DO;
 47     1    2                P_LEFT = NEW;
 48     1    2                NEW -> P_PARENT = TREE;
 49     1    2                GO TO EXTRACT_STRING;
 50     1    2                END;
 51     1    1              ELSE DO;
 52     1    2                TREE = P_LEFT;
 53     1    2                GO TO COMPARE_VALUE;
 54     1    2                END;
 55     1    1            END;
                  /*
 56     1         PRINT_ROUTINE:
                    TREE = TREE_BASE;
 57     1         CHECK_P_LEFT:
                    IF P_LEFT = NULL
 58     1             THEN DO;
 59     1    1          PUT EDIT (VALUE) (A(4));
 60     1    1          PRINT = '1';
 61     1    1        CHECK_P_RIGHT:
                        IF P_RIGHT = NULL
 62     1    1            THEN DO;
 63     1    2            CHECK_PARENT:
                            IF P_PARENT = NULL
 64     1    2                THEN IF PRINT = '1'
 65     1    2                  THEN GO TO GET_NEXT;
 66     1    2                  ELSE GO TO PRINT_VALUE;
 67     1    2                ELSE DO;
 68     1    3                  TREE = P_PARENT;
 69     1    3                  IF PRINT = '1'
 70     1    3                    THEN GO TO CHECK_PARENT;
 71     1    3                    ELSE DO;
 72     1    4                    PRINT_VALUE:
                                    PUT EDIT (VALUE) (A(4));
 73     1    4                      PRINT = '1';
 74     1    4                      GO TO CHECK_P_RIGHT;
 75     1    4                      END;
 76     1    3                  END;
 77     1    2              END;
 78     1    1            ELSE DO;
 79     1    2              TREE = P_RIGHT;
 80     1    2              GO TO CHECK_P_LEFT;
 81     1    2              END;
 82     1    1          END;
 83     1             ELSE DO;
 84     1    1          TREE = P_LEFT;
 85     1    1          GO TO CHECK_P_LEFT;
 86     1    1          END;
                  /*
 87     1         END_OF_JOB:
```

```
STMT LEVEL NEST

                          RETURN;
 88      1                END SAMPLE_PROG;
```

```
    INPUT #1   ORDERED                       INPUT #2  ORDERED                       INPUT #3  ORDERED
<------------------------------------->  <------------------------------------->  <------------------------------------->
AA  AB  PA  C   CC  CCC CCCCDD  X        AA  AB  BA  C   CC  CCC CCCCDD  X        A   AB  BA  C   CC  CCC CCCCDD  X
EXECUTION TERMINATED


