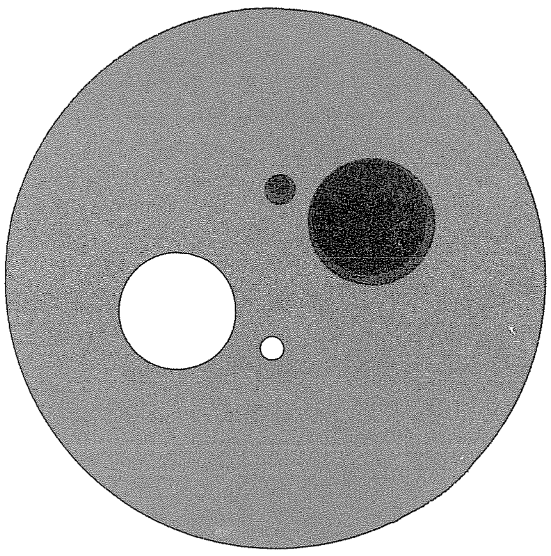


THE SAC-1 LIST PROCESSING SYSTEM*

by

George E. Collins

Technical Report #129†

July 1971

Computer Sciences Department
The University of Wisconsin
1210 West Dayton Street
Madison, Wisconsin 53706

*Research supported by National Science Foundation grant GJ-239, the Madison Academic Computing Center, and the Wisconsin Alumni Research Foundation.

†This report is also listed as Madison Academic Computing Center Technical Report #24 .

ABSTRACT

SAC-1 is a computer-independent system for Symbolic and Algebraic Calculation, programmed in Fortran IV (as defined by USASI). The system is especially directed towards efficient performance of large-scale calculations with multivariate polynomials and rational functions having exact coefficients. The SAC-1 List Processing System is the first of a series of SAC-1 subsystems (seven existing, three nearing completion, others planned). The present report is a new edition, with some corrections and minor changes, of the original report of July 1967.

Table of Contents

1. Introduction 1

2. Concepts and Terminology 2

3. Computer Representation of Lists 4

4. Erasure and the Available Space List 7

5. The List Processing Primitives 8

6. Description of the Fortran Subprograms 10

7. Recursion in SAC-1 14

8. The SAC-1 Character Set and Internal Code 21

9. The Primitive Subroutines READ and WRITE 22

10. Reading and Writing Lists 23

References 27

Appendix A: Listing of the Fortran Subprograms 28

1. Introduction

SAC-1 is a computer-independent system for Symbolic and Algebraic Calculation (the suffix, -1, provides for subsequent versions of the system). SAC-1 is especially designed for efficiently performing a wide variety of operations on polynomials and rational functions in several variables, with the arithmetic operations on coefficients being performed exactly. These polynomials and rational functions, and other mathematical objects, are represented in the computer as lists, or "list structures". Therefore, the SAC-1 List Processing System is provided as the first, and most basic, of a series of SAC-1 subsystems. A second SAC-1 subsystem, [7], performs operations on infinite-precision integers. Other subsystems, [8,9,10,11,12], perform progressively more specialized and sophisticated operations on polynomials and rational functions. Three additional subsystems (for polynomial greatest common divisor and resultant calculation, for polynomial factorization and for exact solution of systems of linear equations) are presently near completion, and still others are being planned.

SAC-1 is computer-independent in the sense that it is programmed, with the exception of a small number of simple primitive subprograms, entirely in the ASA (now USASI) Fortran IV language, [3]. In order to implement SAC-1 on a particular type of computer, it suffices to compile the SAC-1 Fortran subprograms and to program in assembly language according to specifications the primitive subprograms. A programer familiar with the assembly language should be able to program and test the primitive in no more than a week. The SAC-1 system has been implemented on CDC 1604, 3600, 6400, 6600 and 7600 computers, IBM 7094 and system 360 computers (models 50, 65 and 67), GE 635 and 645, UNIVAC 1108, and others.

The SAC-1 List Processing System together with the SAC-1 integer arithmetic and polynomial systems, [7] and [8], constitute a computer-independent counterpart of the earlier PM system, [1], for the IBM 7094, in that similar capabilities are provided using similar algorithms and data structures. In particular, the SAC-1 List Processing System is a reference-count system having origins in [1] and [4]. The SLIP system, [5], is another well-known reference-count list processing system, which however differs in that it employs a symmetric, or doubly-linked, representation of lists. For a thorough discussion of list processing, including the relative advantages and disadvantages of the use of reference counts versus "garbage collection" methods, see [2].

2. Concepts and Terminology

In [4] we introduced the notation of a list as an abstract mathematical object, to be distinguished from its various possible symbolic and computer memory representations. Since this distinction will be assumed in our description of the SAC-1 system, it is appropriate to repeat here some of the relevant definitions and to introduce certain terminology.

Intuitively, a "list" is just a finite sequence, some of whose elements may themselves be finite sequences, etc., to any desired depth. Those elements which are not themselves finite sequences will be referred to as "atoms", and the "depth" to which one may descend before encountering an atom will be called the "order" of a list. This leads to the following more precise definitions. Let A be an arbitrary set. In contexts where A is fixed, the members of A will be called atoms. Now set $A_o=B_o=A$. Assuming B_m has been defined, define A_{m+1} to be the set of all finite sequences, each of whose elements is a member of B_m, and then define B_{m+1} to be the union $\bigcup_{i=o}^{m+1} A_i$. Clearly $B_o \subset B_1 \subset B_2 \subset \ldots$. Set $L_{m+1} = B_{m+1} - B_m$ (for $m \geq o$). L_{m+1} is the set of all lists, over, A of order m+1, and $L = \bigcup_{i=1}^{\infty} L_i$ is the set of all lists over A.

It is customary to denote a list by enclosing its elements in parentheses, separated by commas. For example, if we take A to be the set of natural numbers, then "(3, 1, 1, 2)" denotes a list of order 1, while "((3), 1, (1,2))" denotes a list of order 2. Note that the order of occurrence of elements is significant and that an element may have several occurrences. Thus, (3, 1, 1, 2), (3, 1, 2), and (1, 2, 3) are three distinct lists. The length of a list is its length considered as a finite sequence. For example, (3, 1, 1, 2), (3, 1, 2) and ((3), 1, (1,2)) are lists of lengths 4, 3 and 3 respectively. A list may have no elements at all; there is a unique such list, called the null list, which is denoted by "( )". Its length is zero, and its order is one. By contrast, (( )) is a list of length one and order two; its unique element is the null list.

