ORIGINAL

CGTM 13
Jim Wells
April 1967

List Processing Facilities in High Level Languages

TABLE OF CONTENTS

List Processing Facilities in High Level Language

Abstract

The purpose of this study is to determine the feasibility of including list processing facilities in a programming language not especially designed for list processing. In particular, an attempt to implement the language CORAL as a set of subroutines in PL/1 is described. There is a general discussion of the problems encountered in attempting to embed such facilities in a prespecified environment. This approach is advocated for many situations over the alternative approach, the design and implementation of a new language. The appendices contain complete listings of the PL/1 programs required to implement a CORAL-like language - "PLURAL" (PL/1 Used on Rings And Lists).

Justification

A description of the language CORAL is included as Appendix 1. This description is an excerpt from W. R. Sutherland's MIT doctoral thesis in E.E. "The On-Line Graphical Specification of Computer Procedures". It was decided to attempt to implement a CORAL-like facility on the 360/50 at the Stanford Linear Accelerator Center because the language appears to be ideally suited to the manipulation of data structures representing graphs. Graphical data processing is a major project of the SLAC facility of the Stanford Computation Center. Furthermore the language is well-defined and of a manageable size (as a project) while it presents most of the problems that would arise in the implementation of any list processing facilities.

PL/1 was chosen as the high level language of implementation primarily to expose the author to this recently proposed and partially implemented general purpose programming language. It is postulated by some that PL/1 will bring the scientific and commercial spheres together, that is, that this will be the new scratch point from which all programming libraries are built. In any event it is new, it is deeply encased in conventions, and it does keep the programmer a long way away from the bits. This last is anathema to a list processor and will prove our undoing (or at least force great compromises). This however will be the central point of the discussion. Namely, what are the costs of moving what has usually been implemented at the bit accessing level to a level outside a high-level language.

An initial and vital decision already implied in the preceding paragraphs is that we will not - under any circumstances - use any language but PL/1 in our implementation. No recourse to Basic Assembler. It is hypothesized that a "quick and dirty" unsophisticated implementation, fault-ridden as it may be from the point of view of the cycle-counters and ward-watchers, may be preferable to a tight-beautiful bit-crunching implementation in Basic Assembler, or its equivalent on other machines. The beauty of "the beast" is that it is quickly written, as faultless as simple logic (at the IF ... THEN ... level) and easily understood by the using masses. It is not always clear where a system should be tuned for efficiency before its use.

Frequently in an attempt to make a code efficient assumptions and compromises are made that are incorrect and render the efficient code useless. A higher language code which is logically correct can be tuned (made efficient) where experiment shows it to be necessary. In the current era of almost limitless code and nanosecond cycle times the virtues of lucid logic and quick implementation may outweigh some core wasting or the use of extra machine cycles.

I do not advocate abandoning language design or tight programming. It is merely stated that here we have some facilities that might prove useful in the development of algorithms for graphic data processing. If in a short time we can make these facilities available, try them out, develop some algorithms then we have done something worthwhile. The useful techniques used can be refined for production systems if time and space so dictate. If the approach is not fruitful then we have not wasted great amounts of time on finding this out. Really, what we are seeking is meaningful ways of representing and manipulating data. Representation is the key. CORAL has proven valuable in other environments, we should like to try it.

Well, then, have we justified our attack? Let us persue the study, see what compromises are required, who suffers and who gains.

Limitations Imposed on Implementation

I should be more precise in describing the language of implementation - PL/1. Numerous aids to list-processing are contained in the specifications of the PL/1 language (IBM Manual No. C28-6571-3 "IBM System/360 Operating System PL/1 Language Specifications pp. 145 - 149) however at the time of this investigation most of these features were on the deferred list (Manual No. C28-6594-0 "IBM System/360 Operating System PL1(F) Programmers Guide pp. 174 - 177). As a consequence the implementation is more naive than it would be, were these features provided.

All of the compromises required to use a higher level language boil down to one - the choice of data structure. If processing is done in a

language which provides access to every bit, then the language places no restrictions on what shall be located where. On the other hand working in PL/1 we are faced with a choice between no more than two data arrangements: arrays and structures. Either way we encounter the difficulty that a certain location declared to be of a certain type must always be of that type. Sutherland's implementation of CORAL on the 7090 had pointers and data sharing the same array (namely all of core storage). Here if pointers and data are to share the same array then they must be of the same type. In that case it is impossible to use pointers of type 'POINTER' rather all the pointers must be integers - relative addresses of the fields they reference. This implementation permits the user to work only with fixed point binary data so that pointers and data are of the same data type.

Now we are allotting 16 bits to every pointer -- our array could be as large as $2^{15}-1$ words. This is impossible so we wind up wasting many of the bits in the words used as pointers. We cannot pack more than one pointer into a word because PL/1 will not permit us to access part of a word if the data is of type integer. Sutherland had backward pointers and ring start pointers in alternate ring elements. This space saving move seems futile in our implementation (a small saving beside the huge waste just discussed) so a backward pointer and a ring start pointer are included in every ring element. The structure of PLURAL blocks is shown in Figure 1.

Implementation

The following is a discussion of the implementation of the PLURAL system.

A. Overall Structure

The PLURAL system consists of two elements. One of these is a set of declarations which must be included in the users program. These include the array STORE or S which will contain all the lists constructed. They also include declarations of type for all of the procedures providing the manipulation functions.

The second element is the set of procedures. These are briefly explained below. The discussion is similar to Sutherland's discussion of the Coral operaters.

B. The Functions of the Various Procedures Provided

The arguments passed to all PLURAL procedures must be of type 'fixed binary'.

MMB (type, length, list length)

Locates a block of length "length" in free storage list. Makes a PLURAL block of it. Makes all tie registers in block into ring starts. Returns pointer to first tie register of the block.

MB (type, length)

Locates a block of length "length" in free storage list. Makes a PLURAL block of it. Returns pointer to first tie register of the block.

UP (n,p)

Returns P-N, a pointer to n registers up from P.

DN (n,p)

Returns P+N, a pointer to n registers down from P.

LEFT (n,p)

Moves around ring n registers to the left. Returns pointer to tie register n to left of p.

RIGHT (n,p)

Returns pointer to tie register n to right of p.

BOT (n,p)

Returns pointer to register n from bottom of block pointed at by P.

TOP (n,p)

Returns pointer to register n from top of block pointed at by P.

SRIGHT (n,p)

Returns pointer to ring element n to right of the start of the ring pointed at by P.

SLEFT (n,p)

Returns pointer to ring element n to left of start of ring pointed at by p.

PUTR (p,q)

Place ring element pointed at by p on right of ring element pointed at by q. Returns p.

PUTL (p,q)

Place ring element pointed at by p on left or ring element pointed at by Q. Returns p.

MOVER (p,q)

Move the ring pointed at by p to the right of the ring element pointed at by Q. Returns p.

MOVEL (p,q)

Move the ring pointed at by P to the left of the ring element pointed at by Q. Returns p.

TAKE (p)

Remove the ring element pointed at by p from the ring its in. Returns p.

DELETER (p)

Return all elements of the ring pointed at by p to free storage. Used in a CALL statement.

QSTART (p)

Returns 1 if p points at a ring start tie register. Otherwise returns 0.

QTYPE (p,type)

P must point at first element of a block. Returns 1 if the type of that block is "type" otherwise returns 0.

QEMPTY (p)

P points to a ring start. Returns 1 if ring has no more elements. Otherwise returns 0.

LOAD (n,p)

Returns value of register p+n.

GETLB (p)

P must point to 1st register of a block. Returns length of that block.

GETLL (p)

P must point to 1st register of a block. Returns list length of that block.

GETT (p)

p must point to 1st register of a block. Returns type of that block.

