Dedicated
to my
Mother

THE COMMUNICATION MANAGEMENT SYSTEM (COMS)

AN INVESTIGATION INTO

THE

COMMUNICATION MANAGEMENT SYSTEM (COMS)

By

MARC STEPHEN BADER, B.Sc.

A Project

Submitted to the School of Graduate Studies

in Partial Fulfilment of the Requirements

for the Degree

Master of Science

McMaster University

June 1973

MASTER OF SCIENCE
(Computation)

McMASTER UNIVERSITY
Hamilton, Ontario.

TITLE: An Investigation Into The Communication Management System (COMS)

AUTHOR: Marc Stephen Bader, B.Sc. (McMaster University)

SUPERVISOR: Dr. Nicholas Solntseff

NUMBER OF PAGES: ix, 112, A(3), B(3), C(56), D(1), E(43), F(100), G(3), H(17)

ABSTRACT

This report is concerned with an investigation into a software system designed to allow effect utilization of FORTRAN application programs from a library. The components of this system consist of an interpreter program to manipulate character strings and provide overall control, an evaluator program to carry out operations on numeric data and to provide for the calling of library programs, and an associative memory to store and retrieve facts about the environment or field of study in which the system is being used. Details involving how to use each component and how each component works are discussed. Possible improvements to the system and the relationship of the system to the field of control structures are also considered. The implementation of the system is discussed and this leads to an examination of the algorithms used in the operation of the system. Control is easily maintained so systems constructed from the components may be modified or extended by any user. Thus, these components form a basis for a class of extendable systems.

Acknowledgments

The author gratefully expresses sincere thanks to Dr. N. Solntseff for his guidance through this project and to the National Research Council of Canada for the computer time provided by their financial support.

Special thanks to Mr. Bill Bell whose programming suggestions helped a great deal.

For the manuscript typing the author is indebted to Miss Linda Westfall; and finally for her concern and encouragement in this endeavour, thanks to Miss Debbie Westlake.

TABLE OF CONTENTS

TABLE OF DIAGRAMS

PREFACE

This project involves the study of a group of basic computer programs and methods collectively called the Communication Management System (COMS). Originally designed and implemented in 1969 by Robert C. Gammill at the University of Colorado, COMS was used to develop methods of computer utilization which would allow application programs produced by experienced programmers for different fields of study to be used by other interested people, who, having very little knowledge of computers and computer programming, never had access to such programs before. The result is a system which creates an environment that encourages the authors of application programs to write them as general purpose subroutines, and which allows the inexperienced computer user to operate such application programs with little knowledge of their intricacies.

CHAPTER 1

THE COMMUNICATION MANAGEMENT SYSTEM

1.1 Introduction

The software elements (programs) which make up the COMS system were originally implemented in PL/1 on an IBM 360/65 computer. A second but incomplete Fortran IV implementation was carried out on a CDC 6600 computer. Part of this project involved completing the previous Fortran version and reimplementing it on the CDC 6400 computer at McMaster University.

The individual program elements of COMS are discussed in great detail in Chapter 1 with respect to their function in the COMS system, their relationship to each other and the methods involved in their use. Investigation of a possible improvement in the system is carried out in Chapter 2 showing how particular programming control structures may be used to attain a greater degree of efficiency. Also examined here is the possible use of COMS in the development of a command language for an operating system used in a parallel programming environment. The appendices following Chapter 2 are used to provide summaries of important aspects of the COMS system, to display sample programs involving a variety of applications of the system and to give a detailed account of the program flow of each of the COMS software elements.

1.2. Description of COMS

COMS is a software package which consists of three major programs. These include an Interpreter program which serves as the controlling element of COMS, an Associative Memory program which stores factual information about COMS, its library and the environment in which it is being used, and an Evaluator program which evaluates algebraic formulas, stores and retrieves numeric data, and causes execution of FORTRAN programs from a library. Associated with each of these programs is a data collection as shown in Figure 1.1.

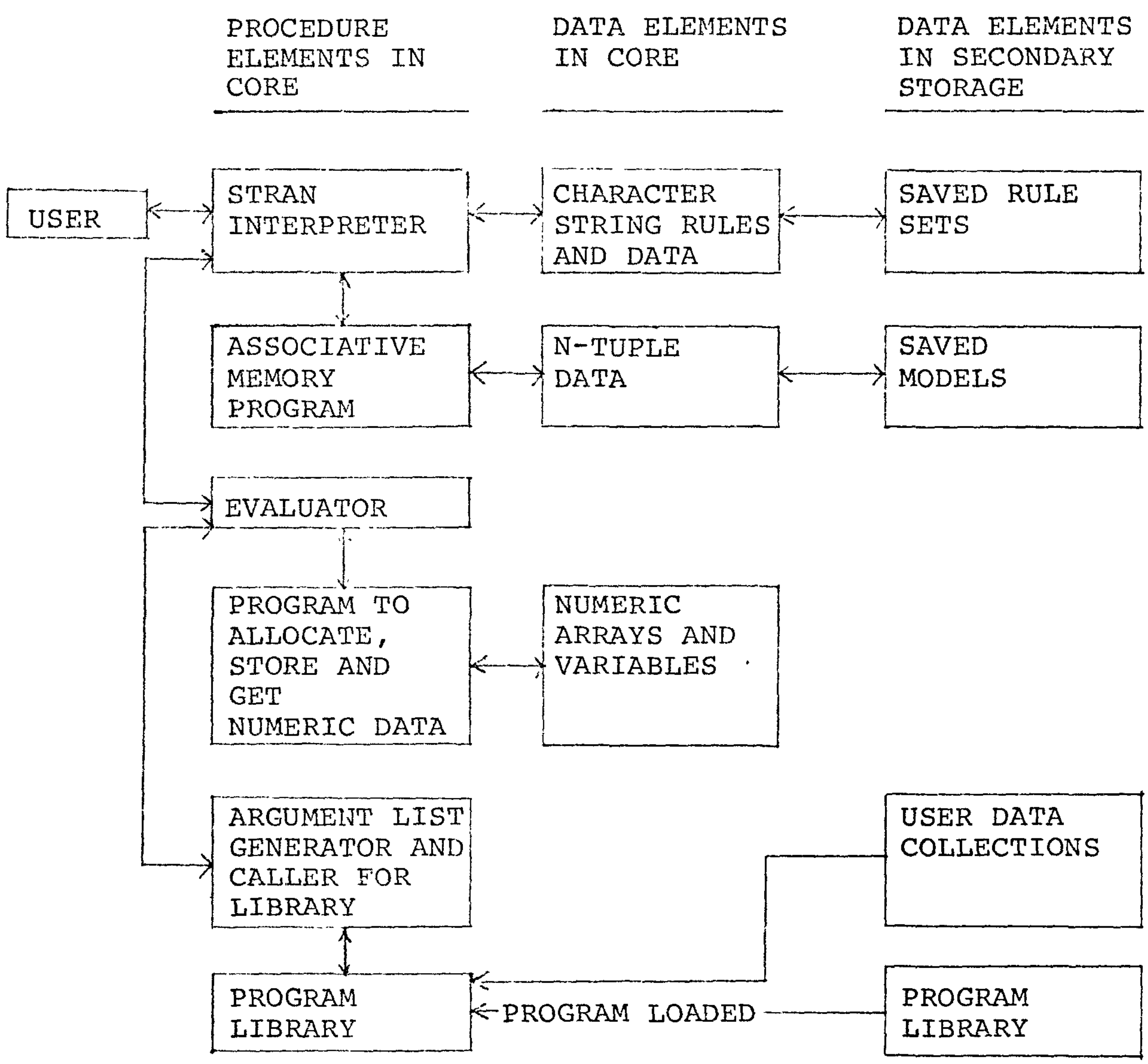
The major aim of COMS is effective utilization of FORTRAN programs from a program library. These programs are supplied by both those who design the COMS system and those who use it.

To be helpful to programmers in all fields of study, COMS has been designed to be changed. This is because COMS is data directed through commands to the interpreter and these commands can be modified or added to by anyone.

One way to view the COMS system is by an analogous comparison to a book-type library, where the man in charge is say Mr. X. Now Mr. X (the computer scientist) is a busy man dealing with not one but many such libraries (computer systems) and he knows that for each of his libraries to be useful to the public, books (application programs) cannot be blindly thrown into rooms where readers (people wanting to

FIGURE 1.1

COMS - DETAILED BLOCK DIAGRAM



use the application programs) are expected to rummage through the mess to find what they want. Mr. X also knows that he hasn't the time to take each person (each programmer) and show them the exact location of each book they wish to read (that is, the computer scientist is too busy to show each programmer how to use a particular application program). Instead, Mr. X must create a system (COMS) which allows library users (programmers) to find and use what they need on their own. Thus Mr. X hires librarians (system programmers) and provides them with money (COMS programs) to develop such a system. The librarian can then transform the library books (application programs) into an effective tool for different readers' (users') goals. The authors of the books in the library are analogous to the authors of the application programs.

The important point in the above comparison is that the person in a particular field of study that desires the use of computing facilities but does not have any idea how to use them is provided with a tool for just this purpose. Once COMS has been set up by experienced programmers for use in a particular field, anyone involved in that field will have the availability of computing facilities that did not exist previously due to the lack of programming "know-how".