The elements of ((3), 1, (1,2)) are (3), 1, and (1,2); 3 and 2 are not elements, but are constituents. The constituency relation is the transitive closure of the elementhood relation: every element of z is a constituent of z, and if y is a constituent of z and x is a constituent of y, then x is a constituent of z.

Length and order are two measures (width and depth) of the complexity of a list. An area-like measure of a list is its extent. Let us say that the extent of an atom is zero. Then the extent of an arbitrary list $(x_1,\ldots,x_n)$ is defined, by induction on its order, to be $n + \Sigma_{i=1}^{n}$ extent (x_i). We will see that in SAC-1 the extent of a list is the number of cells required for its computer memory representation (except when overlapping occurs).

The inverse of any list $(x_1,\ldots,x_n)$ is the list $(x_n,\ldots,x_1)$. Any x_i which are themselves lists are not inverted. For example, the inverse of ((3), 1, (1,2)) is ((1,2), 1, (3)). The concatenation of two lists $(x_1,\ldots,x_m)$ and $(y_1,\ldots,y_n)$ is the list $(x_1,\ldots,x_m,y_1,\ldots,y_n)$.

3. Computer Representation of Lists

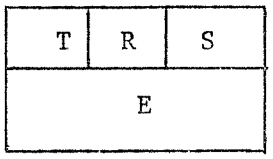
In SAC-1, any list is represented in computer memory by a collection of cells. A cell consists of a fixed number, say k, of memory locations. The number k is fixed for a particular implementation of SAC-1, but may be different for different implementations. For most computers k will be two, but an IBM 360 computer with a 32-bit word, for example, requires K = 3. By convention, the address of a cell is the address of its first memory location.

Each cell consists of four fields: type field (T), element field (E), reference count field (R), and successor field (S). The computer representation of a list $(x_1,\ldots,x_n)$ consists of a set of n cells, say $C_1,\ldots,C_n$, linked via their successor fields, together with representations, assumed already given, of any of the x_i which are themselves lists. I.e., the successor field of cell C_i contains the address of cell C_{i+1} $(1 \leq i < n)$; the successor field of C_n contains zero (it is assumed that zero is not the address of any cell, which is easily arranged). The type field of cell C_i contains the value zero or one according as x_i is an atom or a list. If x_i is an atom, then the element field of C_i contains x_i; if x_i is a list, then the element field of C_i contains the location of some representation of the list x_i. In this connection, the location of the list $(x_1,\ldots,x_n)$ is taken to be the address of the first cell, C_1, in its representation. Of course, a given list may have several representations, simultaneously.

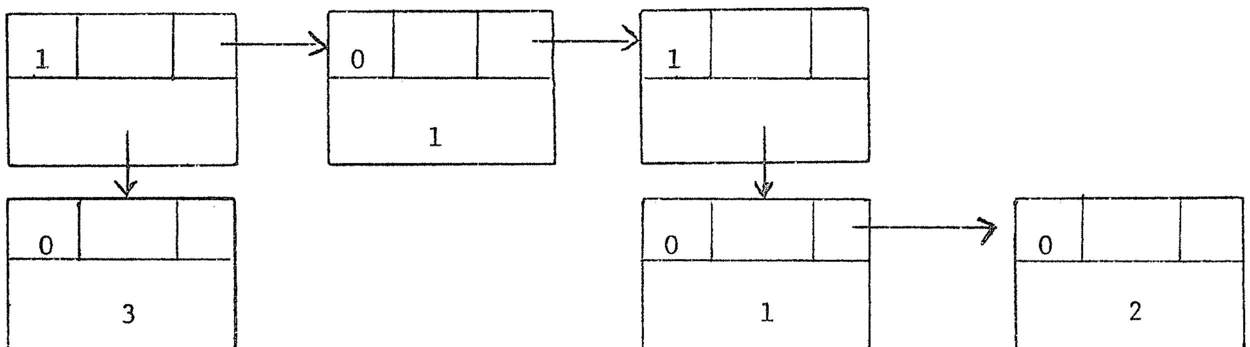
Note that, according to this definition, whenever a representation of a list $(x_1,\ldots,x_n)$ is present, so is a representation of any final segment of it, $(x_k,\ldots,x_n)$, automatically. This turns out to be a very useful property for many applications, polynomial algebra in particular.

The above definition does not apply to the representation of the null list. By special convention, no cells are used in its representation, and the location of its representation is zero.

We draw diagrams of list representations in which each cell of the representation is depicted by a rectangle. Such a rectangle contains four subrectangles corresponding to the four fields of the cell. The geometry of the diagram is derived from the typical case in which there are two words per cell and the element field consists of exactly the second word. Here is a diagram of a cell:



A diagram for our example list ((3), 1, (1,2)) would appear as follows:



The contents of successor fields are suppressed, since they are determined (to within an isomorphism) by the horizontal arrows; similarly for element fields containing the location of a list. The type fields could also be inferred from the diagram.

We have not yet specified the contents of reference count fields. The reference count field of any cell contains the number of arrows which terminate at it. However, this count must include a third type of arrow not shown in the above diagram, the program arrow. At any instant during the execution of a program, there exists a certain number of variables which are currently defined, and whose current values are locations of lists. Each such variable has associated with it a program arrow terminating at the first cell of the list which is its value.

It follows from this discussion, and from the following section on list erasure, that, in a correct program, all reference counts are greater than or equal to one.

With regard to field sizes, one bit obviously suffices for the type (but more could be used). The element field is assumed to be no longer than one word (but it could be shorter). The successor field must be long enough to contain any machine address (whether real or virtual). It is easy to see that the number of distinct computer memory locations (real or virtual) is an upper bound for any reference count. Hence if the reference count field is given the same length as the successor field, there will be no possibility of reference count overflow. In the present version of the system, no check is made for such overflow. However, an alternate version of the system is available in which a shorter field is allotted to the reference count, overflow is detected and, when it occurs, remedial action is taken, which consists in making a partial copy of the list in which the overflow would otherwise occur. This alternate version involves a change in only a single system subprogram, namely BORROW.

4. Erasure and the Available Space List

As in other list processing systems, any cell is, at any instant, included in the representation of at least one list. This is provided for by the presence of a special list, the available space list, which is complementary to all other lists: a cell is in the available space list if and only if it is not in any other list.

In SAC-1, as in most other systems, the available space list, hereafter called AVAIL, is maintained in the form of a list of order one (SLIP, [5], is the known exception). For the sake of easy identification, each cell on AVAIL carries a reference count of zero.

So that it will be accessible to all the subprograms requiring it, the location of the available space list is kept in Common, specifically, in the first location of labelled common block TR1, with the symbolic designation AVAIL.