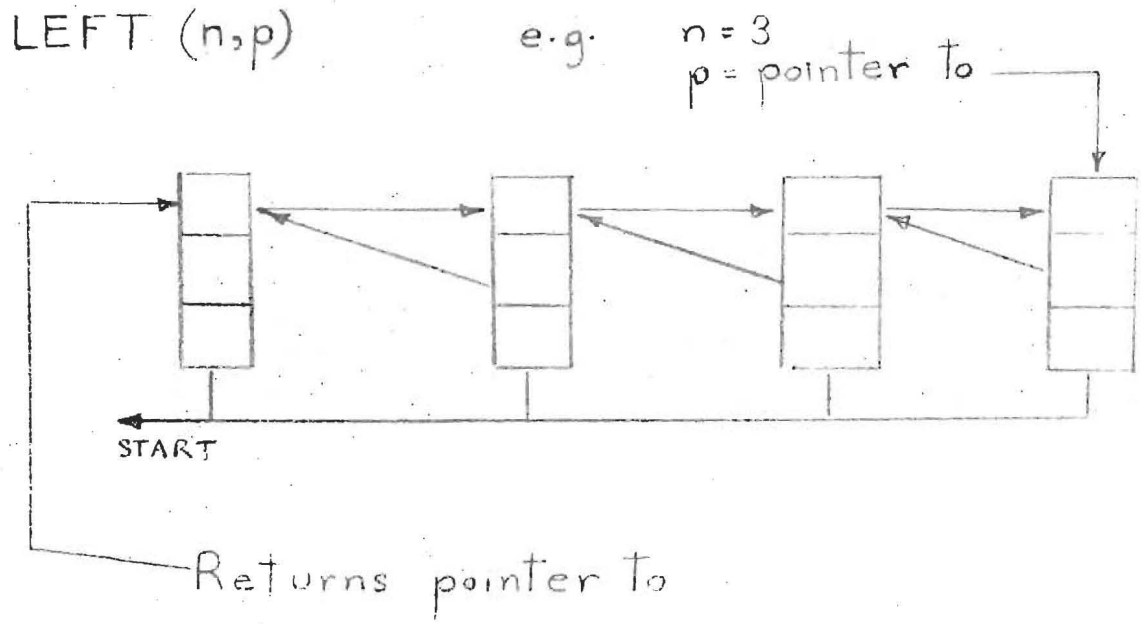
SETUPFS

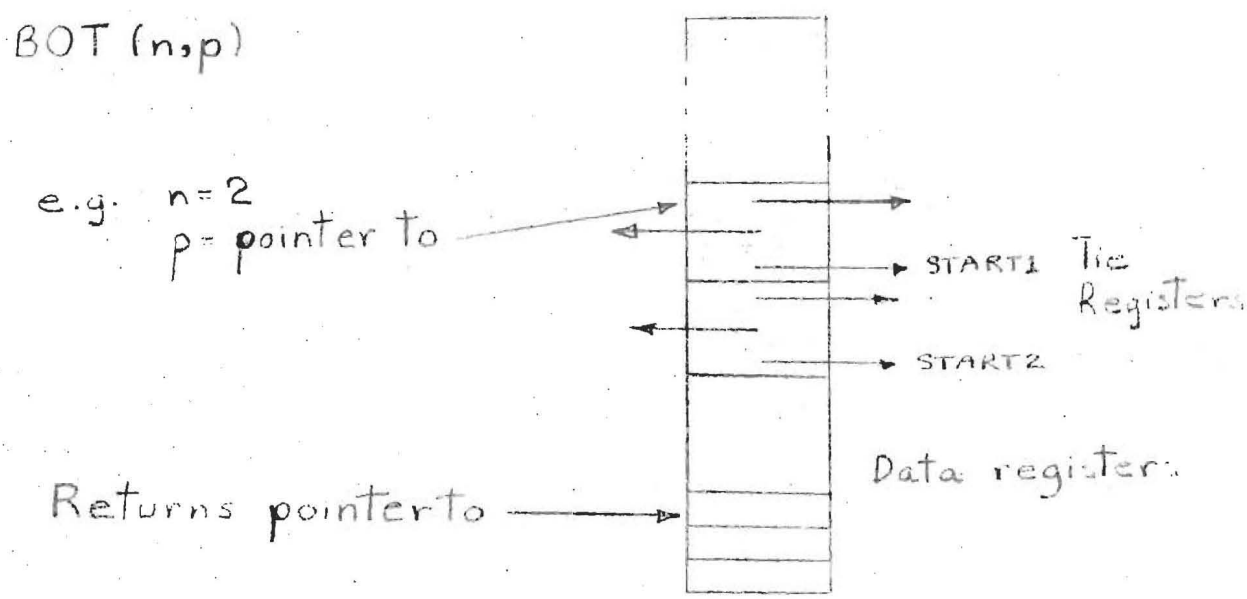
Used in a CALL statement. Initializes free storage list.

BH (p)

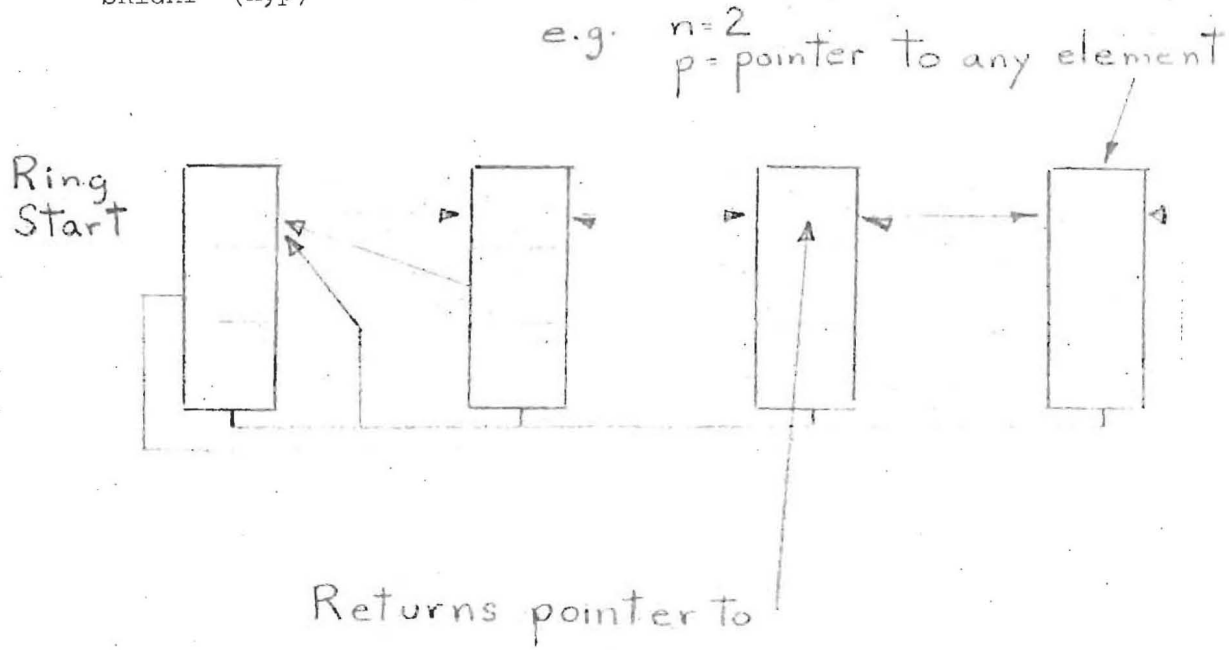
Return 1 if p points at type register of a block. Otherwise returns 0.