Other software systems of the COMS variety[1] have been developed and aimed at people who know little about computers but are knowledgeable in the problem area of interest. None of these however, offer as much user flexibility as that given in COMS. Also, in some cases, application programs in the field of interest which are to be added to the system have to be massively rewritten to conform to restrictive conventions and standards. The result with many systems (and this is definitely not true of COMS) is that they have remained alive only as long as the people who originally designed them continued to work on and develop the system.

In summary then, COMS was conceived with the following philosophy. The computer scientist designs the basic programs of the system. Given these, the systems programmers use them to produce particular COMS implementations (that is, particular to a certain field, say compiler development or the fluid sciences). Library application programs for each implementation are supplied by sophisticated members of the user population who would be rewarded for their work (much like authors getting monetary awards for their efforts). The result should be a system which will rapidly grow and develop.

---

[1]Some of these software systems include SKETCHPAD (Sutherland, 1963), Map (Kaplow, 1966), ANAL 68 (Welsh, 1967), ICES (Roos, 1967) and SIR AND STUDENT (Raphael and Bobrow, 1964).

1.3. Elements of COMS

In the following four sections a description of the elements of COMS will be presented. These include the interpreter, the evaluator, the associative memory and the program library. Each element will be described in detail in terms of its purpose, its use and its relationship to the other elements.

1.3.1. The Interpreter

Of the four elements in the Communication Management System the interpreter serves as the most important. The reason for this is its ability to provide a control mechanism for each of the other elements.

Control is established through the manipulation of data (presented as strings of characters) by a set of rules (also presented as strings of characters) fed to the interpreter. These rules directing the interpreter may cause character strings to be passed to and received from the associative memory and the evaluator (see Figure 1.1 in section 2.1). Also, user communication is established by rules which may cause input of character strings (from the user) and output of character strings (to the user). Since the interpreter has the ability to communicate with both the other elements of COMS and the COMS user, it serves as an intermediary and translator between the two. This leads to the primary function of the COMS interpreter which is to allow the definition of command languages by experienced computer users, thus providing a valuable tool for access to computational facilities not previously available to inexperienced computer users.

Programming the rules which direct the interpreter is done in the string transformation language (STRAN). STRAN may be classified as a general purpose symbol manipulation language and as will be seen forms a very significant part of COMS. STRAN closely resembles the information retrieval language COMIT[1]. Both STRAN and COMIT use sequentially interpreted rules which perform decomposition, transformation and recomposition of character strings. Also both languages have rules which pass control to one of two other rules depending upon the success or failure of the decomposition portion of the rule. The main difference between STRAN and COMIT is that STRAN makes no distinction between rules and data. A character string to be interpreted as a rule must simply conform to certain rules of format if successful interpretation is to occur.

1.3.1.1. The STRAN Language

The interpreter for the string transformation language is a sequentially executed character string processor. Input to the interpreter must be in character string form so that STRAN programs (also referred to as "rules") and data are in the same format. An example of a few simple STRAN rules will serve to motivate the descriptive material that follows. These are shown in Figure 1.2 (page 13).

The interpreter can operate in two different modes: (1) In the "rule reading" mode rules are collected from the input cards (80 columns) and stored. A unique name is associated with each rule. The interpreter always begins execution in the rule reading mode but is switched from this to the "interpretive" mode when a rule name surrounded by parentheses is encountered. An example of this is

(PROG) or

(READ) or

(TEST)

Each of the above rule names are termed "go-to" rule names. When an input such as this is encountered, control is transferred to the rule named by the character string within the parenthesis except where the rule named is a pseudo operator.

1.3.1.2. STRAN Pseudo Operators

In the rule reading mode, the interpreter is able to accept commands from the STRAN user via a list of pseudo operators. (These may be compared to the pseudo operators found in a typical assembly language). Through these commands a user can control certain "switches" (i.e. variables set to ·true· or ·false· in the FORTRAN sense) which in turn control certain aspects of the interpretation operation. Input of a STRAN command does not change the rule reading mode to interpretive mode. This only occurs when a "go-to" rule name is encountered.

The following is a list of pseudo operators available in the present STRAN version:

(ECHO): Causes an echo of each input card to be printed. Each line given by the echo command begins with the phrase INPUT... to distinguish it from the output lines of the interpreter.

(NOECHO): Echoing is discontinued.

(TRACE): Causes each rule currently being interpreted to be output in the form:
INTERPRETING RULE...
and also causes the contents of each variable changed during rule execution to be output in the form:
VARIABLE (name) =

(NOTRACE): Turns off the (TRACE) command.

(PUNCH): Causes punching of a card for each output line produced by a rule. That is, the output line produced by the (TRACE) command is also punched on cards (without the two words shown above)

(NOPUNCH): Punching is discontinued.

(RESTART)
(RETURN)
(DUMP)
(READLX)

— See Appendix B

The above four pseudo operators pertain to the entire COMS system rather than the interpreter and are therefore discussed in the COMS reference manual.

In the interpretive mode execution of the STRAN program takes place. Once the interpretive mode has been entered a return to the rule reading mode can only be accomplished when the name of the next rule to be interpreted is (END) or when the pushdown stack (to be discussed below) is empty.

1.3.1.3. The STRAN Rule

Each STRAN rule is scanned from left to right by the interpreter. There are three main types of STRAN rules, all of which must begin with a left parenthesis followed by a rule name followed immediately by a second left parenthesis. Also, each of these rules must end with two right parentheses surrounding from zero to two rule names. A rule name may be from one to ten characters in length (any characters over 10 are ignored).

Using traditional Backus-Naur form the definition of a STRAN rule is given below. The English words enclosed between the angular brackets < and > denote syntactic constructs. Possible repetition of a construct is indicated by an asterisk (0 or more repetitions) or a circled plus sign (1 or more repetitions). If a sequence of constructs to be repeated consists of more than one element, it is enclosed in the meta-brackets { and }.

Thus in BNF we have

```
<STRAN RULE>::=(<RULE NAME>(<TYPE 1 BODY>|<TYPE 2 BODY>|
               <TYPE 3 BODY>)
 <RULE NAME>::=<NAME>
      <NAME>::=<LETTER><ALPHANUM CH>*
               (maximum of ten characters)
<ALPHANUM CH>::=<DIGIT>|<LETTER>|<ZERO>
     <DIGIT>::=1|2|3|4|5|6|7|8|9
    <LETTER>::=A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|
               W|X|Y|Z
      <ZERO>::=0
```

The user must fit each STRAN rule onto an 80 column card. If a rule is more than 80 characters in length, it must be broken down into smaller length rules so that each can fit on a card (this is easy to do with STRAN).

The three types of STRAN rules are described below with references to Figure 1.2. Rules with type 1 and type 2 bodies are used for storing information while those with type 3 bodies stipulate the processing to be performed by the STRAN interpreter.

Type 1 Body:

The form of this body in BNF is

<TYPE 1 BODY>::=<RULE NAME>{,<RULE NAME>}*)

Thus a rule with a type 1 body must contain a string of rule names separated by commas. An example is (1) of Figure 1.2.

The purpose of this type of rule is to place a list of rule names onto a push-down stack which is referenced by the interpreter when the rule name (END) is encountered. The stack is accessed in a "last-in-first-out" manner (from right to left) so that the last rule to be placed on the stack is the first one to be referenced by the interpreter. In the example referred to above, the rule will give rise to the following stack configuration

READ	→ First rule referenced
EXEC	→ Second rule referenced
TEST	→ Third rule referenced
⋮	⋮

Thus on encountering an (END) rule name, the interpreter will "pop-up" the first rule name in the stack and execute that rule. All the other rule names would then move up one

FIGURE 1.2

SIMPLE STRAN RULES

```
(PROG(READ,EXEC,TEST))                                      (1)

(STOP('FIN'HALT'TERMINATE'))                                (2)

(READ(*INPUT/'*'+$/=*INPUT/'COM...'+2/)READ,END)            (3)

(EXEC(INPUT/$+'('+$+')'+$/=*OUT/3/INPUT/5/)EXEC,END)        (4)

(TEST(INPUT/$+STOP/)END,PROG)                               (5)

(PUNCH)                                                     (6)

(PROG)                                                      (7)
```

slot giving:

Stack	
EXEC	→ First rule referenced
TEST	→ Second rule referenced
⋮	⋮

If now another stran rule of this type is executed by the interpreter, the existing rule names on the stack will be pushed down. For example

(RULE 1(XYZ,RG,SUCCEED))

will result in:

Stack	
XYZ	→ First rule referenced
RG	→ Second rule referenced
SUCCEED	→ Third rule referenced
EXEC	→ Fourth rule referenced
TEST	→ Fifth rule referenced
⋮	⋮

The maximum number of rule names the stack is capable of holding is 100.

<u>Type 2 Body:</u>

The form of this body in BNF is

<TYPE 2 BODY>::=<LITERAL PATTERN>)

<LITERAL PATTERN>::={'<CHARACTER STRING>}⊕'

<CHARACTER STRING>::=<any sequence of one or more basic 6-bit display BCD characters>. An example is (2) of

Figure 1.2.

The purpose of this rule is to store away a literal collection pattern under one name (which is the rule name) for later pattern matching in STRAN rules. This type of STRAN rule can be placed anywhere in the STRAN program and is dealt with by the interpreter while in the rule-reading mode only. Trying to pass control to this rule during execution will cause an interpretation error. An example of the use of this type of rule will be given later on in the discussion.