An initial available space list is created by the subroutine BEGIN (A, N), in which A is a one-dimensional integer array of N elements. N must be a multiple of the number, k, of words per cell. An available space list of N/k cells is created from the memory allocated to A, and its location is stored in AVAIL.

The only system subprograms which remove cells from AVAIL are PFA and PFL. This is a desirable feature to maintain, since it makes it easy to acquire statistics on the numbers of cells used in various applications.

Whenever a particular list representation is no longer needed in the continued execution of a program, its "release" must be programmed through a call of the subroutine ERASE. ERASE returns to AVAIL all cells in the given list which are not shared with other lists, as determined by the reference counts, and decrements by one the reference counts of other cells as appropriate. This system has the advantage over the garbage collection of LISP, [6], that the processing time for reclaiming cells for re-use remains always in direct relation to the number of cells reclaimed (see [1] and [4].

The reference count system makes possible a further gain in efficiency through the use of specialized subroutines for the erasure of lists of pre-determined structure. One such specialized subroutine, ERLA, is included in the present system for erasing lists of atoms, i.e., lists of order 1. Other specialized subroutines for erasure will be available in other parts of SAC-1.

If a program fails to apply ERASE to a list where this is appropriate, the result is a permanent loss of a certain number of cells. If such an error occurs in a section of program which is executed repeatedly, the result may be a rapid, complete depletion of AVAIL. Hence, careful attention should be given to this matter.

Of course, a list need not be immediately erased after its usefulness has terminated. The importance of immediate erasure depends on how extensive the list may be. A similar remark applies in garbage collection systems, where it may be desirable to program abandonment of lists, since automatic abandonment does not occur until a list variable is redefined or execution of a sub-program is completed.

5. The List Processing Primitives

Ten primitive subprgrams are required for the SAC-1 list processing system. Of these, two are associated with each of the four cell fields, for a total of eight; the other two are used only by the BEGIN subroutine. These primitives must, for the most part, be written in an assembly language, in the correct form for linkage with the Fortran programs.

There are four primitive subroutines for storing information in cell fields. They are: STYPE(X,Y) - type field, SCOUNT(X,Y) - reference count field, SSUCC(X,Y) - successor field, and ALTER(X,Y) - element field. In each case the argument Y is the location of some cell and the argument X is a value which is to be stored in the appropriate field of the cell at Y. In each case, the arguments X and Y are Fortran integers, with the possible exception of ALTER. Here, the correct interpretation of X may be something other than an integer (e.g. Hollerith), but with respect to Fortran, it is disguised as an integer; and ALTER merely transfers X from one memory location to another.

There are four corresponding field retrieval function subprograms: Z=TYPE(Y), Z=COUNT(Y), Z=TAIL(Y) - successor field, and Z=FIRST(Y) - element field. As above, each argument Y is the location of some cell. The appropriate field of cell Y is extracted and this value is returned as the Fortran integer Z.

Two additional function subprograms are needed for BEGIN: Z=NWPC(X) and Z=LOC(X). In NWPC, the argument X is not referenced. The value Z returned is always the number of words per cell. NWPC could be programmed in Fortran, but it is classed as a primitive because it is computer-dependent. In LOC, the argument X is some variable (possibly subscripted). The value Z returned is the address of the memory location assigned to X. LOC is unlike the other primitives in that there is no assurance implicit in the A.S.A. Fortran specifications that the subprogram LOC can be written, even in machine language. For most computers and their Fortran compilers, it can be; in those cases where it cannot, it may be necessary to rewrite BEGIN as a primitive.

6. Description of the FORTRAN Subprograms

In the following descriptions, function subprograms are indicated by a prototype of the form $Z = F(X_1,\ldots,X_n)$, while subroutines are indicated by a prototype of the form $S(X_1,\ldots,X_n)$. In SAC-1, the arguments of a function subprogram are never defined or redefined as a result of the execution of the subprogram.

ADV(X,Y). Y is a non-null list $(y_1,\ldots,y_n)$. After execution of ADV, the value of X is y_1 and the new value of Y is the list $(y_2,\ldots,y_n)$. The given list $(y_1,\ldots,y_n)$ remains intact. CALL ADV (X,Y) is equivalent to X=FIRST(Y), Y = TAIL (Y).

BEGIN (A,N). A is a one-dimensional integer array of N elements. N must be a multiple of K, the number of words per cell in the SAC-1 implementation at hand. BEGIN creates an available space list from A consisting of N/K cells and initializes to null the system pushdown stack, STAK. Any SAC-1 main program will therefore normally begin with a call of BEGIN.

Z = BORROW (X). X is the location of an arbitrary list. The reference count of this list is increased by one (unless X is null), and Z = X, in the current implementation. However, in order to maintain compatibility with alternate implementations, it is recommended that users always reference BORROW in the form X = BORROW(X) rather than CALL BORROW(X). The current implementation assumes the increased reference count will never overflow the reference count field. In implementations where such overflow is possible, an alternate version of BORROW will return a value which may be the location of a (partially) different representation of the given list when necessary to prevent overflow.

Z = CINV(X). X is the location of an arbitrary list $(x_1,\ldots,x_n)$. The inverse list $(x_n,\ldots,x_1)$ is constructed, and Z is its location. The list X remains intact. Elements x_i of X which are lists have their reference counts increased by one via an application of BORROW, and hence are not copied.

Z = CONC(X,Y). X and Y are lists, say $(x_1,\ldots,x_m)$ and $(y_1,\ldots,y_n)$. X and Y are concatenated to form the list $(x_1,\ldots,x_m,y_1,\ldots,y_n)$ by storing the location of Y in the successor field of the last cell of X. Hence X no longer exists (unless Y is null). None of the reference counts in either X or Y are modified. Application of CONC is usually inappropriate except when all main level reference counts of X and Y are one and X and Y are not needed after their concatenation.

DECAP(X,Y). Y is the location of a non-null list $(y_1,\ldots,y_n)$. After execution of DECAP, the value of X is y_1 and the new value of Y is the list $(y_2,\ldots,y_n)$. The first cell of the given list is returned to the available space list, whatever its reference count was initially. No reference counts in $(y_2,\ldots,y_n)$ are changed. Application of DECAP is therefore ordinarily appropriate only when all main-level reference counts in the given list are known to be one.

ERASE(X). The arbitrary list X is "erased." I.e., those cells of X which are not parts of some other list representation, as determined by their reference counts, are returned to the available space list and the reference counts of other cells in X are decreased by one as appropriate.