INPUT #1  AA  AB  BA C   CC   CCC   CCCC DD   X   *

INPUT #2  X  DD  CCCC  [illegible]  *

INPUT #3  C   A   X   [illegible]  CC   [illegible]   CCCC DD   CCC   *
```

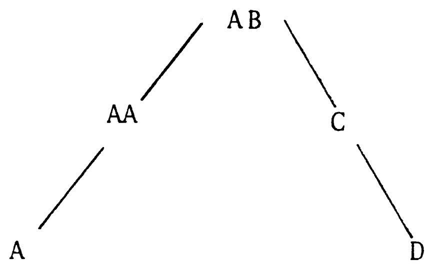
LISP (Carole Hafner)

LISP Program to Alphabetize Character Strings

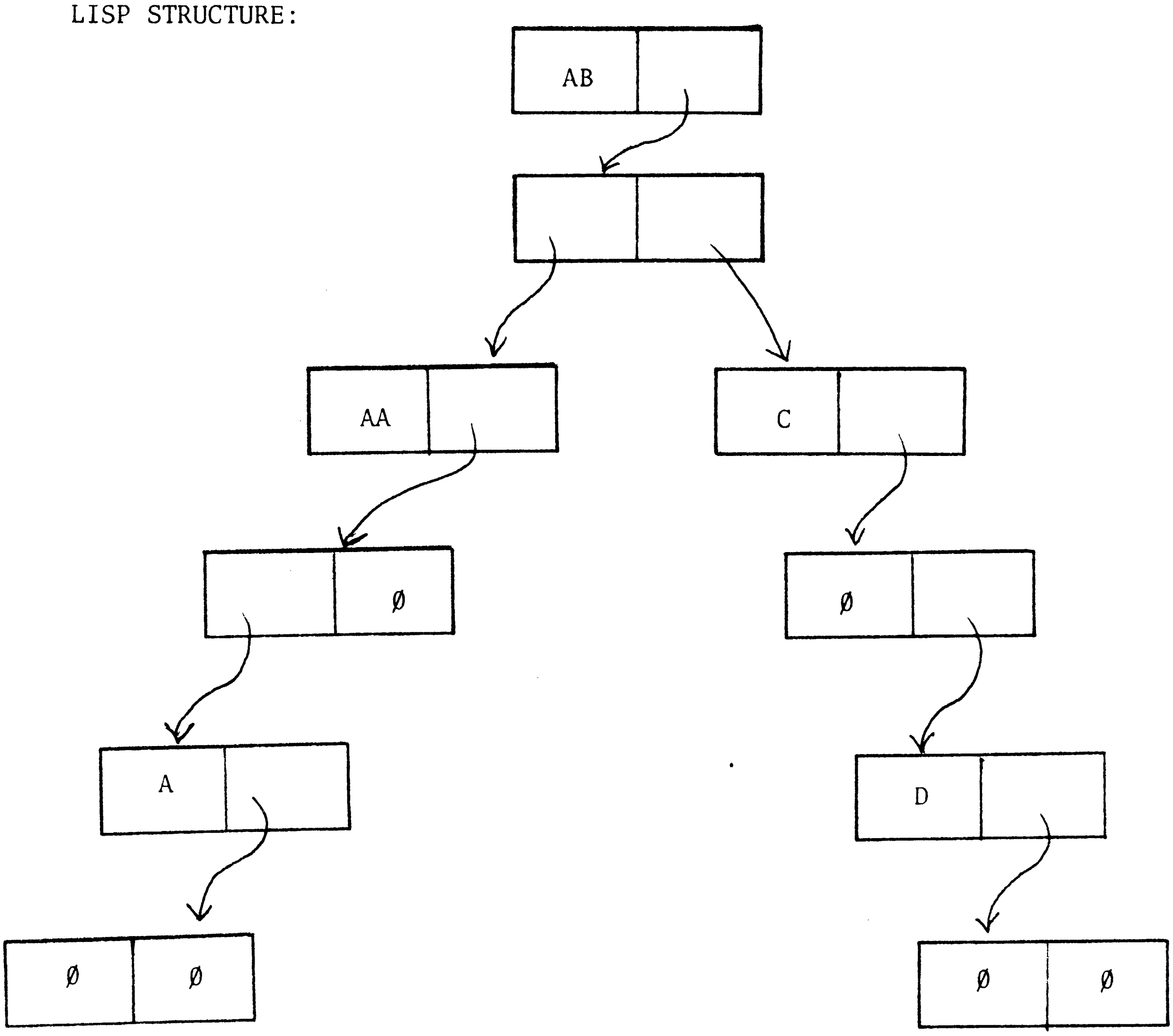
A node on the tree is represented by a LISP cell whose CAR is the element on the node and whose CDR is a dotted pair of the 2 daughter nodes.

When a node is created, the program automatically creates the dotted pair, even though both daughter nodes are NIL at the time. This makes insertion and printing algorithms very simple.

CONCEPTUAL TREE



LISP STRUCTURE:



INPUT FILE TO LISP

V.27