<u>Type 3 Body</u>:

The form of this body in BNF is:

```
   <TYPE 3 BODY>::=<RULE BODY>)<GO-TO SECTION>
  <GO-TO SECTION>::=<RULE NAME>|<RULE NAME>,<RULE NAME>
     <RULE BODY>::=<LHS>=<RHS>|<LHS>|=<RHS>
           <LHS>::={<VARIABLE NAME>/<DECOMP PAT>/|
                   +F<DIGIT>/<FIND PAT>/|
                   +A<DIGIT>/<ACCESS PAT>/}⊕
  <VARIABLE NAME>::=<NAME>
    <DECOMP PAT>::=<DECOMP OP>{+<DECOMP OP>}⊕
     <DECOMP OP>::=$|$<DIGIT>|$LITERAL|<VARIABLE NAME>|
                   ↓<DIGIT>|<LITERAL>|•<DIGIT>
       <LITERAL>::='<CHARACTER STRING>'
      <FIND PAT>::=<FIND OP>{+<FIND OP>}⊕
                   (a maximum of 4 find operators is al-
```

lowed in one find pattern)

<FIND OP>::=$|<LITERAL>|<DIGIT>

<ACCESS PAT>::=<DIGIT>{+<DIGIT>}$^{\oplus}$

(a maximum of 3 digits is allowed in an access pattern)

<RHS>::={{*|,}*<VARIABLE NAME>/<COMP PAT>/|
+S/<STORE PAT>/}$^{\oplus}$

<COMP PAT>::=<COMP OP>{+<COMP OP>}$^{\oplus}$

<COMP OP>::=<DIGIT>|<LITERAL>|↓<DIGIT>|≡L<NUMB>,<DIGIT>|
≡R<NUMB>,<DIGIT>

<NUMB>::=<DIGIT>|<DIGIT><DIGIT>

(numb has a maximum value of 80)

<STORE PAT>::=<STORE OP>{+<STORE OP>}$^{\oplus}$

(maximum of 4 store operators in a store pattern)

<STORE OP>::=<LITERAL>|<DIGIT>

The complete form of the type 3 body is presented above although many of the definitions (those dealing with the associative memory) will not be referred to until section 1.3.3.2. Some examples of type 3 body rules are shown in (3), (4) and (5) of Figure 1.2.

A STRAN rule with type 3 body is the real workhorse of the STRAN language, performing compositions, decompositions and transformations of character strings. The go-to section of the rule contains either one or two rule names.

If two are given they are separated by a comma. The first will receive control if the character string decomposition by the rule body succeeds. The second will receive control if decomposition fails. Of course if only one rule name is present, control is passed to it in either case.

Some examples are:

(A) (RULE1(<RULE BODY>)RULE2)

(B) (RULE2(<RULE BODY>)RULE3,RULE1)

(C) (RULE3(<RULE BODY>)END)

In (A) RULE1 will pass control to RULE2 no matter what happens in decomposition. However, in (B), RULE2 will only pass control to RULE1 on a decomposition failure whereas RULE3 will receive control on decomposition success. In (C) control will be passed to the rule named END regardless of whether RULE3 has failed or succeeded and, as mentioned previously, this will initiate the popping up of a rule name from the push down stack.

The rule body is separated into two sides (left and right) by an equals sign. The left side of the rule body performs three operations.

1) References the contents of STRAN "storage locations" named. A STRAN storage location provides storage for variable length strings of characters (up to a maximum

of 80 characters in one location) where both data and rules are kept. Each location of this storage is referenced by a unique name assigned at execution time.

2) Decomposes the contents of the storage locations named according to the pattern matching directions given.

3) Stores the resulting decomposed elements in one of nine successive special character-string-storage locations. These may be thought of as pseudo-registers or accumulators, each capable of holding up to 80 characters. Referencing of these registers is done by using the integers 1 to 9 as will be shown. (A virtual depiction of STRAN storage can be found in Figure 1.3).

The right side of the rule body performs three operations:

1) Concatenates the contents of specified pseudo registers and literals.

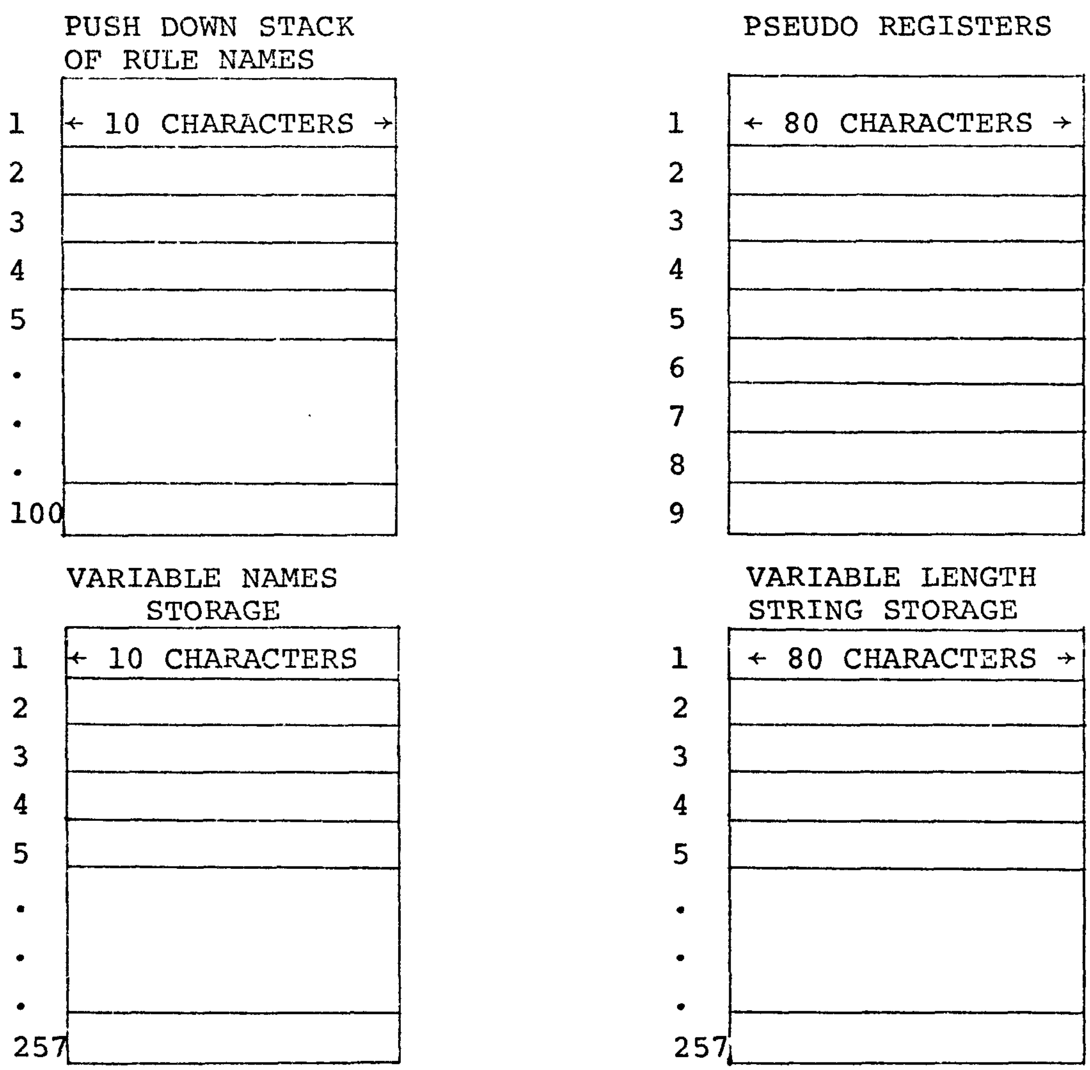
2) Stores the results (character strings) of concatenations in the storage locations named.

3) Associative memory operations (section 1.3.4).

A more detailed description of the rule body can be carried out using the example shown in Figure 1.4(a). Beginning with the left side of the rule body (Figure 1.4(b)) VARB is the name of a storage location (as with rule names the maximum length of a storage location name is 10 characters).

FIGURE 1.3

STRAN Storage



Notes:

1) Pseudo registers are automatically referenced in succession by the left hand side of a STRAN rule with type 3 body.

2) Variable names may also be used as rules names as long as the string referenced by that name is a syntactically correct STRAN rule.

FIGURE 1.4(a)

RULE BODIES

```
     (5)     (6)       (5)     (7)
(EG(VARB/$+'A'+$/=VARB/1+'B'3/)EG,R2)
(1)         (2)            (3)          (4)
```

(1) rule name

(2) left side

(3) right side

(4) succeeding rule names

(5) variable name

(6) decomposition operators

(7) composition operators

FIGURE 1.4(b)

RULE BODIES

```
                 (3)
          ┌─────────────┐
            (4)  (4)
VARB / $ + ' A ' + $ /
└──┘      ↑────────↑
(1)          (2)
```

(1) variable containing character string to be decomposed

(2) decomposition operators separated by plus signs

(3) decomposition operations to be performed on the variable VARB

(4) quotes indicate a literal

FIGURE 1.4(c)

```
                 (3)
          ┌─────────────┐
            (4)  (4)
VARB / 1 + ' B ' + 3 /
└──┘   ↑─────────────↑
(1)          (2)
```

(1) variable under whose name composed string will be placed in STRAN storage

(2) numbers refer to particular pseudo registers

(3) composition operations to be performed on decomposed string

(4) quotes indicate a literal

Operations enclosed between the two oblique strokes are performed on the variable named immediately to the left of the first oblique stroke. The operators for decomposition are seperated by plus signs. The two operators shown are the dollar sign $ and a character string literal 'A'.