ERLA(X). X is the location of an arbitrary list of atoms, i.e., a list of order one. X is erased; i.e., the effect of ERLA is the same as that of ERASE. However, knowledge of the structure of X is used by ERLA, resulting in appreciably faster execution.

Z=INV(X). X is the location of a list $(x_1,\ldots,x_n)$, possibly null. The given list is modified into the inverse list $(x_n,\ldots,x_1)$, whose location, Z, is returned as value. The given list X no longer exists. The inversion is carried out by modifying the successor field only of each of the n cells in the main level of the given list. Hence INV is appropriately applied only to a list all of whose main level reference counts are one (typically a list which has just been constructed).

Z=LENGTH(X). X is an arbitrary list $(x_1,\ldots,x_n)$. Z is the length, n, of X, expressed as a Fortran integer.

NOAVLS. A system subroutine called by PFA or PFL whenever the available space list is empty. It prints the diagnostic "out of available space" and terminates job execution.

Z=PFA(X,Y). X is an atom, Y is the location of an arbitrary list $(y_1,\ldots,y_n)$. X is prefixed to the list Y, resulting in the list $(X,y_1,\ldots,y_n)$. The given representation of Y remains intact, and its reference count remains unchanged. The reference count of the resulting list is one and the value Z returned is its location. A single cell is removed from the available space list, into which is stored the element X.

Z=PFL(X,Y). PFL is exactly like PFA except that here the argument X is a list. The reference count of X is not modified, since typically the list X is not independently needed, or else its reference count has previously been augmented by an application of BORROW. Note that, if $X = (x_1,\ldots,x_m)$ and $Y = (y_1,\ldots,y_n)$, then the resulting list is $(X,y_1,\ldots,y_n)$, a list of length n+1 whose first element is the list X, _not_ the list $(x_1,\ldots,x_m,y_1,\ldots,y_n)$ as in concatenation.

STACK2 (X,Y). X and Y are placed on the system pushdown stack - first X, then Y - by two applications of PFA. X and/or Y may, however, be the locations of lists; this causes no difficulty since the pushdown stack is never processed as a general list in which the types of the elements must be checked.

STACK3(X,Y,Z). Like STACK2 except for the third argument Z, which is stacked last.

UNSTK2(X,Y). The top element on the pushdown stack is removed (via DECAP) and assigned to Y; then the next is removed and assigned to X. Note that a CALL STACK2(X,Y) will normally be paired with a CALL UNSTK2(X,Y) - as opposed to CALL UNSTK2(Y,X).

UNSTK3(X,Y,Z). The extension of UNSTK2 to three arguments. Z is unstacked first.

7. Recursion in SAC-1

One of the main deficiencies in A.S.A. Fortran as a language for list processing and for symbolic and algebraic manipulation is its failure to provide for recursive subprograms (see [3], p. 615 and p. 618). For use in SAC-1 a method has been developed for circumventing the prohibition against, and lack of provision for, recursion. The method is somewhat inconvenient to use, but completely general. The SLIP system, [5], provides a mechanism for recursion in Fortran, but it requires the use of two additional primitives; moreover these primitives violate certain A.S.A restrictions and, indeed, there is no assurance that these primitives can be realized in a way that is consistent with an arbitrary Fortran compiler.

A subprogram is _recursive_ in case the completion of a particular execution of that subprogram may require the prior initiation of a new execution. This may come about either _directly_, as a result of the subprogram referencing itself, or _indirectly_, as when the given subprogram references another subprogram which in turn references the former (either directly or indirectly). Recursion in Fortran poses three distinct problems, which we shall now discuss, one by one.

The first problem is the syntactic prohibition, stated in [3], that a subprogram shall not reference itself, directly or indirectly. Its effect is that any subprogram which references itself directly will likely result in a diagnostic, and will fail to be compiled. The syntactic prohibition against indirect recursive reference, on the other hand, is unenforceable since its detection would require that the several subprograms in the circular chain of references be compiled simultaneously. Since a direct self-reference is easily translatable into an indirect self-reference, by the introduction of a new subprogram which does nothing more than to call the original subprogram with the same argument list, this first problem is easily circumventable.

The other two problems pertain to possible inadequacies in the compiled programs. When an interrupted execution is later resumed, it is essential that all its variables have the same definitional status and values as prior to the interruption, although these may have been altered by intervening executions. The A.S.A. specifications do not require this restoration and it must be explicitly programmed. The solution to this problem is quite simple with the use of a pushdown stack.

The SAC-1 system has a single pushdown stack, which is in the form of a list of order one, in which all reference counts are one. Values are placed on the stack by prefixing them to the list, and removed by the list decapitation process. The location of the pushdown stack is kept in the second memory location of the labelled common block TR1, bearing the symbolic name STAK. Special system subroutines, STACK2, STACK3, UNSTK2 and UNSTK3, described in Section 6, are provided to facilitate stacking and unstacking. Just prior to a recursive reference, current values of program variables are stacked and, immediately following the recursive reference, these values are unstacked and reassigned to the same variables. Not every program variable needs to be so preserved and restored - only those whose current value will be used in completing the interrupted execution. This will be illustrated in the example below.

The third problem connected with recursion pertains to the flow of control, and the Fortran Return statement. When a subprogram may have several executions in progress simultaneously, there is associated with each execution a return location which must be saved along with the current values of program variables. Since, however, these return locations are not, in Fortran, associated with any program variable, it is impossible to use the stack mechanism. We believe, in fact, that, for this reason, it is impossible to program in A.S.A. Fortran a recursive subprogram in the strict sense. We can, however, devise an effective substitute for a recursive subprogram, which we call a recursive procedure.

A <u>recursive procedure</u> is a sequence of executable statements together with a list of variables occurring in these statements, which function as procedure parameters. Among the procedure parameters, one acts as the return location of the procedure. In the case of a function procedure, one also serves to carry the function value. All other procedure parameters play the roles of the dummy arguments of a subprogram.

A recursive procedure has all the essential features of a subprogram and, in addition, a variable associated with its return location. The actual value of the return location variable is not a location, but a statement number, or a code number for use in a computed-go-to statement. This value can be stacked along with the values of other variables. An inessential, although very convenient, feature of subprograms, not shared by recursive procedures, is the automatic association of actual arguments with dummy arguments, as well as the possession of a symbolic name.