C. Pictorial Explanation of some Subroutines

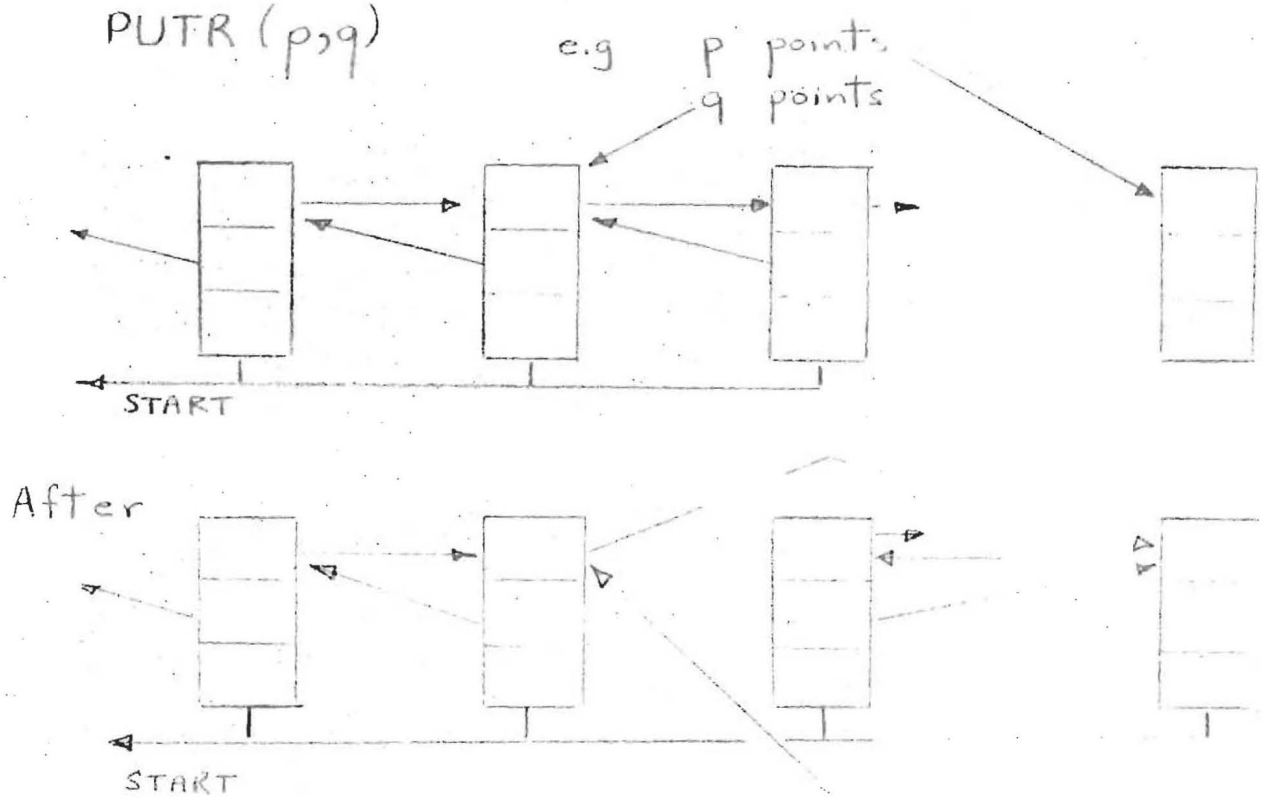




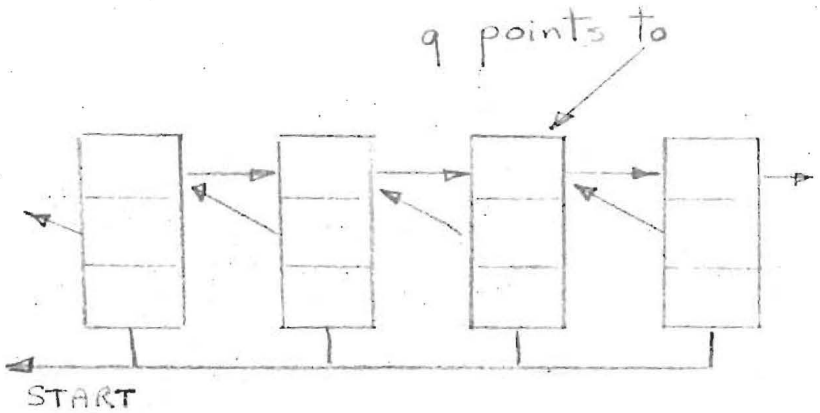
SRIGHT (n,p)

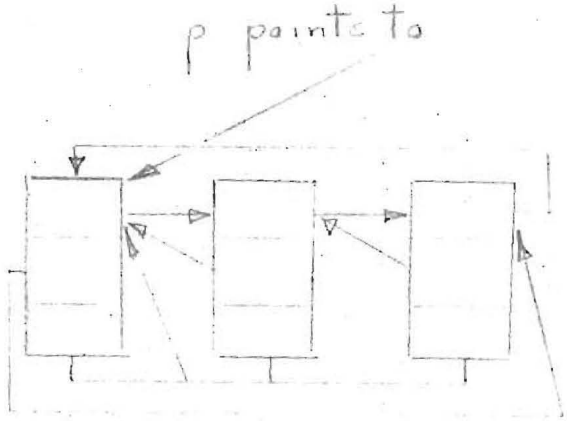


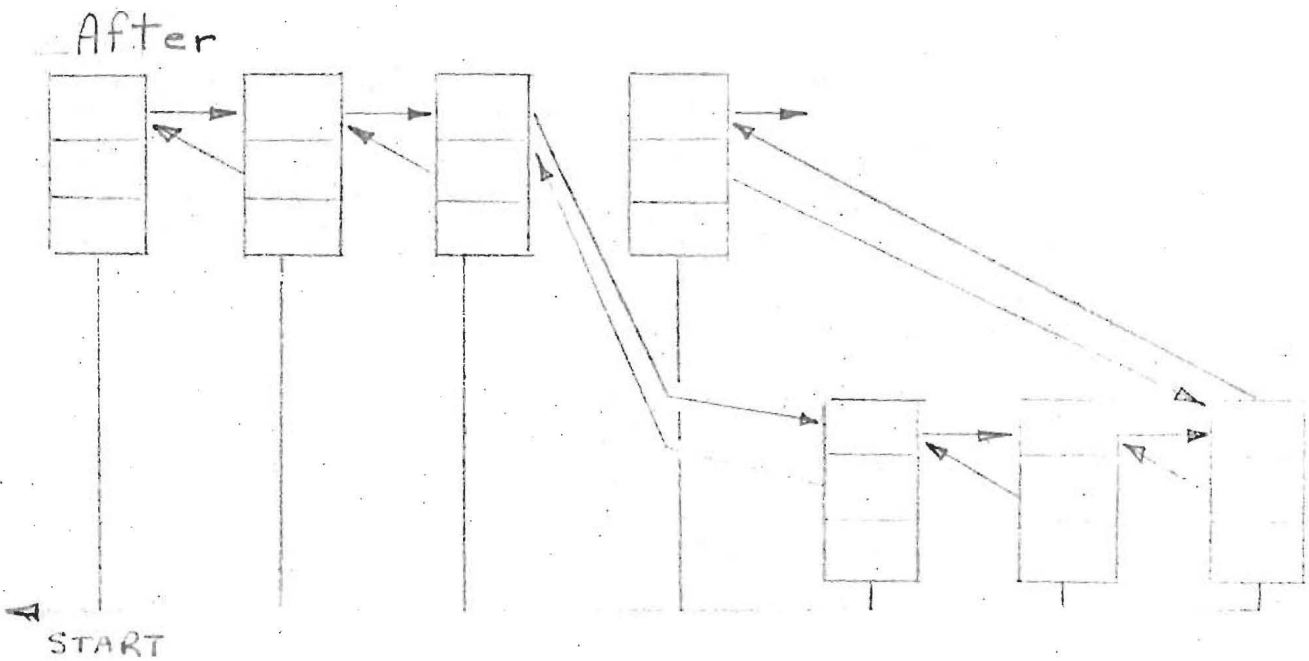
PUTR (p,q)



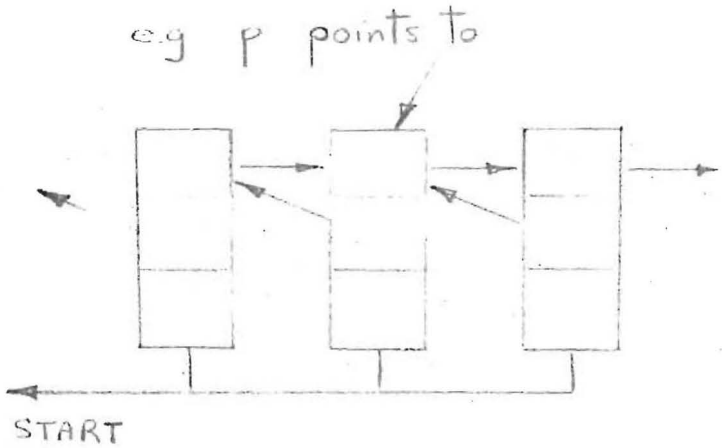
MOVER (p,q)