A complete list of STRAN decomposition and composition operators along with their meaning is given in Appendix B.

The dollar sign operator matches any arbitrary character string including the null string. The character string literal (string of characters surrounded by single quotes) matches only an exact occurrence of the contained string of characters. Thus the above operators attempt to find an A in the character string stored in VARB. If an A is found (i.e. decomposition is successful) all characters preceding the left most A are placed in pseudo register 1, the A is placed in pseudo register 2 and the remaining characters in pseudo register 3.

For example if VARB references the string

I_b_RUN_b_WITH_b_SAM

(where _b_ indicates a blank character), the result of the above decomposition will be

pseudo register 1 contains I_b_RUN_b_WITH_b_S

pseudo register 2 contains A

pseudo register 3 contains M

If VARB references the string

IbRUNbWITHbPETER

decomposition fails, the rest of the rule body is skipped and control is immediately transferred to rule R2.

Now consider the right side of the rule body as shown in Figure 1.4(c). The storage location named immediately to the left of the first oblique stroke receives the result of the character string composed by the operators between the oblique strokes. Composition operators like decomposition operators are separated by plus signs. The integers 1 and 3 refer to the contents of pseudo-registers 1 and 3 respectively.

The contents of VARB after the above operations are performed will be the contents of pseudo register 1 concatenated with a B, concatenated with the contents of pseudo-register 3. Using the previous example VARB will contain

```
                (3)
                 ↓
IbRUNbWITHbSBM
                ↑
       (1)     (2)
```

(1) pseudo register 1

(2) literal

(3) pseudo register (2)

Control will now be passed to the rule named EG and the whole process will be repeated resulting in all the A's being changed into B's.

An asterisk placed in front of a variable name indicates that an input-output operation is to be carried out. Thus on the left side of a rule body an asterisk pre-ceeding a variable name tells the interpreter that before decomposition begins, an input line (i.e. an 80 column card) is to be read into a STRAN storage location for future reference by that variable. Similarly an asterisk preceding a name on the right side of a rule body tells the interpreter that after results have been placed in the storage location named, its contents are to be printed as a line of output. An example of this is:

```
(READ(*INPUT/'C'+$/=*INPUT/2/)READ,END)
```

This rule reads an input line, tests to see if that line begins with a C and if so it outputs the rest of the line. This process continues until an input line without a C at its start is found.

The comma is also used as a variable name prefix but only in the right side of the rule body. Its presence indicates that before the results of composition are placed in the storage location named, they are to be passed to the COMS evaluator (section 1.3.2). The result returned by the evaluator (always a string of characters) is then placed into this storage location. The use of the comma in this way is the only means by which the STRAN interpreter may be caused to communicate with the COMS evaluator. An example of the use of a comma is:

```
(EVAL(INPUT/$/=,RESULT/1/)END)
```

This rule causes the whole contents of the storage location named INPUT to be passed to the evaluator before it is placed in the storage location RESULT.

A comma may be used in conjunction with an asterisk to send a string to the evaluator and output the result when it returns. The order is not significant, either *, or ,* will work. For example:

If the variable INPUT contains the string I=COS(0) then the rule

```
(EVAL(INPUT/$/=,*RESULT/1/)END)
```

will print out

```
I=1
```

with the string I=1 stored in RESULT.

A rule body need not have both a left and right side. The equals sign is included only when it is necessary to mark the beginning of the right side. An example of a rule body with left side only is:

```
(READ(*INPUT/$+'CAT'+$+'bb'+$/)R2)
```

If a rule body has no left side, the operations specified are performed on the contents of the pseudo registers left from interpretation of previous rules. Examples of a rule body with right side only are:

```
(RIGHT(=OUT/1+'b'+5+6/)SUCCEED,FAIL)
(REMARK(=*LINE/'PROGRAMbbSTRAN'/)END)
```

The second example will cause the literal character string PROGRAMbbSTRAN to be printed out as well as stored in LINE.

It is possible to mention more than one variable name on either side of a rule body. For the left side of a rule body the operators in the seperate strings place their respective components of the decomposition in consecutive pseudo registers as shown by the following:

```
             1 2 3       4 5 6
             ↓ ↓ ↓       ↓ ↓ ↓
(DECOMP(VARB/$+'A'+$/TEMP/$+'B'+$/)END)
```

If VARB contains an A and TEMP contains a B, the A will appear in pseudo register 2 and B in pseudo register 5 after decomposition. A failure at any point causes the rest of the rule to be skipped.

An example of more than one variable on the right side is:

```
(COMP(=VARB/1+'ABC'+5/TEMP/1+2+3+4/)END)
```

The "literal" decomposition operator (number (2) as listed in Appendix B) is worthy of special note as its creation is dependent on a STRAN rule with type 2 body (section 1.3.1.3). Pattern matching takes place in the left side of a type 3 body rule using the name of the type 2 body rule as the decomposition operator.

For example say we had previously issued the following type 2 body rule:

(ANIMALS('bDOGb'CATb'bHORSEb'))

The string ANIMALS used as a decomposition operator would match the first occurrence of any of the three above literals (i.e. bDOGb or bCATb or bHORSEb). Assume the variable SENTEN contains the string

WALKbYOURbDOGbFIRSTbTHENbYOURbCAT

Now the following decomposition operation is possible:

(LIT(SENTEN/$+ANIMALS+$/))

Thus the following would be the pseudo registers' contents:

pseudo register	contents
1	WALKbYOUR
2	bDOGb
3	FIRSTbTHENbYOURbCAT

i.e. the leftmost occurrence of one of the literals in the collection will be matched.

A more useful example of this type of decomposition operator can be seen by having a STRAN rule such as

(PREPOSITS('bINb'bONb'bTOb'bBYb'bFORb'))

where all the prepositions in a sentence could be matched and printed out by a rule such as

```
(MATCH(SENTENCE/$+PREPOSITS+$/=*OUT/2/SENTENCE/3/)MATCH,END)
```

1.3.1.4. STRAN and the Associative Memory

The Associative Memory (section 1.3.4) forms a major part of the COMS system as does the STRAN interpreter, the evaluator and the program library. Interpreter communication with the evaluator has already been mentioned and now associative memory communication will be discussed.

The associative memory stores and retrieves ordered pairs, triples and quadruples of character strings. These are refered to as n-tuples for simplicity. Some examples of n-tuples are:

```
    (ANIMAL,CAT)
     (NUMBER,20)
      (A,B,C,D)
(NUMERICbFOR,20,TWENTY)
```

Storing of n-tuples in the associative memory occurs only on the right side of a rule body by using a STORE request. The components of an n-tuple may be literals or the contents of pseudo registers. A typical STORE request is written as

+S/pseudo register numbers and literals/

Plus signs are used as separators for the different parts of the n-tuple. For example

+S/1+'ONE'+5/

stores an ordered triple whose first element is the character string in pseudo register 1, whose second element is the literal 'ONE' and whose third element is the character string in pseudo register 5. It is interesting to note that we have already used the associative memory without knowing it in a previous example (section 1.3.1.3). The type 2 body STRAN rule (literal collection patterns) uses the associative memory automatically to store its literals. Taking for example the rule

(PREPOSITS('bINb'bONb'bTOb'bBYb'bFORb'))

this automatically generates stores of the following form:

+S/'PREPOSITS'+'bONb'/

+S/'PREPOSITS'+'bONb'/ and so on.

When the rule name PREPOSITS was used as a left side decomposition operator what in effect was happening was an automatic reference to the associative memory for retrieval of one of the above ordered pairs. That is, the above would have been stored as

(PREPOSITS,IN)

(PREPOSITS,ON)

.

.

.

etc.

Failure of a STORE request occurs only if the n-tuple being stored is already in the associative memory. If this occurs, execution of the rest of the rule is skipped and control is transferred as if the rule had succeeded.

Retrieval of stored character strings from the associative memory occurs on the left side of a rule body. It is carried out by two separate requests, FIND and ACCESS. The FIND request attempts to find suitable stored n-tuples to serve as answers for the n-tuple sought. If all the components of that sought n-tuple are known, then the only information that can be given the user is whether or not the n-tuple is stored. However, if some components of the n-tuple are not known, then they can be obtained by the ACCESS request. Before proceeding consider the following example.

Assume the following triples have previously been stored:

(LETTERS IN,3,BOY)

(NUMERIC FOR,2,TWO)

(NUMERIC FOR,10,TEN)

(NUMERIC FOR,13,THIRTEEN)

(LETTERS IN,5,TRAIN)

Now if a FIND request is issued for the triple (NUMERIC FOR, 10,TEN) the only result will be successful request. Conversely (NUMERIC FOR,5,FIVE) will result in an unsuccessful request. However, (NUMERIC FOR, ,TWO) will cause a search of the associative memory for a triple having as its first element the character string 'NUMERIC FOR' and as its last element the character string 'TWO'. To obtain the second component of this n-tuple an ACCESS request is issued. Thus the character string '2' will be picked up.

Now, looking at the actual STRAN commands (BNF on page A-1), the FIND request is written as:

+Fn/pseudo register numbers, literals and dollar signs/

As with decomposition and composition operators plus signs are used as separators for the above components. Using the previous example, assume that pseudo register 2 contains the character string THIRTEEN. Thus, the FIND request will be

+F4/'NUMERIC<u>b</u>FOR'+$+2/

The above will retrieve all ordered triples whose first element is the string NUMERIC<u>b</u>FOR , second element is an arbitrary string and third element is the string contained in pseudo register 2. The number immediately following the +F is used by the ACCESS operation to identify which FIND request will receive the result. This identification is necessary due to the possibility of several ACCESS and FIND operations occurring in one program. Each ACCESS operation must know which FIND requested it.