We now illustrate the use of recursive procedures by an example. Suppose we wish to write a function subprogram Z=ORDER(X) which, when applied to an arbitrary list X, returns as value the order, Z, of X, as a Fortran integer. This function is naturally defined by recursion since the order of X is just one greater than the maximum order of any element of X which is a list, or one if all elements of X are atoms. Thus, if recursion were permissible, we might define ORDER as follows.

```
                INTEGER FUNCTION ORDER(X)
                INTEGER FIRST, TAIL, TYPE
                INTEGER X,T
                T = X
                N = o
                IF  (T.EQ.o)  GO TO 3
1               IF  (TYPE(T).EQ.o) GO TO 2
                M = ORDER (FIRST(T))
                IF  (M.GT.N)  N = M
2               T = TAIL(T)
                IF  (T.NE.o)  GO TO 1
3               ORDER = N + 1
                RETURN
                END
```

We now rewrite the ORDER subprogram, so that it contains within it a recursive procedure for the order function and is not, itself, recursive. We first choose the procedure parameters: Y for the function argument, N for the function value, and R for the return location. The procedure will be "called" from two places: from outside the procedure, but within the ORDER subprogram and from just one place, in this case, within the procedure itself. We assign code numbers, 1 and 2, to these two places. The following program is obtained, in which comments have been inserted to delimit the structural parts.

```
                INTEGER FUNCTION ORDER(X)
                INTEGER FIRST, TAIL, TYPE
                INTEGER X,T,Y,R,E
C               PROCEDURE CALL
                Y = X
```

```
            R = 1
            GO TO 4
 5          ORDER = N
 C          END PROCEDURE CALL
            RETURN
 C          RECURSIVE PROCEDURE N = ORDER(Y), RETURN TO R
 4          T = Y
            N = o
            IF (T.EQ.o)  GO TO 3
 1          IF (TYPE(T).EQ.o)  GO TO 2
            E = FIRST(T)
 C          RECURSIVE PROCEDURE CALL
            CALL STACK3 (N,T,R)
            Y = E
            R = 2
            GO TO 4
 6          M = N
            CALL UNSTK3(N,T,R)
 C          END RECURSIVE PROCEDURE CALL
            IF (M.GT.N)  N = M
 2          T=TAIL(T)
            IF (T.NE.o)  GO TO 1
 3          N = N + 1
 C          PROCEDURE RETURN
            GO TO (5,6), R
 C          END RECURSIVE PROCEDURE N=ORDER(Y), RETURN TO R
            END
```

This program is by no means optimal because, by intent, it illustrates a more or less algorithmic translation of the former program. Also, this simple function could be quite easily programmed without the aid of recursion. But there are much more complex situations where recursion is practically indisdensable.

The example above illustrates the case of direct recursion. In the general case of indirect recursion, one has subprograms $A_1,\ldots,A_n$ such that A_i calls A_{i+1}, for $1 \leq i < n$, and A_n calls A_1. It is then necessary to set up procedures for each A_i within a single subprogram, say B. In addition, there is a subprogram for each A_i, the sole function of which is to accept computational requests from the system and relay them to subprogram B for the actual computation, together with a code number i, which tells subprogram B which procedure to invoke.

Suppose, for simplicity, that n = 2 and that each A_i is a function of one argument. Subprograms A1, A2 and B (in outline) would then appear as follows.

```
INTEGER FUNCTION A1(X)
INTEGER X,B
A1 = B(1,X)
RETURN
END

INTEGER FUNCTION A2(X)
INTEGER X,B
A2 = B(2,X)
RETURN
END
```

```
                  INTEGER FUNCTION B(I,X)
                  INTEGER X,U,R,W,V,S,Z,...
                  IF (I.EQ.2)  GO TO 2
      1           U = X
                  R = 1
                  GO TO 3
      4           B = W
                  RETURN
      2           V = X
                  S = 2
                  GO TO 5
      6           B = Z
                  RETURN
      3           (procedure A1)
      5           (procedure A2)
                  END
```

Here U, R and W (V,S and Z) are the argument, return location, and value variables for procedure A1 (A2).

The above programs serve to illustrate another programming device not related to recursion, but to another inconvenience of Fortran. In a system such as SAC-1, we generally want to eliminate the distinction of the various types of variables because it may, for example, result in unwanted conversions from fixed to floating-point, or vice versa. Since we do, however, wish some data to be interpreted as integers, the solution is to disguise all data to Fortran as integers. It is unnatural and anti-mnemonic to begin every variable and function name with I, J, K, L, M or N. So we choose identifiers freely and, after writing any program or subprogram, list in an integer declaration all identifiers which occur. This is a simple mechanical process which has been found to be non-error-inducing, and is recommended to users of SAC-1 as well. It also results in the useful byproduct that all function subprograms referenced by a given subprogram are listed at the beginning of the subprogram; we usually list them in a separate declaration.

8. The SAC-1 Character Set and Internal Code

Since different computers have different character sets and, also, different internal codes for the same characters, some scheme is required for maintaining computer independence in the SAC-1 input-output system. The obvious solution to the problem of differing character sets is the choice of a small set of characters common to all computers. Such a choice is provided by the A.S.A. Fortran character set ([3], p. 593), consisting of all the characters used in writing an A.S.A. Fortran program. The adoption of this set as the SAC-1 character set implies that all input and output data processed by SAC-1 will consist of these characters only.

A common internal code for these characters is also required. Since the internal hardware codes of different computers are inconsistent, a conversion from the hardward code to the SAC-1 code upon input, and the inverse conversion just prior to output, is, in general, required. Hence, instead of choosing the hardware code of one particular computer as the SAC-1 code, a code has been chosen which facilitates the kind of character testing required in the SAC-1 input conversion subprograms. In this code, functionally similar characters (digits, letters, operators and punctuators) are assigned contiguous codes. The coding system is given in the following table.

Characters	Decimal Codes
0 - 9	0 - 9
A - Z	10 - 35
+ - * /	36 - 39
( ) , . blank	40 - 44
=	45
$	46

9. The Primitive Subroutines READ and WRITE.

The READ subroutine reads a standard size record from a specified unit U, performs the conversion from hardware code to SAC-1 code, and delivers the input characters as the elements of an array A . Correspondingly, the WRITE subroutine converts the elements of an array A from SAC-1 code to hardware code, and writes the result on unit U as a standard size record. U and A are the subroutine arguments: READ(U,A), WRITE(U,A).

The standard record length is 72. Thus, A is a one-dimensional integer array of 72 elements. Input records may be longer (e.g. 80 in the case of card-generated input), but only the first 72 characters will read and stored in A . Output records generated by WRITE will be of either length 72 or 73 , depending on the unit number U . Certain unit numbers, according to local option (as reflected in the implementation of WRITE), are regarded as designating printer output units (either printers or tapes destined for printing). When such a unit is designated, a record of length 73 is written of which the first character is blank, resulting in single line printer spacing, and A(I) corresponds to the (I + 1)- th character of the record for $1 \leq I \leq 72$. Otherwise, A(I) always corresponds to the I-th character of the record (input or output). A 73-character record so written cannot, of course, be later read as meaningful input.