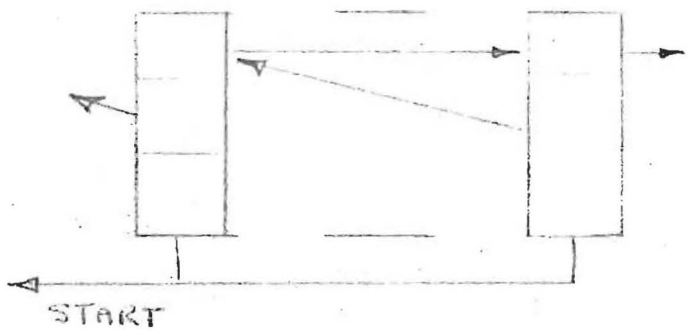




TAKE (p)



After



D. Management of Free Storage List

There is only a finite amount of storage that can be used for lists. Each time the procedure MB or MMB is used some of that storage leaves the "free storage list". Each time the procedure RETURNTØFS is used a block is released to free storage. Therefore it is the function of the routines MB (which MMB uses to get storage) and RETURNTØFS to manage free storage. The procedures SETUPFS chains together all of free storage. MB takes out contiguous blocks of length (length) wherever they are. If there is no block of the required length anywhere in free storage then the system dies. Each time RETURNTOFS is used a block is added to the free storage list.

It is apparent that, after the functions have been in use some time, free storage is dotted with blocks that are assigned to lists and blocks that are in free storage. It is possible that no blocks of the desired size might be found although there are many smaller spaces available. A desirable addition would be a repacking routine to compress the used storage spaces. This is absolutely non-trivial and requires further consideration.

E. Error Checking Included

The extent of the error checking included in the procedures bears some discussion. There is only a limited amount. The idea is that the error checking inherent in the PL/1 system (and in the O/S monitor) will be working for us. There are very few correctable errors so that the PLURAL systems action on error is a brief plural message followed by an error end to the program's execution. This is perhaps harsh, but to communicate run-time error information to the using program would require a very cumbersome structure -- essentially involving anticipation of all errors by the user.

The present system is very spare, once again the philosophy being that the primary objectives are (1) to have a system available which permits use of a possibly advantageous data structure and (2) to discover the areas of difficulty and compromise in constructing such a high level system. The system's operations are clear enough that additions and refinements can easily be made.

Desirable Language Features for List Processing

It should be apparent that PL/1 has not proven to be the ideal language in which to construct and manipulate a unique data structure. There are a few features, which, if included in a general purpose language, would make this job much easier. Some of these are in the specifications of PL/1 and not yet implemented, some exist in other languages (notably Extended Algol on the Burroughs B5500[2]).

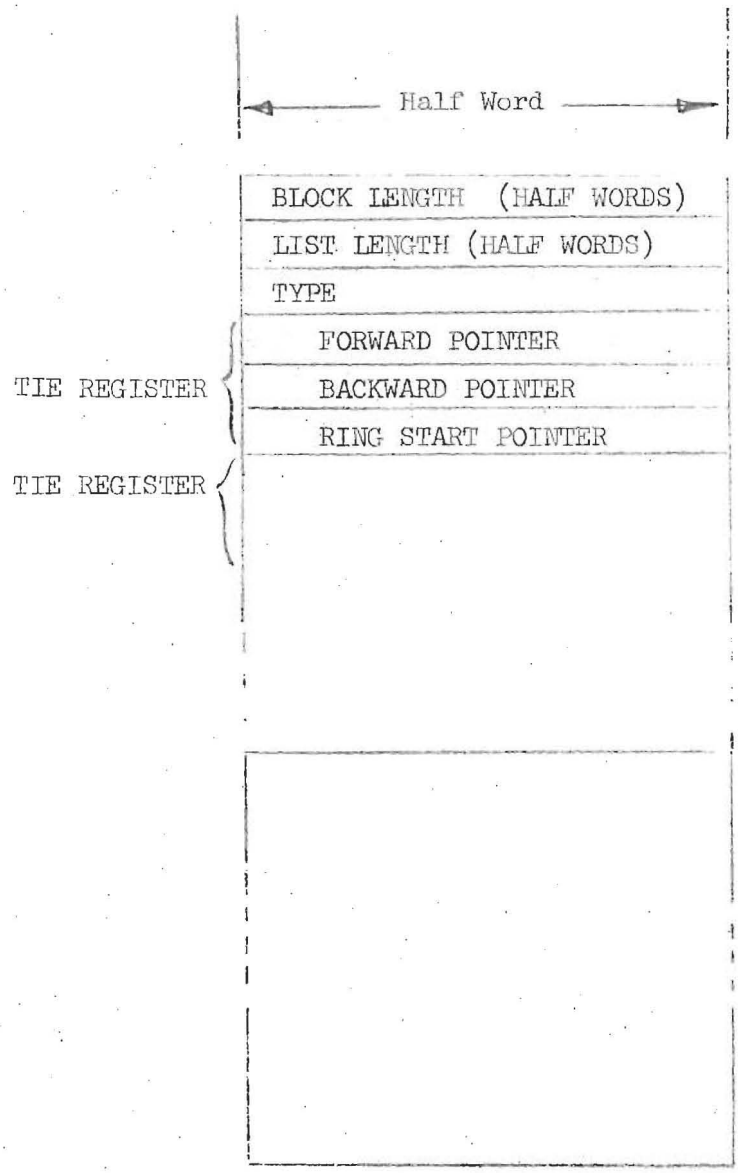
Firstly, bit accessing must be provided. Extended Algol's partial word operations would suffice. This is a requirement for compact data storage.

Secondly, the treatment of a piece of data should be dependent on the operators applied to it rather than on a global declaration. The rigidity of the declarations (and consequently of the treatment of a variable) in PL/1 makes the construction of a virtual machine (here the list processing machine) unwieldy. This is the reason we must say all data must be 16 bit fixed point rather than permitting data of any type.

Aside from the language features PL/1 lacks, I cannot resist a comment on the quality of the language's present implementation. It is not very impressive. Many compiler bugs were discovered. The implementation is not very clean as yet.

Figure 1

Structure of PLURAL Blocks



BIBLIOGRAPHY

1. "A Fortran-Compiled List Processing Language", H. Gelernter, J. R. Hansen and C. L. Gerberich, J. A. C. M. 7, 2 (April, 1960) pp. 87 - 101.
2. "Burroughs B5500 Extended Algol", Burroughs Corporation, 1964.
3. "The Coral Language and Data Structure", W. R. Sutherland, M.I.T. Lincoln Lab Tech Report - 405.
4. "IBM System/360 Operating System PL/1 Language Specifications", IBM Manual No. C28-6571-3.
5. "IBM System/360 Operating System PL1(F) Programmer's Guide", IBM Manual No. C28-6594-0.

Present Status

At present all the procedures are written but only a few have been debugged. Appendix 2 is a listing of the required global declarations and of the procedures. Debugging of the procedures is going on apace and a subsequent document will give their final form and a sample program that uses them.

A Similar Project

One reference deserves more than passing mention. Gelernter et al[1], desiring to simulate a geometry theorem proving machine on the 704 as had been done in IPL on the Johniac machine, were contemplating the implementation of IPL on the 704. (This was done concurrently by Newell et al as IPL V). J. McCarthy suggested that the facilities for the manipulation of IPL type data structures might be included in Fortran as functions. The nesting facility provided by Fortran would allow the construction of complex operations. This suggestion was followed up and the "FLPL" system was created. The authors successfully simulated their theorem proving machine and compare their system favorably with the IPL systems. They claim a great speed advantage over the interpretive IPL systems and on other points (ease of use, economy of core) come out even.