The ACCESS request is written as

+An/pseudo register numbers/

(Separators of pseudo register numbers are again plus signs).

The number of pseudo registers needed depends entirely on how many dollar signs occur in the associated FIND request. Continuing with the present example the ACCESS request is:

+A4/4/

since only one dollar sign appears in the FIND operation. The end result of the example is that the character string '13' will be placed in pseudo register 4.

The ACCESS request is thus used to collect results of FIND requests which have included dollar signs. If no dollar signs are included then the success or failure of that FIND request is the only useful information obtained and any ACCESS request associated with the FIND will not return any valid information. Failure of a FIND request causes control to be passed in the exact same manner as a decomposition failure (to the rule whose name has been given for the failure case). Further examples of STRAN associative memory requests can be found in program (4) listed in Appendix C.

Summary of STRAN rule syntax

The following is a summary of the syntax of rule bodies (to be used for quick reference). Curly brackets indicate a choice of one or more from a list and square brackets indicate an optional element.

(1) left and right sides of a rule body are separated by an equals sign.

(2) left side: one or more syntactic units from the following-

  (a) [*] storage name/decomposition operator string/

  (b) +Fn/string of $'s, literals and pseudo register numbers/

  (c) +An/string of pseudo register numbers/

(3) right side: one or more syntactic units from the following-

  (a) [{*,}] storage name/composition operator string/

  (b) +S/string of literals and pseudo register numbers/

(4) All strings of operators between slanted bars are seperated by plus signs.

1.3.1.5. STRAN Errors

The STRAN user must take care that he never causes the interpreter to process a string of characters that is not a well formed rule. This mistake can easily be made due to the fact that STRAN rules and STRAN data are stored in the same mechanism. If an error such as this does occur the message "Error has occurred in interpretation of" followed by the rule name is printed. The interpreter then automatically pops up another rule name from the push down stack and begins executing it. If the stack has no more rule names in it the interpreter switches to the rule reading mode and proceeds to read the next input card.

The problem here is that in most cases all rules will have previously been read in, so the result is either data cards are read in as rules (which leads to another interpretation error unless the data is itself a well formed rule) or there are no input cards left, thus causing the termination of execution entirely.

If a user tries to perform a decomposition operation on a variable which has nothing stored in it then the error message "Variable named (variable name) is not yet stored" appears and the same procedure described above is followed by the interpreter. This type of error commonly occurs due to a misspelling of previously used variable names.

The error message "Error in evaluation of algebraic expression" occurs when a string of characters sent by the interpreter to the evaluator has caused the evaluator to go into error. Another evaluator error is caused by using more than 5 subscripted variable names in one algebraic expression. The error message given is "Error, algebraic expression contains more than 5 subscripted variable names".

A variable in an arithmetic expression not yet assigned a value is assigned a default value of zero and the message "The variable (variable name) has been assigned a value of zero" is printed. Any errors due to storing or retrieving values of either subscripted or unsubscripted variables by the evaluator results in the error message "Error in numeric storage or retrieval". An overflow of evaluator storage results in "Numeric storage has overflowed". "Error in indices" is caused by incorrect referencing of array variables. Overflow of the storage of rule names and variable names (which are both stored in the same area) causes "Dictionary full, execution terminated". Finally, a STRAN program trying to read more data cards than actually exist results in the message "End of file read on input tape (tape number)". This cannot really be classified as an error message since the termination of a STRAN program (with no errors) is brought about by there being no more input cards left on the input file. At this point the above message is also written out and execution is stopped.

A list of all the above mentioned error messages along with the resulting action by the interpreter can be found in Appendix D.

1.3.1.6. Conclusions and Examples

STRAN can be described as a simple but powerful language that is easy to learn and easy to use. Take for example the fact that rules and data are stored under the same mechanism. This allows the user to change the meaning of a rule during its execution. The example below shows this feature of STRAN. The reader must keep in mind that when a STRAN rule is stored internally, the pair of left parentheses and the rule name they surround are removed from the rest of the rule and stored else where - i.e. when say (S1(=*OUT/1/)S2) is stored, only the characters =*OUT/1/)S2) are stored together. Consider the rule

(EXAMPLE(=RULE1/'RULE2,RULE3))'/RULE1)

This stores in the variable RULE1 the character string RULE2, RULE3)) and thus when this rule passes control to RULE1 (i.e. in the go-to section) what the interpreter sees stored under rule name RULE1 is a type 2 body rule telling it to place the rule name RULE2, RULE3 on the push down stack. Thus the rule EXAMPLE actually creates another STRAN rule called RULE1 and transfers control to it.

Now to extend the above example and make it more general examine the following section of a STRAN program:

PROGRAM

```
(XYZ(                              )FOUND,FAIL)
(RULE4(                            )FOUND,FAIL)
                    .
                    .
                    .
(READ(*INPUT/$+'('+$+')'+$/=OUT/3/)S1,END)
(S1(OUT/$/=OUT/1+'))'/OUT)
                    .
                    .
                    .
```

DATA FOR PROGRAM

```
(READ)
(XYZ)
(RULE4)
```

When the rule called READ is executed the character string READ is placed in the variable OUT and control transferred to rule S1. Rule S1 adds the two closing parentheses to the character string READ obtaining the character string READ)) which is again stored in the variable OUT. Now control is transferred to the name OUT and this results in the rule name READ being placed on the push down stack. Thus the next rule to be executed is READ (since it lies at the top of the stack) and the whole process repeats itself, reading in the character string (XYZ).

The above examples plus the ones given in Appendix C should demonstrate some of the more useful characteristics of STRAN.

One can view STRAN and its interpreter as a segment of the overall COMS system working to provide simple communication links with the other elements. This is an extremely important task since although STRAN serves as a powerful character manipulation language in its own right, the effectiveness of the whole COMS system is dependent entirely on communicable results between the user and the COMS elements.

1.3.2. The Evaluator

Any implementation of a COMS system must use the evaluator program to achieve the goal of effective utilization of the program library (1.3.3). Besides serving the purpose of a communication link to the program library, the evaluator also provides an interface mechanism for the interpreter to the realm of numeric data. More specifically, this element of COMS evaluates algebraic formulas, allocates space for numeric variables and arrays, stores values in and retrieves values from these variables and arrays, generates argument lists for the COMS Fortran library programs and causes the execution time loading of these same programs upon request from the COMS user.

Commands to direct evaluator action are received from the interpreter as character strings. Results from the evaluator are sent back to the interpreter also as character strings.

To the knowledgeable computer user desiring a modification of a particular COMS system, the evaluator is more resistant to change than any of the other COMS elements. The reason for this is found in the actual program set-up of the evaluator which deals with the practical and concrete difficulties of the Scope operating system. Also, because the evaluator deals specifically with library programs and sets of data, it is more dedicated to a specific problem area (say geophysics or astronomy) than either the interpreter or the associative memory (1.3.4).

1.3.2.1. Algebraic Formula Evaluation

As in Fortran, algebraic expressions are accepted by the evaluator in infix notation. There are basically two ways to cause evaluation of expressions. The first is by sending the evaluator a character string consisting of a STRAN variable name followed by an equals sign followed by the expression. In this case the character string returned to the interpreter includes the name of the target variable, the equals sign and the resulting value of the expression. For example if the string

```
RESULT=Y+5*(2+Z**2)
```

is sent to the evaluator, then the string

```
RESULT=33
```

will be returned (assuming Y and Z have previously been assigned the values 3 and 2 respectively). The second way to cause evaluation is by sending only the expression itself. For example sending the string

```
Y+5*(2+Z**2)
```

will return the string 33.

Most of the built-in functions and operators available in Fortran are also available to the evaluator. These include:

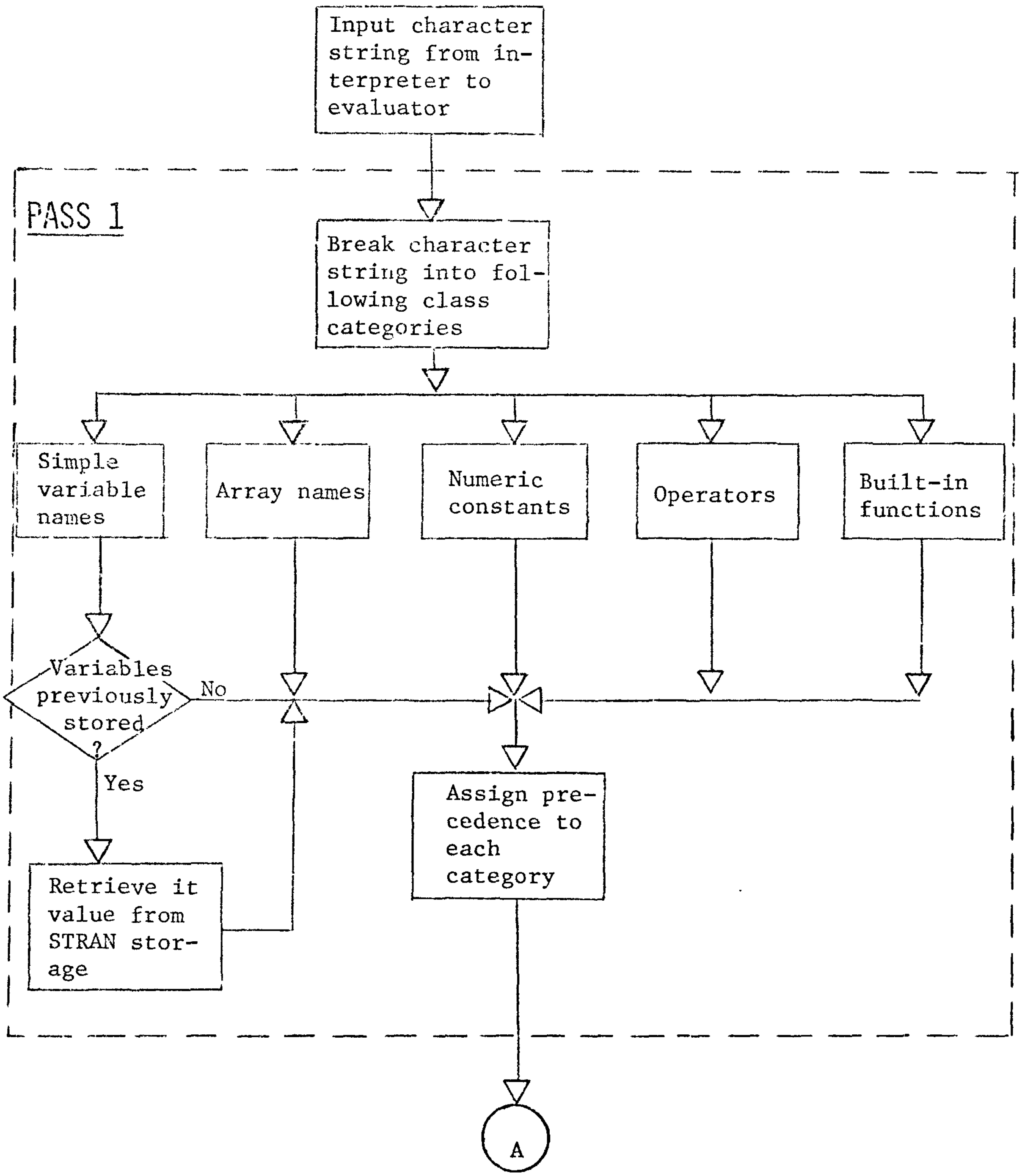
+, -, *, /, **, unary +, unary -, AMOD, MOD, FLOAT, FIX, ABS, IABS, SIN, COS, TAN, ATAN, EXP, ALOG, ALOG10, SQRT.

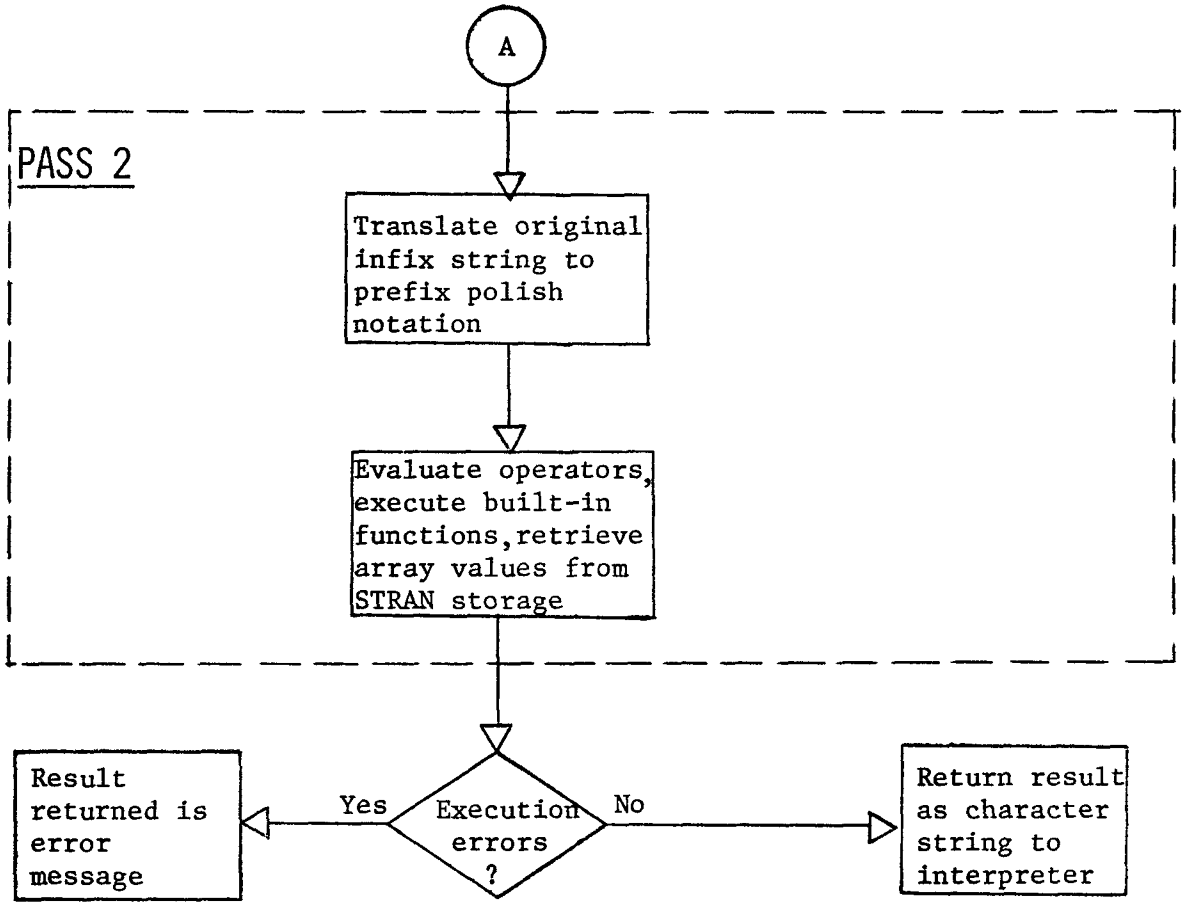
The mode (fixed or floating) of a particular STRAN variable is defined by the mode of the number assigned to it. For example Y=5 makes Y integer while Z=5.2 makes Z real. An expression containing mixed modes has all its fixed point numbers floated. Also if a fixed point number appears as the argument of a built-in function requiring a floating point argument, that number is floated and vice-versa.

The evaluator operates in two passes as shown in Figure 1.5. The first pass collects contiguous characters into symbols and interprets these symbols as variable names, numeric constants, array names, operators or built-in function names. Also, the values of simple variables are retrieved from STRAN storage and a precedence is assigned to each operand and built-in function. The second pass translates the original infix notation fed to the evaluator into prefix Polish notation. This is accomplished under the control of operator precedence and parentheses. Also during the second pass, evaluation of operators, execution of the built-in functions and retrieval of array values takes place.

FIGURE 1.5

EVALUATOR BLOCK DIAGRAM




A
PASS 2
Translate original infix string to prefix polish notation
Evaluate operators, execute built-in functions, retrieve array values from STRAN storage
Execution errors ?
Yes
No
Result returned is error message
Return result as character string to interpreter

1.3.2.2. Infix to Prefix Polish

The translation scheme used for the conversion of infix to prefix Polish notation is based on a publication of the Burroughs Corporation called a "Compilogram" which was specifically adapted for this use by D. McCracken [2]. A flowchart of the assignment of operator precedence and the translation process is shown in Figure 1.6. Although the evaluator has many more intricacies than are shown, the basic method of conversion is the same. Also, for simplicity, variable names have been assumed only one character in length and no error situations are examined.

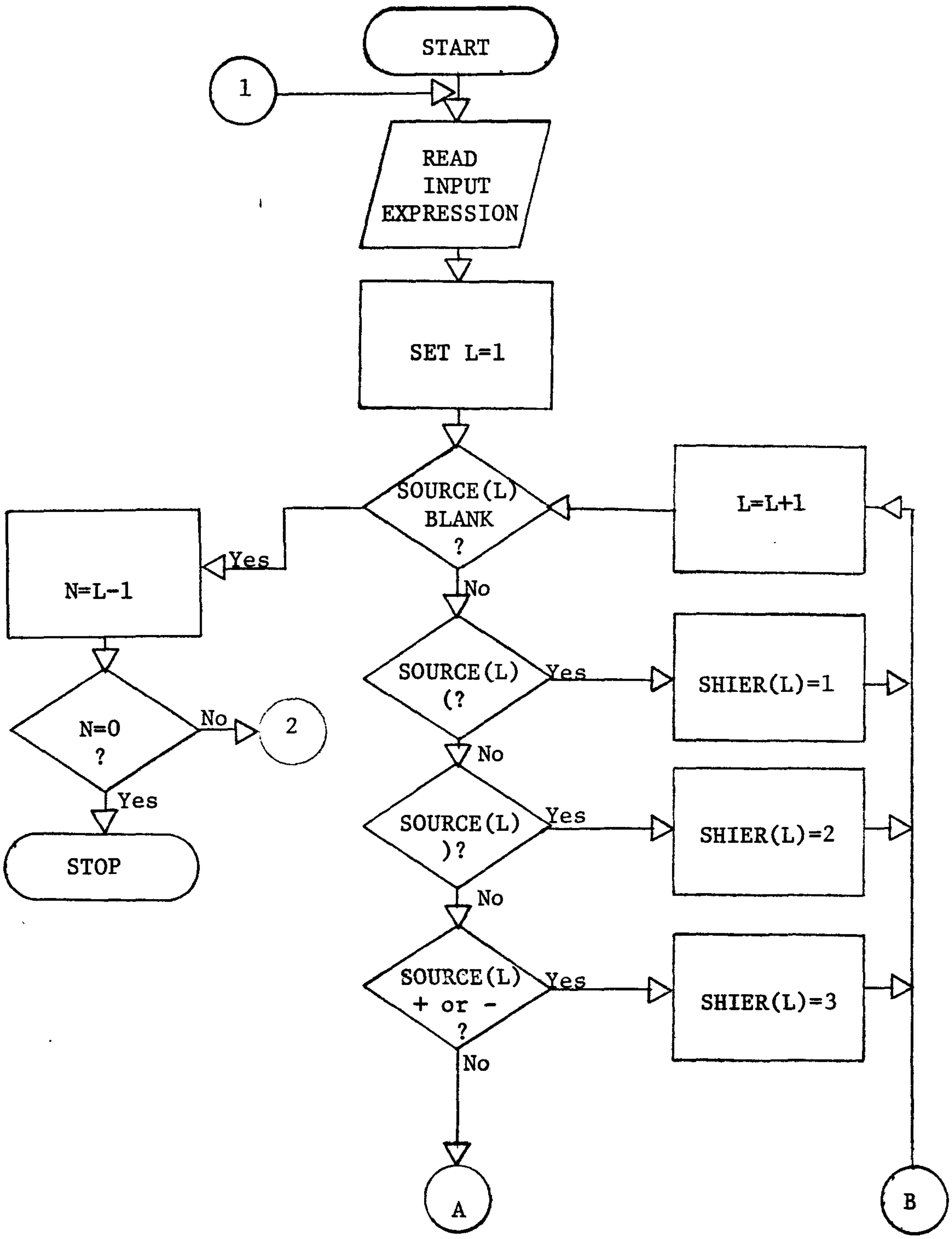
The input expression is stored in the array SOURCE and the associated precedence of each operator, operand or built-in function is stored in SHIER (standing for Source HIERarchy). Allocation of precedence (in the order lowest to highest) is as follows:

OPERAND	0
(	1
)	2
+,-	3
*,/	4
**,MOD	5
UNARY +, - BUILT IN FUNCTION }	6

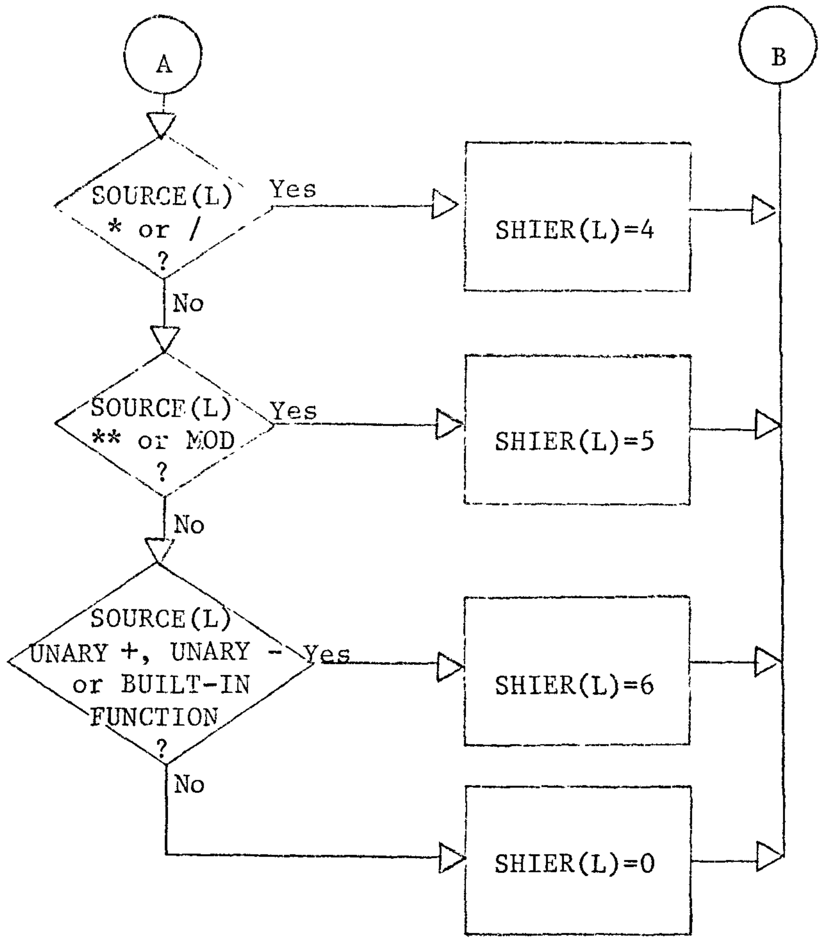
Operands are transferred to the array POLISH as soon

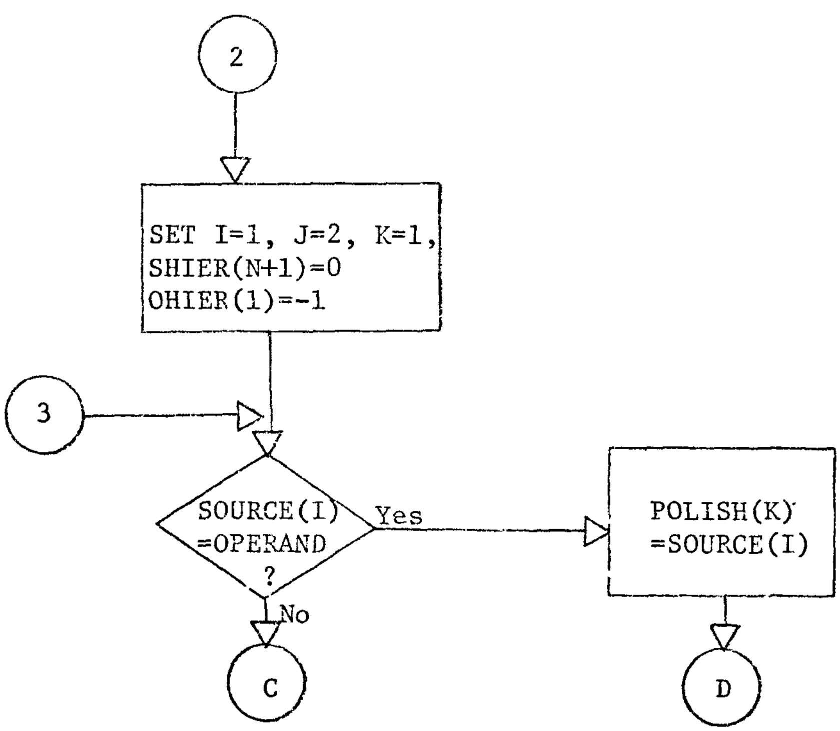
FIGURE 1.6

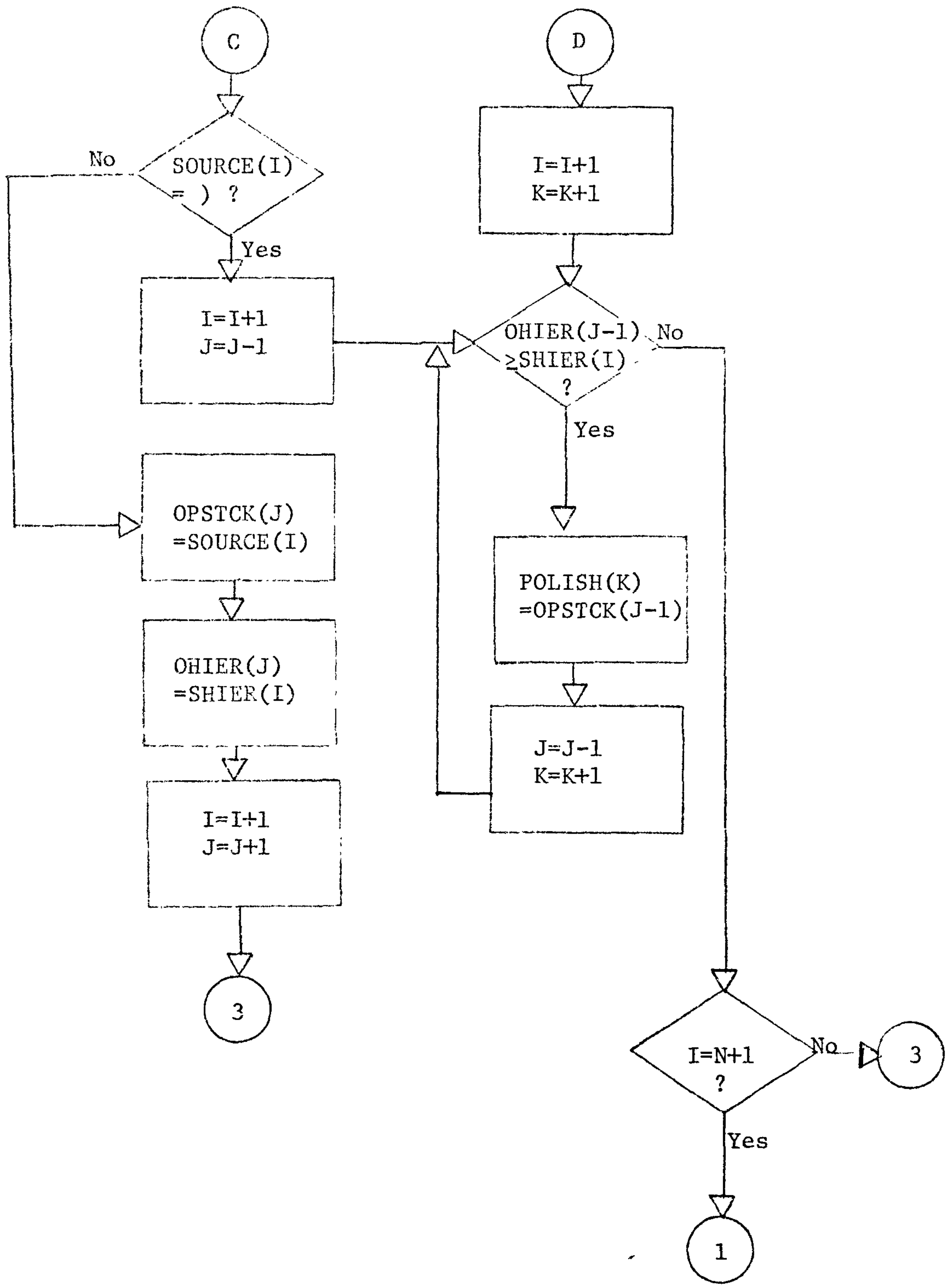
ASSIGNMENT OF OPERATOR PRECEDENCE AND TRANSLATION OF INFIX TO PREFIX POLISH NOTATION



Note: assume no blanks precede the input expression.