If, when the READ subroutine is called, unit U is positioned at an end-of-file, A(1) is given a value of -1 as a means of relaying this information to the calling program. Execution of WRITE may result in redefining the array A (e.g., A may then contain the hardware code for the record written).

Subprograms provided as parts of the SAC-1 system will process only records containing only characters from the Fortran character set. However, individual users of SAC-1 may wish to make extensions of the system to process additional characters. This can be done by assigning codes to these characters in the range 47-63 and implementing READ and WRITE so as to perform the required code conversions for the expanded character set.

10. Reading and Writing Lists

SAC-1 contains two subprograms, LREAD and LWRITE, for reading and writing arbitrary lists. This requires that an arbitrary list have an "external representation" as a sequence of characters. This external representation has been illustrated in Section 2 for the case that atoms are integers represented in standard decimal notation. In SAC-1, atoms are always sequences of m binary digits, where m is the word length of a particular computer.

(We assume throughout this section that SAC-1 is implemented on a binary computer. All parts of SAC-1 other than those described in this section are independent of this assumption, including the parts to be described in subsequent papers. The subprograms here described are useful, but inessential, adjuncts to the system.)

These m binary digits are clustered into groups of three, thereby determining a sequence of octal digits. (If the word length is not a multiple of 3, one or two leading binary zeros are considered implicit.) The external representation is then a sequence of left and right parentheses, commas, and octal digits. The main list and its constituent lists are delimited by parentheses; list elements are separated by commas. Since a sequence of octal digits representing an atom will, in common practice, frequently have several leading zeros, special provision is made for their suppression. In input to LREAD, any number of leading zeros may optionally be suppressed. In output produced by LWRITE all leading zeros will be suppressed. An exception to this rule is that if all digits are zero, one zero must be retained (and is retained by LWRITE). The number of characters in an external list representation may be arbitrarily large, and hence span an arbitrary number of records. The first character always occurs in the first position of the first record, the 73rd character, if it exists, occurs in the first position of record two, etc. The last record is filled out with blanks as required; intervening blanks are not permitted.

Z=LREAD(U) is a function subprogram which reads a list from unit U and generates an internal representation of the list read, returning the location of the internal representation as the value Z. If, when LREAD was called, unit U was positioned at an end-of-file, then the value Z returned is instead -1.

LREAD makes various checks on validity of the input list. If the input is invalid, then the value $Z = -2$ is returned.

LWRITE(U, X) is a subroutine, which converts the list X to external representation and writes it on unit U . The internal representation remains intact and unerased. Since LWRITE uses WRITE, the record length may be either 72 or 73 depending on U .

Three additional primitive subprograms, LSHIFT, RSHIFT and NBPW, are required by LREAD and LWRITE. These primitives are not used elsewhere in SAC-1 and are therefore dispensable if LREAD and LWRITE are omitted from the system. LREAD and LWRITE serve two main purposes: debugging and the conservation of addressable memory. LREAD may be used to input simple test lists. LWRITE may be used to output intermediate results during program testing. Main memory may be conserved during a calculation by writing lists temporarily on tape, erasing them, and later reading them with LREAD. Later increments to SAC-1 will contain input-output subprograms for lists representing particular types of objects, e.g., large integers, polynomials, rational functions, etc., but the I/O conversions employed are more complex and time consuming, and are not needed for saving intermediate results on tape.

Z=LSHIFT(X, Y, N) is a function subprogram in which $X=X_1 \ldots X_m$ and $Y=Y_1 \ldots Y_m$ are atoms, regarded as sequences of bits, and $N = n$ is a shift amount, expressed as a Fortran integer. Z is the atom resulting from left-shifting X n places and inserting the last n bits of Y into the vacated bit positions. I.e., $Z = X_{n+1} \ldots X_m Y_{m-n+1} \ldots Y_m$. X and Y are not redefined.

RSHIFT(X, Y, N) is a subroutine in which $X = X_1 \ldots X_m$ and Y are atoms and $N = n$ is a shift amount. X is right-shifted n places so that the redefined value of X is $0 \ldots 0\, X_1 \ldots X_{m-n}$. Y need not be initially defined; its resulting value is $0 \ldots 0\, X_{m-n+1} \ldots X_m$.

Z = NBPW(X) is a function subprogram whose argument X is ignored and whose value Z is the number of bits per word in the computer of implementation, expressed as a Fortran integer.

LREAD and LWRITE depend on one additional assumption, which holds for all known computers, but which should probably be made explicit, nevertheless. This assumption is that non-negative Fortran integers in the range $0 \le N \le 7$ are represented in standard base two notation; i.e., 0 by 0 ... 0, 1 by 0 ... 0 1, 2 by 0 ... 010, etc.

References

1. Collins, George E. PM, A System for Polynomial Manipulation. Comm. A.C.M., Vol. 9, No. 8 (Aug. 1966), pp. 578-589.

2. Knuth, D. E., The Art of Computer Programming, Vol. 1 (Fundamental Algorithms), Additon-Wesley, 1968.

3. Fortran vs. Basic Fortran. Comm. A.C.M., Vol. 7, No. 10 (Oct. 1964), pp. 592-625.

4. Collins, George E. A Method for the Overlapping and Erasure of Lists, Comm. A.C.M., Vol. 3, No. 12 (Dec. 1960), pp. 655-657.

5. Weizenbaum, J. Symmetric List Processor. Comm. A.C.M., Vol. 6, No. 9 (Sept. 1963), pp. 524-544.

6. McCarthy, John, et. al. LISP 1.5 Programmer's Manual. M.I.T. Press, 1962.

7. Collins, George E., and James R. Pinkert,The Revised SAC-1 Integer Arithmetic System, Univ. of Wisconsin Computer Center Technical Report No. 9, Nov. 1968, 50 pages.

8. Collins, George E., The SAC-1 Polynomial System, Univ. of Wisconsin Computing Center Technical Report No. 2, March 1968, 68 pages.

9. Collins, George E., The SAC-1 Rational Function System, Univ. of Wisconsin Computing Center Technical Report No. 8, July 1968, 31 pages.

10. Collins, George E., L. E. Heindel, E. Horowitz, M. T. McClellan, and D. R. Musser, The SAC-1 Modular Arithmetic System, Univ. of Wisconsin Computing Center Technical Report No. 10, June 1969, 50 pages.