In their discussion they point out that the central good is the data organization - the use of lists. The means for manipulating the lists is definitely secondary. This is exactly my point in the justification of this study. Ring data structure appears to be a powerful tool for graphic data processing. If we can manipulate ring data structures easily within PL/1 we have a good system. Gelernter et al seem to feel it a great advantage to be operating within as highly developed and helpful a general purpose system as Fortran II. We will surely find it an advantage to be aided by the System/360 Operating System PL/1.

THE CORAL LANGUAGE AND DATA STRUCTURE*

William R. Sutherland

Massachusetts Institute of Technology
Lincoln Laboratory**
Lexington, Massachusetts

* Excerpt (Appendix C) from Electrical Engineering Doctoral Thesis "The On-Line Graphical Specification of Computer Procedures", submitted to M.I.T. on 10 January 1966. To be printed as Lincoln Laboratory Technical Report – 405.

** Operated with support from the U.S. Air Force.

THE CORAL LANGUAGE AND DATA STRUCTURE

The term "CORAL" stands for Class Oriented Ring Associative Language. The acronym is inexact as CORAL is really a service system consisting of a basic data structure form, a number of action subroutines, and a macro language for programming actions upon the data structure. CORAL was developed at the M. I. T. Lincoln Laboratory for the TX-2 Computer during 1964. The basic ideas of the data structure and language are not new and are applicable to any computer. However, the particular details of the structure and language are intimately related to the pecularities of TX-2, its macro assembler "Mark IV", and the character set of the Lincoln Writer keyboards.

CORAL DATA STRUCTURE

The CORAL data structure consists of table-like blocks of list elements. CORAL list ties are always formed into rings. Each element in a ring requires one 36-bit word and contains a 17-bit pointer to the next element. One element of the ring is the start or head and all the other elements are subordinate to it. The ring start element contains, besides its forward pointer, 9 bits of data and a 9-bit identification code (-0) which marks it as a start point. Every other element of a ring has a second 17-bit pointer which either points to the ring start element or is used as a backward pointer. One bit marks which type of pointer is being used and rings are built with back pointers and

start pointers alternating. Back pointers point to the closest previous element with a back pointer so that they form a complete ring. Alternation of the less useful pointer types retains the advantages of both pointers in half the space (one word per element) and with only a small loss of time. (Figure 1 illustrates the basic ring structure.)

BLOCKS

A block of elements collects many ties together and thus allows the multi-dimensional associations required for graphical data structures. Although it is sufficient to use at most two ring elements per block, more are often used for increased efficiency. A block is formed from a sequence of registers of any length and contains a blockhead identifier at the top, a group of ring elements and any number of data registers. (Figure 2 illustrates an example of a block.) Blocks are used to represent items or entities and the rings form associations between blocks. Thus, an element may be reached by moving up or down in a block rather than going around its ring. To find out what group the element belongs to it is necessary to find the head element of its ring and this is made efficient through the use of the start ties. If it is necessary to delete an element or insert a new element before it, the back ties are used to find the previous element. Thus, the full set of ties are necessary in the list elements if deletion, insertion, and identifying the elements' group are to be accomplished efficiently.

CORAL PROGRAMMING LANGUAGE

The CORAL language provides a means of programming actions which will operate on a CORAL data structure. A CORAL program consists of a combination of operators and named variables of type "POINTER". A pointer has as its