```
ET 'ALPHA '(NIL A B C D E F G H I J K L M N
            O P Q R S T U V W X Y Z))
EFUN ORDER NIL
    (PROG (X TREE)
 B  (SET 'TREE (LIST (READ) NIL))
    (COND ( (NULL (CAR TREE))
            (PRINT 'ALL-DATA-PROCESSED) (STOP)))
 A  (COND ( (NULL (SET 'X (READ) )) (PRTREE TREE))
          ( T (PUTINTREE X TREE) (GO A)))
    (GO B)))
EFUN PUTINTREE (X TREE)
    (COND ( (LOWER X (CAR TREE))
            (COND ( (NULL (CADR TREE))
                    (RPLACA (CDR TREE) (LIST X NIL)))
                  (   (PUTINTREE X (CADR TREE)))))
          ((NULL (CDDR TREE))
           (RPLACD (CDR TREE) (LIST X NIL)))
          ((PUTINTREE X (CDDR TREE)))))
EFUN PRTREE (TREE)
    (PRINT 'IN-FORWARD-ORDER)
    (PRF TREE)
    (PRINT 'IN-REVERSE-ORDER)
    (PRR TREE)
    (PRINT 'IN-ANOTHER-ORDER)
    (PRO TREE))
EFUN PRF (TREE)
    (COND ((NULL TREE) NIL)
          (T (PRF (CADR TREE)) (PRINT (CAR TREE)) (PRF (CDDR TREE)))))
EFUN PRR (TREE)
   (COND ((NULL TREE) NIL)
      (T (PRR (CDDR TREE)) (PRINT (CAR TREE)) (PRR (CADR TREE)))))
DEFUN PRO (TREE)
   (COND ((NULL TREE ) NIL)
        (T (PRINT (CAR TREE)) (PRO (CADR TREE) )(PRO (CDDR TREE)))))
EFUN LOWER (LIST1 LIST2)
   (PROG (Z)
      (SET 'Z ALPHA)
       (COND  ((NULL LIST1) (RETURN T))
           ((EQ (CAR LIST1 ) (CAR LIST2)) (RETURN (LOWER (CDR LIST1)
                 (CDR LIST2)))))
A   (COND ((EQ (CAR LIST1) (CAR Z)) (RETURN T))
          ((EQ (CAR LIST2) (CAR Z)) (RETURN NIL)))
    (SET 'Z (CDR Z))
    (GO A)))))
RDER)
A B C)
)
A)
 C)
 D)
 C C)
 X Y Z)
 C C C)
 R S T)
 S D)
 R S)
 B I)
 I A)
B A)
 T P)
 B J)
 C S)
L
'
```

OUTPUT PRODUCED

```
*EXECUTION BEGINS
 (NIL A B C D E F G H I J K L M N O P Q R S T U V W X Y Z)
  ORDER
  PUTINTREE
  PRTREE
  PRF
  PRR
  PRO
  LOWER
  IN-FORWARD-ORDER
 (A)
 (A B C)
 (C B A)
 (C C C C)
 (C C C)
 (C C S)
 (C C)
 (C I A)
 (C)
 (D D)
 (F B I)
 (I R S)
 (L B J)
 (L S D)
 (Q R S T)
 (S T P)
 (W X Y Z)
  IN-REVERSE-ORDER
 (W X Y Z)
 (S T P)
 (Q R S T)
 (L S D)
 (L B J)
 (I R S)
 (F B I)
 (D D)
 (C)
 (C I A)
 (C C)
 (C C S)
 (C C C)
 (C C C C)
 (C B A)
 (A B C)
 (A)
  IN-ANOTHER-ORDER
 (A B C)
 (A)
 (C)
 (C C)
 (C C C)
 (C C C C)
 (C B A)
 (C C S)
 (C I A)
 (D D)
 (W X Y Z)
 (Q R S T)
 (L S D)
 (I R S)
 (F B I)
 (L B J)
 (S T P)
  ALL-DATA-PROCESSED
*EXECUTION TERMINATED
```

POP-2 (Brindle)

String & List Processing Seminar
Assigned problem, written in XPOP (POP-2)

The program consists of 6 functions and one global function call, followed by the data. The functions:

INSERT - a general tree-inserting function which, given an item and a function with which to tell if items are in order (f(A,B) true iff A<B in some sense), inserts the item in tree TREE.

BUILD - INSERT partially applied to a function comparator of character strips - lines 17 thru 26. BUILD is then a function of one variable, a cstrip, to insert cstrip in tree TREE. LAMBDA notation is for nameless function for two variables, S1 and S2

PRINT_TREE - a specific function to linearize a tree of cstrip.

EXISTS - produces a truth value. I represent null pointers by integer 0 in my tree structure. EXISTS(PTR) tells whether a pointer exists. Could also have been written
  FUNCTION EXISTS; .DATAWORD="NODE" END for example
missing parameter for DATAWORD and = (in line 4) comes from stack - truth value left on stack by EXISTS.

PR_CSTRIP - standard system routine PR prints cstrips with enclosing primes, so I wrote this for neater O/P.

EXECUTE - performs bookkeeping for this application.

Explanation of some XPOP features:

SUBSCRC(I,STR) produces integer value of character I of cstrip STR

SP(n) spaces n in O/P, NL(n) produces n newlines,

CHAROUT takes integer code and outputs character

ITEMREAD produces next token from I/P stream as either an integer or a cstrip

function application is effected either with FN( ) or .FN

RECORDFNS creates a record class called "NODE" with 3 full item fields, accessed by VAL, LEFT, & RIGHT in order. Also produces constructor and destructor

conditionals are terminated with CLOSE; EXIT is a system macro for RETURN CLOSE

#N (%ITEM%) denotes partial application, leaves a new function on stack

```
XPOP COMPILER --- VERSION I

TODAY IS    JUNE 24, 1972. TIME = 21:28:41.95.