11. Collins, George E., and Ellis Horowitz, The SAC-1 Partial Fraction Decomposition and Rational Function Integration System, Univ. of Wisconsin Computing Center Technical Report No. 12, Feb. 1970, 47 pages.

12. Collins, George E., and Lee E. Heindel, The SAC-1 Polynomial Real Zero System, Univ. of Wisconsin Computing Center Technical Report No. 18, Aug. 1970, 72 pages.

```
      SUBROUTINE ADV(EL,LOC)
      COMMON /TR1/AVAIL,STAK,RECORD(72)
      INTEGER AVAIL,STAK,RECORD
      INTEGER EL,LOC,L,TAIL,FIRST
      L=LOC
      EL=FIRST(L)
      LOC=TAIL(L)
      RETURN
      END

      SUBROUTINE BEGIN(ARRAY,N)
      COMMON /TR1/AVAIL,STAK,RECORD(72)
      INTEGER AVAIL,STAK,RECORD
      INTEGER ARRAY
      DIMENSION ARRAY(N)
      J=NWPC(L)
      L=LOC(ARRAY(1))
      M=N-J
      AVAIL=L
      DO 1 K=1,M,J
      LN=LOC(ARRAY(K+J))
      CALL SSUCC(LN,L)
      CALL SCOUNT(0,L)
      CALL STYPE(0,L)
      CALL ALTER(0,L)
1     L=LN
      CALL SSUCC(0,L)
      CALL SCOUNT(0,L)
      CALL STYPE(0,L)
      CALL ALTER(0,L)
      STAK=0
      RETURN
      END

      INTEGER FUNCTION BORROW(LIST)
      COMMON /TR1/AVAIL,STAK,RECORD(72)
      INTEGER AVAIL,STAK,RECORD
      INTEGER COUNT
      BORROW=LIST
      IF(BORROW.EQ.0)RETURN
      CALL SCOUNT(COUNT(BORROW)+1,BORROW)
      RETURN
      END

      INTEGER FUNCTION CINV(X)
      COMMON /TR1/AVAIL,STAK,RECORD(72)
      INTEGER AVAIL,STAK,RECORD
      INTEGER TYPE,PFA,PFL,BORROW
      INTEGER X,Z,R,E,T
      Z=0
      R=X
1     IF (R.EQ.0) GO TO 3
      T=TYPE(R)
      CALL ADV(E,R)
      IF (T.EQ.1) GO TO 2
      Z=PFA(E,Z)
      GO TO 1
2     Z=PFL(BORROW(E),Z)
      GO TO 1
3     CINV=Z
      RETURN
      END

      INTEGER FUNCTION CONC(X,Y)
      COMMON /TR1/AVAIL,STAK,RECORD(72)
      INTEGER AVAIL,STAK,RECORD
      INTEGER X,Y,XAD,XNEXT,XSUC,TAIL
      CONC=Y
      XSUC=CONC
      XNEXT=X
      IF(XNEXT.EQ.0)RETURN
      CONC=XNEXT
      IF(XSUC.EQ.0)RETURN
1     XAD=XNEXT
      XNEXT=TAIL(XNEXT)
      IF(XNEXT.NE.0)GO TO 1
      CALL SSUCC(XSUC,XAD)
      RETURN
      END

      SUBROUTINE DECAP(EL,LOC)
      COMMON /TR1/AVAIL,STAK,RECORD(72)
      INTEGER AVAIL,STAK,RECORD
      INTEGER B,EL,TAIL,FIRST
      B=AVAIL
      AVAIL=LOC
      EL=FIRST(AVAIL)
      LOC=TAIL(AVAIL)
      CALL SSUCC(B,AVAIL)
      CALL SCOUNT(0,AVAIL)
      RETURN
      END

      SUBROUTINE ERASE(XX)
      COMMON /TR1/AVAIL,STAK,RECORD(72)
      INTEGER AVAIL,STAK,RECORD
      INTEGER COUNT,FIRST,TAIL,TYPE
      INTEGER X,XX,T,BIN
      X=XX
      BIN=0
1     IF(X.EQ.0)GO TO 3
2     K=COUNT(X)-1
      CALL SCOUNT(K,X)
      IF(K.GT.0)GO TO 3
      IF(TYPE(X).EQ.1)GO TO 4
      T=TAIL(X)
      CALL SSUCC(AVAIL,X)
      AVAIL =X
      X=T
      GO TO 1
3     IF(BIN.EQ.0)RETURN
      X=FIRST(BIN)
      T=TAIL(BIN)
      CALL SSUCC(AVAIL,BIN)
      AVAIL=BIN
      BIN=T
      GO TO 1
4     T=TAIL(X)
      CALL SSUCC(BIN,X)
      BIN=X
      X=T
      GO TO 1
      END

      SUBROUTINE ERLA(X)
      COMMON /TR1/AVAIL,STAK,RECORD(72)
      INTEGER AVAIL,STAK,RECORD
      INTEGER X,U,V,TAIL,COUNT
      IF(X.EQ.0)GO TO 3
      U=0
      V=X
1     N=COUNT(V)-1
      CALL SCOUNT(N,V)
      IF(N.NE.0)GO TO 2
      U=V
      V=TAIL(V)
      IF(V.NE.0)GO TO 1
2     IF(U.EQ.0)GO TO 3
      CALL SSUCC(AVAIL,U)
      AVAIL=X
3     RETURN
      END

      FUNCTION INV(LIST)
      COMMON /TR1/AVAIL,STAK,RECORD(72)
      INTEGER AVAIL,STAK,RECORD
      INTEGER TAIL
      INV=0
      NEXT=LIST
      IF(NEXT.EQ.0)RETURN
1     ISTORE=NEXT
      NEXT=TAIL(ISTORE)
      CALL SSUCC(INV,ISTORE)
      INV=ISTORE
      IF(NEXT.NE.0)GO TO 1
      RETURN
      END

      FUNCTION LENGTH (LIST)
      COMMON /TR1/AVAIL,STAK,RECORD(72)
      INTEGER AVAIL,STAK,RECORD
      INTEGER TAIL
      LENGTH=0
      K=LIST
1     IF(K.EQ.0)RETURN
      K=TAIL(K)
      LENGTH=LENGTH+1
      GO TO 1
      END

      INTEGER FUNCTION LREAD(U)
      COMMON /TR1/AVAIL,STAK,RECORD(72)
      INTEGER AVAIL,STAK,RECORD
      INTEGER PFA,PFL,INV,LSHIFT,NBPW
      INTEGER B,C,D,E,P,T,U,X,Y,Z
      T=0
      X=0
      Z=0