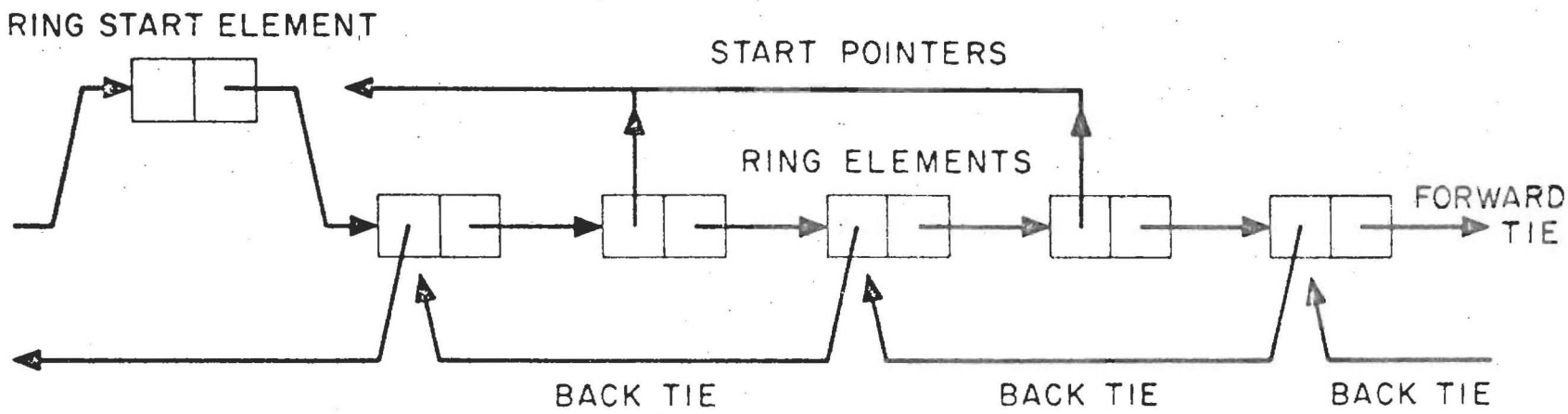


FIG 1 BASIC RING ELEMENT STRUCTURE

C23-356

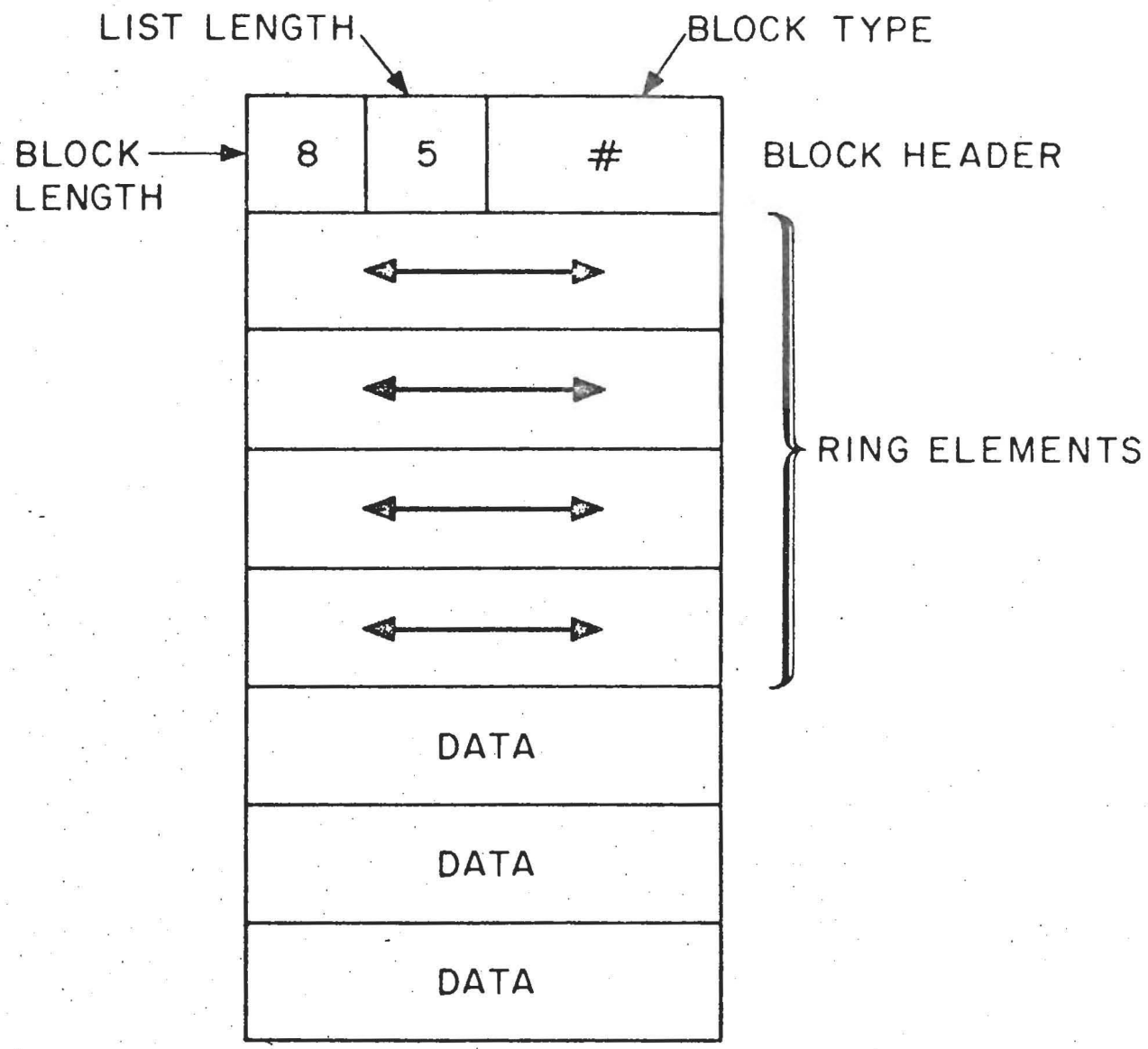


FIG 2 EXAMPLE OF BLOCK FORM

C23-357

value an integer address of a data structure register. Some operators will change the value of the pointer variables. For example, an operation for moving down two registers in a block would simply add two to the value of a pointer, thus making it point to a new location in the block.

Other operations will not change the value of the pointers, but will change the data structure pieces which are pointed to. Consider the operation of inserting a new tie element into a ring. This operation will have two arguments; the pointer to the element to be inserted, and a pointer to a reference element in an already existing ring. At the completion of the operation the pointers will be unchanged but the data structure will be changed since the two elements are now ring neighbors. It is useful to think of the pointers as named fingers keeping track of elements in the data structure. CORAL contains operators which move the fingers to new places in the structure and also operators which make the fingers manipulate whatever part of the structure they are grasping.

The various CORAL operators are used with pointer parameters and also with numerical parameters. The operator symbols are compound characters made from Lincoln Writer symbols. The list at the end of this appendix contains the principal operators and a description of how they work. The computer accumulator is used as an inter-operator communication register. Just as the result of an X + Y operation is normally available in the accumulator, so the result of a pointer moving operation is also available. CORAL operators may thus be nested in an almost arithmetic-like fashion. An operator parameter may be a named pointer variable, or it may be another operator expression. In this latter case the value resulting from the first operation on a pointer

is left in the accumulator and used directly as the input value for the second operation. Nesting to any depth may be accomplished.

CORAL OPERATORS

The list describing the principal CORAL operators uses a number of standard parameter symbols as follows:

Parameter Symbol	Meaning
A	Computer Accumulator
P	Pointer
Q	Reference Pointer
N	Numerical Value
S	Pointer
R	Control Transfer Label
TB	Numerical Value of Block Type

In a CORAL program actual named pointers or expressions will appear in place of the parameter symbols above. The appearance of "S" in macros containing "→ S" will cause A to be stored in pointer S at the end of the macro. "☛ R" at the end of a macro will cause a transfer of control to statement R. Omission of these parameters causes their actions to be omitted.

CORAL OPERATOR LISTING

MOVEMENT

These macros enable one to move from one part of the list structure to another. P points to a starting place which must be a tie register for left-right movement; a tie register or block head for up-down movement, except where indicated. N indicates how far to move. After any of the macros has been executed the accumulator points to the resulting location in the data structure.

Symbol	Format	Description
↑	P↑N→S⌐R	Go up N memory locations from P. P may point to any register of block. If N is negative, goes down.
↓	P↓N→S⌐R	Go down N memory locations from P. P may point to any register of block. If N is negative, goes up.
⊤↑	P⊤↑N→S⌐R	Go to bottom of block P, then go up N-1 memory locations. If N is omitted goes to top. If N is 0, ends up at register below bottom. If N is negative, goes down from bottom to outside of block.
⊤↓	P⊤↓N→S⌐R	Go to top of block P, then go down N memory locations. If N is negative, goes up above block, normally undesirable.

Symbol	Format	Description
⟩	P⟩N→S⇐R	Go right from P around ring N places. If N is negative, does nothing.
⟨	P⟨N→S⇐R	Go left from P around ring N places. If N is negative, does nothing.
⊤⟩	P⊤⟩N→S⇐R	Go to ring start of P, then right N places around ring. If N is negative, stays at ring start.
⊤⟨	P⊤⟨N→S⇐R	Go to ring start of P, then left N places around ring. If N is negative, stays at ring start.
[D]	P[D]N→S⇐R	Go down to N+1th data register of block P.

STRUCTURE CHANGING

These macros enable one to build, delete, and arrange list structure elements. After any of the macros has been executed, the accumulator points to some part of the list structure just referenced.

Symbol	Format	Description
⦶	X⦶L→S☛R	Make master master block at location determined by X for list structure L long. A points to header of block.
⊜	TP⊜L×LL→S☛R	Make master block for block-type TP with length L and list length (non-data length) LL. A points to master block.
Ⓑ	ⒷTP^L→S☛R	Make block of type TP and length L. If L is omitted, the length specified in the master block for type TP is used. A points to first tie register of block.
⊳	P⊳Q→S☛R	Put P right of Q. If P points to a ring start, error message results. For this and the operators listed below A has pointer P.
⊲	P⊲Q→S☛R	Put P left of Q. If P points to a ring start, error message results.
$\overline{⊳}$	P$\overline{⊳}$Q→S☛R	Move ring P to right of Q. P must point to a ring start which thereby becomes empty after its ring elements are moved.
$\overline{⊲}$	P$\overline{⊲}$Q→S☛R	Move ring P to left of Q. P must point to a ring start which thereby becomes empty after its ring elements are moved.
ⓛ	Pⓛ→S☛R	Take P out of ring it is in, making it an empty tie register (pointing to itself). If P points to a ring start, does nothing.

Symbol	Format	Description
$\otimes$	$P\otimes\rightarrow S\leftarrow R$	Delete ring P. If P points to empty tie register, does nothing. If P points to a ring element, takes it. Then checks to determine if the block containing the ring start (of this ring element) thereby has all of its tie registers empty; if so, return block to free storage, otherwise, do nothing more. If P points to a ring start, does $\otimes$ to each block containing a ring element of the ring start. A has pointer P.
$\overline{\otimes}$	$P\overline{\otimes}\rightarrow S\leftarrow R$	Delete Block P. This macro does $\otimes$ to each tie register of block P and then returns block P to free storage. A has pointer P.

TESTS

These macros are used for branching by "asking a question". If "answer" is yes, go to JYES; if "answer" is no, go to JNO. If JYES and JNO are expressions, do them as appropriate.

Symbol	Format	Description
H	P[H]⊃JYES|JNO☛R	P pointing to ring start ? NOTE: An empty tie register is considered a ring start. A has pointer P.
[illegible]	P[illegible]TP⊃JYES|JNO☛R	P pointing to block of type TP ? A has pointer P.
E	P[E]⊃JYES|JNO☛R	P pointing to empty tie register ? (i.e., tie register pointing to itself) A has pointer P.

DATA

These macros retrieve data into the accumulator.

Symbol	Format	Description
K	P[K]N→S☛R	Load the contents of the register whose address is N plus the pointer P into A.
I	P[I]→S☛R	Get length of block containing P. A has numerical answer.
J	P[J]→S☛R	Get list length (non-data length) of block containing P. A has numerical answer.
T	P[T]→S☛R	Get type number of block containing P. A has numerical answer.

GENERATORS

These macros cause an action to be performed for each ring element in a ring.

Symbol	Format	Description
[>]	P[>]ACTION^PDL→S☛R	Go right around ring P doing ACTION for each element except element P. Save pointer to each element at S before doing ACTION. Usually P points to ring start. "Current element" may be deleted by ACTION without hurting go-around. If an element is put right of current element by ACTION, it is not visited next. For recursive go-around include parameter PDL which specifies a push-list. Each time ACTION is called, A points to current element. At the end, A has the original pointer P.
	P[>]|ACTION^PDL→S☛R	Go right insertable around ring P. Same as above, except that "current element" must not be deleted by ACTION and, if an element is put right of current element by ACTION, it is visited next.
[<]	P[<]ACTION^PDL→S☛R	Go left around ring P. Same as [>], except goes left around ring.
	P[<]|ACTION^PDL→S☛R	Go left insertable around ring P.