   VARS DESTNODE CONSNODE RIGHT LEFT VAL TREE BUILD;
   RECORDFNS("NODE",%0 0 0%)->RIGHT->LEFT->VAL->DESTNODE->CONSNODE;

FUNCTION EXISTS; NOT(=0) ; END; COMMENT DETERMINES IF TREE POINTER;

FUNCTION INSERT STRING LT;  VARS T;
      TREE->T; CONSNODE(STRING,0,0);   COMMENT LEAVE NEW NODE ON STACK;
      IF TREE=0 THEN ->TREE EXIT;
      LOOP:  IF LT(STRING,VAL(T)) THEN    COMMENT TEST ORDER USING GIVEN FUNC;
         IF T.LEFT.EXISTS  THEN LEFT(T)->T; GOTO LOOP
            ELSE ->LEFT(T) EXIT;
         ELSE IF  T.RIGHT.EXISTS THEN RIGHT(T)->T; GOTO LOOP
            ELSE ->RIGHT(T) EXIT;
      CLOSE;
      END;

   INSERT(% LAMBDA S1 S2; VARS N1 N2 CH1 CH2 I;
      DATALENGTH(S1)->N1;  DATALENGTH(S2)->N2;
      1->I;
      L: SUBSCRC(I,S1)->CH1;
      SUBSCRC(I,S2)->CH2;
      IF NOT(CH1=CH2) THEN CH1<CH2 EXIT;  COMMENT LEAVES TRUTH VAL ON STACK;
      I+1->I;  IF I>N2 THEN FALSE EXIT;   COMMENT FIRST STR >=SECOND IN LENGTH;
         IF I>N1 THEN TRUE EXIT; COMMENT FIRST SHORTER;
      GOTO L;
   END%) -> BUILD;   COMMENT BUILD NOW FUNC OF ONE VAR TO INSERT USING
                              TEST OF TWO CSTRS;

FUNCTION PRINT_TREE TREE;
      IF TREE.LEFT.EXISTS THEN PRINT_TREE(LEFT(TREE)) CLOSE;
      PR_CSTRIP(VAL(TREE));  SP(2);
      IF TREE.RIGHT.EXISTS THEN PRINT_TREE(RIGHT(TREE)) CLOSE;
END;
COMMENTS PR_CSTRIP

FUNCTION PR_CSTRIP CSTR:  VARS I N;
      1->I:  DATALENGTH(CSTR)->N;
      L: CHAROUT(SUBSCRC(I,CSTR));
      I+1->I;
      IF I=< N THEN GOTO L:   CLOSE;
   END;

FUNCTION EXECUTE:  VARS TERMIN STR; COMMENT PERFORMS PROBLEM ASSIGNED;
      SUBSCRC(1,.ITEMREAD)-> TERMIN; COMMENT GET TERMINATING CHAR;  0->TREE;
      L: .ITEMREAD->STR;
      IF SUBSCRC(1,STR)=TERMIN THEN
         2.NL; PRINT_TREE(TREE); 2.NL;
         .ITEMREAD->STR;
         IF SUBSCRC(1,STR)=TERMIN THEN .KILL;
   52 |       GOTO L;
   53 |    END;
   54 |.EXECUTE;
   55 |$  A ABC C CC CCC CCCC PAUL
   56 |        $


A  ABC  C  CC  CCC  CCCC  PAUL

                PAUL CCCC CCC CC C ABC A    (REST OF LINE 56)
   57 |$


A  ABC  C  CC  CCC  CCCC  PAUL

         CC PAUL CCC ABC C CCCC A $        (REST OF LINE 57)


A  ABC  C  CC  CCC  CCCC  PAUL

 END OF COMPILATION    JUNE 24, 1972. TIME = 21:28:45.87.