      I=1
      B=NBPW(X)
      J=B
      CALL READ(U,RECORD)
      LREAD=-1
      IF (RECORD(1).EQ.-1) RETURN
      IF (RECORD(1).NE.40) GO TO 90
10    P=RECORD(I)
      I=I+1
      IF (I.LE.72) GO TO 11
      I=1
      CALL READ(U,RECORD)
      IF (RECORD(1).LT.0) GO TO 90
11    C=RECORD(I)
      IF (P.EQ.40) GO TO 20
      IF (P.EQ.41) GO TO 21
      IF (P.EQ.42) GO TO 22
      GO TO 23
20    IF (C.EQ.40) GO TO 30
      IF (C.EQ.41) GO TO 31
      IF (C.LT.8) GO TO 32
      GO TO 90
21    IF (C.EQ.41) GO TO 31
      IF (C.EQ.42) GO TO 10
      GO TO 90
22    IF (C.EQ.40) GO TO 30
      IF (C.LT.8) GO TO 32
      GO TO 90
23    IF (C.EQ.41) GO TO 33
      IF (C.EQ.42) GO TO 33
      IF (C.LT.8) GO TO 32
      GO TO 90
30    Z=PFL(X,Z)
      X=0
      GO TO 10
31    X=INV(X)
      IF (Z.EQ.0) GO TO 80
      CALL DECAP(Y,Z)
      X=PFL(X,Y)
      GO TO 10
 32   T=PFA(C,T)
      J=J-3
      GO TO 10
 33   IF (J.LT.-2) GO TO 90
      T=INV(T)
 35   IF (J.LE.0) GO TO 36
      T=PFA(0,T)
      J=J-3
      GO TO 35
 36   CALL DECAP(D,T)
      E=LSHIFT(E,D,3)
      IF (T.NE.0) GO TO 36
      J=B
      X=PFA(E,X)
      IF (C.EQ.41) GO TO 31
      GO TO 10
 80   LREAD=X
      RETURN
 90   CALL ERLA(T)
      CALL ERASE(X)
      CALL ERASE(Z)
      LREAD=-2
      RETURN
      END

      SUBROUTINE LWRITE(U,Y)
      COMMON /TR1/AVAIL,STAK,RECORD(72)
      INTEGER AVAIL,STAK,RECORD
      INTEGER NBPW,FIRST,TAIL,TYPE,PFA
      INTEGER B,C,D,S,T,U,X,Y
      S=0
      X=Y
      I=1
      B=NBPW(T)
 10   C=40
      GO TO 50
 11   IF (X.NE.0) GO TO 12
 20   C=41
      GO TO 50
 13   IF (S.EQ.0) GO TO 60
      CALL DECAP(X,S)
 14   X=TAIL(X)
      IF (X.EQ.0) GO TO 20
 40   C=42
      GO TO 50
 12   T=FIRST(X)
 15   IF (TYPE(X).EQ.0) GO TO 30
      S=PFA(X,S)
      X=T
      GO TO 10
 30   L=0
      DO 31 J=1,B,3
      CALL RSHIFT(T,D,3)
```

```
31      L=PFA(D,L)
32      CALL DECAP(C,L)
        IF (C.NE.0) D=C
        IF (L.EQ.0) GO TO 50
        IF (D.NE.0) GO TO 50
        GO TO 32
33      IF (L.NE.0) GO TO 32
        GO TO 14
50      IF (I.LE.72) GO TO 51
        CALL WRITE(U,RECORD)
        I=1
51      RECORD(I)=C
        I=I+1
        IF (C.LT.8) GO TO 33
        IF (C.EQ.42) GO TO 12
        IF (C.EQ.41) GO TO 13
        GO TO 11
60      IF (I.EQ.73) GO TO 62
        DO 61 J=I,72
61      RECORD(J)=44
62      CALL WRITE(U,RECORD)
        RETURN
        END

        SUBROUTINE NOAVLS
        COMMON /TR1/AVAIL,STAK,RECORD(72)
        INTEGER AVAIL,STAK,RECORD
         PRINT 1
         STOP
1       FORMAT (23H OUT OF AVAILABLE SPACE)
         END

        INTEGER FUNCTION PFA(ATOM,LIST)
        COMMON /TR1/AVAIL,STAK,RECORD(72)
        INTEGER AVAIL,STAK,RECORD
        INTEGER ATOM,TAIL
        PFA=AVAIL
        IF(PFA.EQ.0)CALL NOAVLS
        AVAIL=TAIL(AVAIL)
        CALL SSUCC(LIST,PFA)
        CALL STYPE(0,PFA)
        CALL SCOUNT(1,PFA)
        CALL ALTER(ATOM,PFA)
        RETURN
        END

        INTEGER FUNCTION PFL(LIST1,LIST)
        COMMON /TR1/AVAIL,STAK,RECORD(72)
        INTEGER AVAIL,STAK,RECORD
        INTEGER TAIL
        PFL=AVAIL
        IF(PFL.EQ.0)CALL NOAVLS
        AVAIL=TAIL(AVAIL)
        CALL SSUCC(LIST,PFL)
        CALL STYPE(1,PFL)
```

```
      CALL SCOUNT(1,PFL)
      CALL ALTER(LIST1,PFL)
      RETURN
      END

      SUBROUTINE STACK2(X,Y)
      COMMON /TR1/AVAIL,STAK,RECORD(72)
      INTEGER AVAIL,STAK,RECORD
      INTEGER X,Y,PFA
      STAK=PFA(Y,PFA(X,STAK))
      RETURN
      END

      SUBROUTINE STACK3(X,Y,Z)
      COMMON /TR1/AVAIL,STAK,RECORD(72)
      INTEGER AVAIL,STAK,RECORD
      INTEGER X,Y,Z,PFA
      STAK=PFA(Z,PFA(Y,PFA(X,STAK)))
      RETURN
      END

      SUBROUTINE UNSTK2(X,Y)
      COMMON /TR1/AVAIL,STAK,RECORD(72)
      INTEGER AVAIL,STAK,RECORD
      INTEGER X,Y
      CALL DECAP(Y,STAK)
      CALL DECAP(X,STAK)
      RETURN
      END

      SUBROUTINE UNSTK3(X,Y,Z)
      COMMON /TR1/AVAIL,STAK,RECORD(72)
      INTEGER AVAIL,STAK,RECORD
      INTEGER X,Y,Z
      CALL DECAP(Z,STAK)
      CALL DECAP(Y,STAK)
      CALL DECAP(X,STAK)
      RETURN
      END
```