CORAL DATA STRUCTURE AND OPERATORS IMPLEMENTED IN PL/1 */

```
/*   CORAL DATA STRUCTURE AND OPERATORS IMPLEMENTED IN PL/1     */
 MN:PROCEDURE OPTIONS(MAIN);
DECLARE S(500)FIXED BINARY,STORE(500) FIXED BINARY DEFINED S;
DECLARE(UP,DN,LEFT,RIGHT,BOT,TOP,MOVER,PUTR,PUTL,TAKE,
QSTART,QTYPE,QEMPTY,LOAD,GETLB,GETLL,GETT,BH,MB,MMB)ENTRY
RETURNS  (FIXED BINARY);
DECLARE(SRIGHT,SLEFT,NTH,MOVEL)ENTRY RETURNS(FIXED BINARY);
DECLARE (EFSL,BFSL,FSIZE)FIXED BINARY ;
       /* MAKE MASTER BLOCK  */
       MMB:PROCEDURE(TYPE,L,LL)FIXED BINARY;
            DECLARE(CTRMMB,MMBT,MMBA,LL,TYPE,L)FIXED BINARY;
                 MMBA=MB(TYPE,L);CTRMMB=0;
             MMBT=MMBA;
            MMBC:STORE(MMBA)=MMBA;
            S(MMBA+1)=MMBA;
            S(MMBA+2)=MMBA;
            CTRMMB=CTRMMB+1;
            IF CTRMMB=LL/3 THEN GO TO MMBB;
            ELSE DO;MMBA=MMBA+3;GO TO MMBC;END;
            MMBB:S(MMBT-2)=LL;
            RETURN(MMBT);
       END MMB;
 /* MAKE A BLOCK OF TYPE 'TYPE' AND LENGTH 'L' .
              RETURNS POINTER TO THE FIRST TIE REGISTER OF BLOCK
       MB:PROCEDURE(TYPE,L)FIXED BINARY;
              DECLARE(TP,TYPE,L)FIXED BINARY;
        DECLARE TL FIXED BINARY;
 /*   OBTAIN A BLOCK OF LENGTH L OF ADJACENT REGISTERS.    */
              TL=1;TP=BFSL;
              ZA: IF STORE(TP)=TP+1  THEN DO;
                            TL=TL+1;
                            IF TL=L THEN DO;
                                  IF BFSL=TP-L+2 THEN BFSL=BFSL+L ;
                                  ELSE STORE(TP-L+1)=TP+2;
                                  TP=TP-L+2;
                                  STORE(TP)=L;
                                  STORE(TP+1)=0;
                                  STORE(TP+2)=TYPE+999999999;
                                  RETURN(TP+3);
                                  END;
                            ELSE DO;
                                  TP=TP+1;
                                   GO TO ZA;
                                  END;
              END;
              ELSE DO;
                            TL=1;
```

```
                                        TP=STORE(TP);
                                        IF TP=EFSL THEN DO;
                                        /* WRITE ERROR MESSAGE 'OUT OF FREE STOR
                                              */
                                              /*    DIE    */
                                              END;
                                        GO TO ZA;
                        END;
                        END MB;
                  /*    END MB MAKE BLOCK     */
UP:                 PROCEDURE(N,P)FIXED BINARY;
                   DECLARE(P,N)FIXED BINARY;
                    RETURN(P-N);
                    END UP;
DN:                 PROCEDURE(N,P)FIXED BINARY;
                    DECLARE(N,P)FIXED BINARY;
                    RETURN(P+N);
                    END DN;
LEFT:                     PROCEDURE(N,P)FIXED BINARY;
                   DECLARE(N,P)FIXED BINARY;
                    DECLARE(TEMP1,TEMP2)FIXED BINARY;
                    TEMP2=STORE(P+1);
                    TEMP1=1;
  A:                      IF TEMP1>=N THEN DO;
                          IF N>0 THEN RETURN(TEMP2);
                                ELSE RETURN(P);
                          END;
                    ELSE DO;
                          TEMP2=STORE(TEMP2+1);
                          TEMP1=TEMP1+1;
                          GO TO A;
                          END;
                    END LEFT;
RIGHT:              PROCEDURE(N,P)FIXED BINARY;
                    DECLARE(N,P,TEMP1,TEMP2)FIXED BINARY;
                    TEMP2=STORE(P);
                    TEMP1=1;
BA:                 IF TEMP1>=N THEN DO;
                          IF N>0 THEN RETURN(TEMP2);
                          ELSE RETURN(P);
                          END;
                    ELSE DO;
                          TEMP2=STORE(TEMP2);
                 GO TO BA;
                          END;
                    END RIGHT;
BOT:                PROCEDURE(N,P)FIXED BINARY;
```

```
CORAL DATA STRUCTURE AND OPERATORS IMPLEMENTED IN PL/1     */

            DECLARE(N,P,PT)FIXED BINARY;
            PT=P;
CA:                 IF BH(STORE(PT))THEN
                    RETURN(PT+STORE(PT-1)-N-2);
                    ELSE DO;
                          PT=PT-1;
                          GO TO CA;
                          END;
                    END BOT;
TOP:        PROCEDURE(N,P)FIXED BINARY;
            DECLARE(N,P,PT)FIXED BINARY;
            PT=P;
DA:         IF BH(STORE(PT))=1 THEN RETURN       (PT-2+N);
            ELSE DO;
                    PT=PT-1;
                    GO TO DA;
                    END;
            END TOP;
SRIGHT:PROCEDURE(N,P)FIXED BINARY;
             DECLARE(N,P,TEMP1,TEMP2)FIXED BINARY;
             TEMP2=STORE(P+2);
             TEMP1=0;
    EA:      IF TEMP1>=N THEN RETURN(TEMP2);
             ELSE DO;
                   TEMP2=STORE(TEMP2);
                   TEMP1=TEMP1 +1;
                   GO TO EA;
                   END;
             END SRIGHT;
SLEFT:PROCEDURE(N,P)FIXED BINARY;
        DECLARE(N,P,TEMP1,TEMP2)FIXED BINARY;
             TEMP2=STORE(P+2);
             TEMP1=0;
    AN:      IF TEMP1>=N THEN RETURN(TEMP2);
             ELSE DO;
                   TEMP2=STORE(TEMP2+1);
                   TEMP1=TEMP1+1;
                   GO TO AN;
             END;
             END SLEFT;
 NTH:PROCEDURE(N,P)FIXED BINARY;
             DECLARE(N,P)FIXED BINARY;
             RETURN(P-1+3+3*STORE(P+1)+N+1);
             END NTH;
     /* PUT SINGLE ELEMENT P TO RIGHT OF Q    */
PUTR:PROCEDURE(P,Q)FIXED BINARY;
             DECLARE(P,Q)FIXED BINARY;
```

```
CORAL DATA STRUCTURE AND OPERATORS IMPLEMENTED IN PL/1     */

            DECLARE T1PT FIXED BINARY;
                  /* CHECK FOR RING START    */
                  IF STORE(P+2)=P THEN DO;
                        /* PUT OUT ERROR MESSAGE   */
                        RETURN(P);
                        END;
                  ELSE DO;
                        T1PT=STORE(Q);
                        STORE(Q)=P;
                        STORE(P+1)=Q;
                        STORE(P)=T1PT;
                        STORE(P+2)=STORE(Q+2);
                         STORE(T1PT+1)=P;
                        RETURN(P);
                  END;
            END PUTR;
      /* PUT SINGLE ELEMENT P LEFT OF Q    */
PUTL:PROCEDURE(P,Q)FIXED BINARY;
            DECLARE(P,Q      ,T1PT)FIXED BINARY;
            IF STORE(P+2)=P THEN DO;
                        /* PUT OUT ERROR MESSAGE   */


                  RETURN(P);
                  END;
            ELSE DO;
                  T1PT=STORE(Q+1);
                  STORE(T1PT)=P;
                  STORE(Q+1)=P;
                  STORE(P)=Q;
                  STORE(P+1)=T1PT;
                  STORE(P+2)=STORE(Q+2);
                  RETURN(P);
                  END;
            END PUTL;
      /* MOVE RING P TO RIGHT OF Q   "MOVER"    */
MOVER:PROCEDURE(P,Q)FIXED BINARY;
             DECLARE(P,Q,T1PT)FIXED BINARY;
                  /* CHECK FOR RING START   */
                  IF STORE(P+2)=P THEN DO;
                        RETURN(P);
                        END;
            T1PT=STORE(P+1);
            STORE(P+1)=Q;
            STORE(T1PT)=S(Q);
            S(P+2)=S(Q+2);
            S(Q)=P;
```

```
            T1PT=P;
      /* GO AROUND P CHANGING RING START POINTERS    */
      O:    T1PT=S(T1PT);
            IF T1PT¬=P THEN DO;
                  S(T1PT+2)=S(Q+2);
                  GO TO O;
                  END;
            ELSE RETURN(P);
            END MOVER;
      /* MOVE RING P TO LEFT OF Q  "MOVEL"    */
MOVEL:PROCEDURE(P,Q)FIXED BINARY;
            DECLARE(P,QT1PT)FIXED BINARY;
            IF STORE(P+2)=P THEN DO;


                  RETURN(P);
                  END;
            T1PT=S(S(Q+1));
            S(S(P+1))=T1PT;
            S(S(Q+1))=P;
            S(P+1) =S(Q+1);
            S(Q+1)=P;
            S(P+2)=S(Q+2);
            T1PT=P;
      R:    T1PT=S(T1PT);
            IF T1PT¬=P THEN DO;
                  S(T1PT+2)=S(Q+2);
                  GO TO R;
                  END;
            ELSE RETURN(P);
            END MOVEL;
        /*  END MOVEL;    */
      /* TAKE                                        */
TAKE:PROCEDURE(P)FIXED BINARY;
             DECLARE(P,T1PT)FIXED BINARY;
            IF STORE(P+2)=P THEN RETURN(P);
            T1PT=S(S(P+1));
            S(S(P+1))=S(P);
            S(S(P)+1)=S(P+1);
            S(P)=T1PT;
            S(P+1)=T1PT;
            S(P+2)=T1PT;
            RETURN(P);
            END TAKE;
    /*      END TAKE; */
DELETER:PROCEDURE(P);
            DECLARE(P,SETC,SETS       ,MT)FIXED BINARY;
```

```
CORAL DATA STRUCTURE AND OPERATORS IMPLEMENTED IN PL/1     */

                DECLARE (T1,T)FIXED BINARY;
            IF S(P+2)=P THEN GO TO B;
            T=S(P+2);
            IF S(P)¬=P THEN DO;
                  T1=TAKE(P);
                  T1=TOP(T,0);
                  SETC=1;
                  SETS=S(T1+1);
         AA:      IF S(T1+3)=T1+3 THEN MT=1;ELSE MT=0;
                  IF MT=1 THEN DO;
                                  SETC=SETC+1;
                                  IF SETC>=SETS THEN      /*RETURN BLOCK
                                  FSL     */
                                DO;CALL        RETURNTOFS(TOP(T1,0));RETU
                                          END;
                                 ELSE DO;
                                  T1=T1+3;
                                  GO TO AA;
                                  END;
                                  END;
                  ELSE;
                  END;   /* IF STORE(P)¬=P   */
            B:    T=STORE(P);
            C:   CALL DELETEB(T);
                  T=S(T);
                  IF T=P THEN DO;
                       CALL       DELETEB(T);
                                  RETURN;
                                  END;
                  ELSE GO TO C;
               END DELETER;
         /*  END DELETER;     */


     DELETEB:PROCEDURE(P);
               DECLARE(P,T,SETS,SETC)FIXED BINARY;
            T=TOP(P,0);T1=T;
            SETS=S(T+1);
            SETC=1;
             AAA:CALL DELETER(T+3);
            IF SETC>= SETS THEN DO;CALL RETURNTOFS(T1);
                                  RETURN;
                                  END  ;
                  ELSE DO;
                                  SETC=SETC+1;
                                  T=T+3;
                                  GO TO AAA;
```

```
                                    END;
                END DELETEB;
  /*    END DELETEB;          */
RETURNTOFS:PROCEDURE(P);
                DECLARE(P,T)FIXED BINARY;
             DECLARE LENGTH FIXED BINARY;
            S(EFSL)=P;
            T=EFSL=P;
            LENGTH=S(P);
        AAAA:S(EFSL)=EFSL+1;
            EFSL=EFSL+1;
            IF EFSL-T+1  >= LENGTH THEN RETURN;ELSE GO TO AAAA;
             END RETURNTOFS;
/* TESTS                  */
QSTART:PROCEDURE(P)FIXED BINARY;
                  DECLARE P FIXED BINARY;
            IF STORE(P+2)=P THEN RETURN(1);ELSE RETURN(0);
           END QSTART;
QTYPE:PROCEDURE(P,TYPE)FIXED BINARY;
                DECLARE P FIXED BINARY;
            DECLARE TYPE FIXED BINARY;
            IF STORE(P+2)-999999999= TYPE THEN RETURN(1);ELSE RETURN(0);
                END;
QEMPTY:PROCEDURE(P)FIXED BINARY;
                DECLARE P FIXED BINARY;
            IF S(P)=P THEN RETURN(1);ELSE RETURN(0);
                END;
/* DATA MOTION          */
LOAD:PROCEDURE(N,P)FIXED BINARY;
             DECLARE(N,P)FIXED BINARY;
            RETURN(S(N+P));
                END;
GETLB:
     PROCEDURE(P)FIXED BINARY;DECLARE P FIXED BINARY;RETURN(S(P));END
GETLL:
     PROCEDURE(P)FIXED BINARY;DECLARE P FIXED BINARY;RETURN(S(P+1));E
;
GETT:PROCEDURE(P)FIXED BINARY;DECLARE P FIXED BINARY;RETURN(S(P+2)
    -999999999);END;
 /* CHAUN THRU FREE STOREGE  */
SETUPFS:PROCEDURE;
                DO I=1 TO FSIZE-1;
                S(I)=I+1;
             END;
            BFSL=1;EFSL=FSIZE;
              RETURN;
              END SETUPFS;
```