 57 CARDS WERE CHECKED.
 NO ERRORS WERE DETECTED.


STACKING DECISIONS= 1643
SCAN              = 459
SYNTHESIZE        = 1211


MAXIMUM STACK USAGE
   EXEC STACK:5 OF 50
   FUNCTION STACK:59 OF 150
   FN NESTING:0 OF 10
   COND NESTING:4 OF 10
STORAGE REMAINING
   DICTIONARY 164
   POLISH STREAM 651
   STRUCTURE STORE 843
FREE STRING AREA  = 20398


TOTAL TIME IN COMPILER     0:0:3.57.
SET UP TIME               0:0:0.06.
ACTUAL COMPILING TIME      0:0:3.49.
CLEAN-UP TIME AT END      0:0:0.02.
EXECUTION TERMINATED
```

SNOBOL (G. Lift)

BRIEF DESCRIPTION OF SNOBOL PROGRAM FOR STRING-LIST PROBLEM

The tree data structure is represented by a user-defined data type. Each node is an instance of the data type tree, having three fields. The first field, TWIG, is a pointer to a node containing a word "less than" this node; STEM is the word at this node; BRANCH is a pointer to a node containing a word "greater than" the word at this node. Null pointers are used to indicate no entry. The tree always contains at least one node, even if that node is completely empty.

There are two recursive functions to manipulate the tree. HANG(NODE,LEAF) looks at the subtree defined by NODE to find (or create) the proper node for the LEAF (word). If NODE has no STEM (i.e., there is no word associated with this node) LEAF is placed on NODE. Otherwise, LEAF is compared to STEM(NODE) to see whether it should be placed on the right (greater than) or left (less than) subtree. If no subtree exists in the direction chosen, a one node subtree is created. Then HANG calls itself with the top node of the subtree and LEAF.

FALL(NODE) prints the tree in sorted order. FALL calls itself to print the words in the left subtree of NODE, then prints the word at NODE, and then calls itself to print the words in the right subtree of node (i.e., a call with argument BRANCH(NODE)). Hence calling FALL with the root node of the tree will print the entire tree.

The "main program" is quite simple. First an empty node is created; this is the root node of the tree. HANG is always called with this node. The program then reads in words, one at a time, calling HANG to put each on the tree. When the "end of set" is read, FALL is called with the root node to print the tree in sorted order. The tree is then emptied (by assigning a new value to the root node), and a new dataset is read in.

notes:
1) the end of dataset character is "/".
2) words may be of any length, as long as they contain only alphabetic characters.
3) words are read in one per line (the word should be the first non-blank characters on the line. A word may be terminated by any non-alphabetic character, or the end of the line.
4) any end-of-file terminates the program.

```
SNOBOL4 (VERSION 3.6, APRIL 1, 1971)

(MTS IMPLEMENTATION JULY 18, 1971)


*1                  &ANCHOR = 1 ;  &TRIM = 1 ;
*4                  ALPHA = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
*5                  DEFINE('HANG(NODE,LEAF)')
*6                  DEFINE('FALL(NODE)')
*7                  DATA('TREE(TWIG,STEM,BRANCH)')
*8                  LEAFPILE = (SPAN(' ') | NULL) (SPAN(ALPHA) | '/' |
*8        +              NULL) . WORD
*9                  OUTPUT = '    *** TREE SORT   ' ; OUTPUT =
          *
*11       SEEDLING  TRUNK = TREE()
*12       NEWLEAF   INPUT LEAFPILE                            :F(END)
*13                 IDENT(WORD)                               :S(NEWLEAF)
*14                 DIFFER(WORD,'/') HANG(TRUNK,WORD)         :S(NEWLEAF)
*15                 OUTPUT =
*16                 OUTPUT = FALL(TRUNK) DUPL('.',50)         :(SEEDLING)
          *
*17       HANG      STEM(NODE) = IDENT(STEM(NODE)) LEAF       :S(RETURN)
*18                 LGT(LEAF,STEM(NODE))                      :F(L)
*19       R         BRANCH(NODE) = IDENT(BRANCH(NODE)) TREE()
*20                 HANG(BRANCH(NODE),LEAF)                   :(RETURN)
*21       L         TWIG(NODE) = IDENT(TWIG(NODE)) TREE()
*22                 HANG(TWIG(NODE),LEAF)                     :(RETURN)
          *
*23       FALL      DIFFER(TWIG(NODE)) FALL(TWIG(NODE))
*24                 OUTPUT = STEM(NODE)
*25                 DIFFER(BRANCH(NODE)) FALL(BRANCH(NODE)) :(RETURN)
*26       END

  0 SYNTACTIC ERROR(S) IN SOURCE PROGRAM
```

```
          *** TREE SORT

a
 b
  bac
 bbc
 c
 cc
 ccc
 cccc
 cd
 d
 efg
 hjz
 zlfq
 /

  A
  B
  BAC
  BBC
  C
  CCC
  CCCC
  CD
  D
  EFQ
  HJZ
  ZLFQ
  ....................................................

zlfq
hjz
efq
dc
cd
cccc
ccc
cc
c
bbc
bac
b
a
/

  A
  B
  BAC
  BBC
  C
  CC
  CCC
  CCCC
  CD
  DC
  EFQ
  HJZ
  ZLFQ
  ....................................................

x
the
cc
eark
qarf
c
oil
tree
leaf
ant
a
cccc
aB
xx
ccc
east
ear
earl
quip
muon
west
and
real
ion
/
```

```
A
AN
AND
ANT
C
CC
CCC
CCCC
EAR
EARL
EAST
ION
LEAF
MOON
OIL
PARE
PARK
QUIP
REAL
THE
TREE
WEST
X
XX
 ..................................................

/


 ..................................................

c
/

 C
 ..................................................
¢


 NORMAL TERMINATION AT LEVEL  0
 LAST STATEMENT EXECUTED WAS   12
```

[redacted]

```
#**LAST SIGNON WAS: 10:14.02   06-27-72
#  USER "3AWL" SIGNED ON AT  10:50.05 ON 06-27-72
#r ic55:stds-d
#EXECUTION BEGINS
  **  SET-THEORETIC DATA STRUCTURE:  INTERACTIVE DEMONSTRATION  **
          [VER:UM02.14]
  EXPLANATION?

?data(example,4,11)
    FILE =  sample
  ENTER INPUT FORMAT: (4a4)
  ENTER OUTPUT FORMAT: (1x,4a4)
  @ DONE!  L=    4  C=      10 [CPU=    0.401 ELAPSE=     70.92]
?*set(example)
  @ DONE!  L=    4  C=      10 [CPU=    0.008 ELAPSE=      4.63]
?$1 sample
>       1       AT
>       2       BALL
>       3       BALL
>       4       C
>       5       CA
>       6       CC
>       7       CCC
>       8       COW
>       9       CCCC
>      10       DOG
>      11       LOVE
# END OF FILE
?list(example,-1)
  [1 < # < 22]: #= 13
   FORMAT: (1x,4a4)
  AT
  BALL
  C
  CA
  CCC
  CCCC
  COW
  DOG
  LOVE
?data(example,4,11)
    FILE =  sample1
  ENTER INPUT FORMAT: (4a4)
  ENTER OUTPUT FORMAT: (1x,4a4)
  @  "EXAM" FREED
 DONE!  L=    4  C=      10 [CPU=    0.225 ELAPSE=     38.89]
?*set(example)
  @ DONE!  L=    4  C=      10 [CPU=    0.008 ELAPSE=      1.84]
?$1 sample1
>       1       LOVE
>       2       DOG
>       3       CCCC
>       4       COW
>       5       CCC
>       6       CC
>       7       CA
>       8       C
>       9       BALL
>      10       BALL
```

STDS (Katz)

In STDS, once data becomes a set it is well-ordered and all duplication is removed. Hence, this exercise just becomes the following:

a) read in the data -

data(STDS file name, maximum length of record, number of records to be read)

file = MTS file name

b) make it a set -

*SET (STDS file name)

c) listing the original data set (using MTS command preceeded by a $).

d) listing STDS file

list(STDS file name, type of format)

```
?list(examole,-1)
  [1 < # < 22]: #= 13
   FORMAT: (1x,4a4)
  AT
  BALL
  C
  CA
  CC
  CCC
  CCCC
  COW
  DOG
  LOVE
?oata(example,4,11)
    FILE =  samole2
  ENTER INPUT FORMAT: (4a4)
  ENTER OUTPUT FORMAT: (1x,4a4)
  @  "EXAM" FREED
 DONE!  L=     4   C=       10 [CPU=     0.208 ELAPSE=     38.05]
?*set(example)
  @ DONE!  L=     4   C=       10 [CPU=     0.009 ELAPSE=      3.42]
?$l samole2
>        1      LOVE
>        2      BALL
>        3      C
>        4      COW
>        5      AT
>        6      CCCC
>        7      DOG
>        8      CA
>        9      CCC
>       10      BALL
>       11      CC
# END OF FILE
?list(example,-1)
  [1 < # < 22]: #= 13
   FORMAT: (1x,4a4)
  AT
  BALL
  C
  CA
  CC
  CCC
  CCCC
  COW
  DOG
  LOVE
?
```