CORAL DATA STRUCTURE AND OPERATORS IMPLEMENTED IN PL/1 */

```
   /* TELL WHETHER P POINTS AT A BLOCKHEAD --I.E. TYPE SPECIFIER  */
BH:PROCEDURE(P)FIXED BINARY;DECLARE P FIXED BINARY;IF S(P)>999999999
 THEN RETURN(1);ELSE RETURN(0);END;


/* TESTS OF ROUTINES BEGIN HERE        */
DECLARE(W1,W2,W3,W4,W5,W6)FIXED BINARY;
       FSIZE=501;
        PUT DATA('THIS IS THE SIZE OF STORE',FSIZE)PAGE;
        CALL SETUPFS;
        PUT DATA('RETURNED FROM SETUPFS',S)SKIP;
    W4=10;W5=12;W6=3;
         W1=MMB(W4,W5,W6);
        PUT DATA('RETURNED FROM MMB',W1)SKIP;
       DO I=1 TO 20;
      W2=MB(W4,W5);
              PUT DATA(W2);
            STORE(W2+3)=I;
         W6=PUTR(W2,W1);
            PUT DATA(I);
       END;
        PUT DATA('END LOOP')SKIP;
         PUT DATA(BFSL,EFSL)SKIP;
        PUT DATA(S)SKIP;
/* NOW TRY OUY LEFT AND RIGHT W1 IS START OF RING*/
        W6=LEFT(5B,W1);
        PUT DATA(W6);
        W6=LEFT(1B,W6);
        PUT DATA(W6);
        W6=RIGHT(25B,W1);
        PUT DATA(W6);
        W6=LEFT(5B,W6);
        PUT DATA(W6)SKIP;
END;
```