A
B
SOURCE(L) * or / ?
Yes
SHIER(L)=4
No
SOURCE(L) ** or MOD ?
Yes
SHIER(L)=5
No
SOURCE(L) UNARY +, UNARY - or BUILT-IN FUNCTION ?
Yes
SHIER(L)=6
No
SHIER(L)=0


2
SET I=1, J=2, K=1,
SHIER(N+1)=0
OHIER(1)=-1
3
SOURCE(I) =OPERAND ?
Yes
POLISH(K) =SOURCE(I)
No
C
D


C
D
SOURCE(I)
=) ?
No
Yes
I=I+1
K=K+1
I=I+1
J=J-1
OHIER(J-1)
≥SHIER(I)
?
No
Yes
OPSTCK(J)
=SOURCE(I)
POLISH(K)
=OPSTCK(J-1)
OHIER(J)
=SHIER(I)
J=J-1
K=K+1
I=I+1
J=J+1
3
I=N+1
?
No
3
Yes
1

Notes

(1) The built-in functions available to the evaluator are not in actual fact all given the same precedence as shown above. A separate section of coding hanles their individual classifications, however for simplicity this will not be shown.

(2) The exponentiation operator and the MOD function are both treated as binary infix operators.

(3) Also for simplicity, operands are taken as single letter unsubscripted variables.

(4) Blanks are ignored in the input expression.

as they are encountered. This array holds the resulting expression in prefix Polish form. Operators (except right parentheses) are held temporarily in the array OPSTCK (OPerator STack) and their associated precedence is stored in the array OHIER (Operator HIERarchy). When an operand is picked up from the input expression, it is transferred to the array POLISH immediately and a check is made to see if the last entry made in the operator array OPSTCK has the same or a higher precedence than the next operator in the input string. If so, the last entry operator is placed in POLISH. If not, the next character of the input string is examined and the process repeated.

If an operand is not encountered as the next element of the input string, a check for a right parenthesis is made. If one is found, it is ignored and its matching left parenthesis which will always be the last entry in the operator array, is also ignored.

If a right parenthesis does not turn up, an operator must have been encountered and thus it is transferred to the operator array with its precedence placed in the array OHIER. The next element of the string is then examined.

The whole process continues until the total number of characters in the input string (established previously when precedence was originally being assigned) has been examined.

A few examples are shown below:

INFIX INPUT	PREFIX OUTPUT
-(A**B)/C	AB**C/-
A*(B-C)+D	ABC-*D+
A+B/(-(D**E*F)/G)	ABDE**F*G/-/+

For a much more detailed and complete description of both the first and second passes of the evaluator, the reader is referred to the COMS reference manual (Appendix E).

1.3.2.3. Variables and Arrays

The evaluator dynamically allocates storage for variables and arrays. For the former this is done by the assignment of a value to the variable (an example is SPEED=50.2). For the latter, a statement of the following form must be passed to the evaluator.

INTEGER<NAME>(<DIM1>,<DIM2>,....)

REAL<NAME>(<DIM1>,<DIM2>,....)

As in Fortran, arrays are classified as integer or real depending on the mode of the data they are referencing. <NAME> refers to the name of the array being declared (as with STRAN variable names, array names may be up to 10 characters in length). Following the array name as many dimensions as desired are listed. These are separated by commas and enclosed completely by parentheses. A particular dimension may be stated as a constant, a variable or an expression. For example, the five dimensional real array TIMINGS may be declared as

```
REAL TIMINGS(K,6,2.5*SQRT(Z),7,(P+3)/2.8)
```

Retrieval of a value stored in a variable or array is caused by the appearance of the variable name or the array name with subscripts in an expression.

Although the evaluator serves well as an arithmetic processor for the COMS system, its more important task is communication with the program library (1.3.3). Examples of the use of the evaluator in different types of calculations are shown in program (1) of Appendix C.

1.3.2.4. Communication with the Program Library

The loading and execution of Fortran subprograms (subroutines and functions) from the COMS library is carried out by a special assembly language routine in the evaluator program called LOADIT. The original PL1 version of COMS made use of the IBM linkage editor program to perform the same jobs as LOADIT, but of course this routine is not available in CDC computer software systems (and neither is any reasonable facimile). It was therefore necessary to develop such a program specifically for the current Fortran version of COMS. Details of the set-up and operation of LOADIT can be found in Appendix F.

A library subprogram is loaded and executed when either of the following character strings are sent by the interpreter to the evaluator:

CALL <PROGNAME> [a]

CALL <PROGNAME>(<ARG1>,<ARG2>,....) [b]

In [a] and [b] <PROGNAME> refers to the name given the subprogram when it was originally placed in the COMS library (placing subprograms in the COMS library is discussed in section 1.3.3). Only a subroutine subprogram may be called with either no arguments (as in [a]) or with an argument list (as in [b]). Function subprograms must be called with an argument list. This list is enclosed by parentheses and each argument (up to thirty allowed) is separated from the next by a comma.

The only method of communication between a library subprogram being executed and the rest of the COMS system (i.e. through the interpreter) is via the argument list introduced above[1]. A primary restriction on all subprograms

---

[1]The original version of COMS has a second method of communication involving the passing of NAMELIST data input di-

placed in the library is that the passing of arguments by any method which depends on some previous linkage technique between the calling program and library routine is not allowed. This in fact means that arguments can in no way be passed through a common block shared with COMS. The reason for this major restriction involves the independence of both the library and COMS programs. To change the actual Fortran coding of COMS every time a new subprogram is to be placed in the library would lessen the efficiency of a system that was wholly designed to provide its own communication management. Also, the library subprogram has to maintain its independence if it is to be classified as a true "utility" routine.

The type of arguments allowed in a call to a library routine include nearly all those available in standard Fortran programming. This includes expressions to be evaluated, array names, specific array elements and simple numeric variables and constants.

---

rectly from the evaluator to NAMELIST read instructions located in the current library program being executed. Details of this NAMELIST interface mechanism (which is not currently available to the present version of COMS) can be found in the COMS thesis by GAMMILL [3].

Correspondence between actual and formal parameters[1] is carried out by reference. This means that at runtime, prior to the subroutine call, the actual parameters are processed. If they are not variables (simple and array variables) or constants, they are evaluated and stored in temporary locations assigned by the calling routine. The addresses of the variables, constants and temporary locations are then calculated and passed to the called subprogram. The subprogram uses these addresses to perform the desired calculations on the values referenced by them - thus COMS library programs have the ability to change the formal parameter values sent them by the evaluator. By use of the dollar sign character $ placed immediately in front of a simple or array variable name, it is possible to send the address of such a formal parameter to the called procedure as the actual value of the variable. For example if the simple variable A is located at machine address 000122 and has as its contents the real number 5.2, when a call such as

---

[1]Formal parameters are identifiers in a subroutine which are replaced by other identifiers or expressions when the subroutine is called. For example in the Fortran statement SUBROUTINE A(X,Y), X and Y are formal parameters. Actual parameters are those listed in the subroutine call. For example, in the Fortran statement CALL A(B,C*D), B and C*D are the actual parameters.

CALL XYZ ($A) is made, the address of a temporary location is sent to the subprogram having as its value the integer 000122. The reason this particular feature is available in the current version of COMS (which has no real use for it) is simply because it was left over from the previous IBM 360 Fortran version of COMS which used it to override the simple variable call-by-value correspondence inherent in this compiler. Examples of various possible actual parameters used in a calling statement are shown in Figure 1.7. The following section will discuss the placing of subroutines and functions in the program library and the organization of these for maximum programming effectiveness.

FIGURE 1.7

EXAMPLES OF POSSIBLE ACTUAL PARAMETERS, USED IN CALLING LIBRARY PROGRAMS

Note: assume the declaration REAL SSQ(50,50) and the values of I, J and Q have previously been passed to the evaluator.

(1) simple variable I

The address of a temporary location containing the present value of I is passed.

(2) array element SSQ(12,14)

The address of a temporary location containing the present value of SSQ(12,14) is passed.

(3) array name SSQ

The address of a temporary location containing the present value of the first element of SSQ is passed.

(4) array element SSQ(I,J)

The present value of I and J are used to calculate the address of a temporary location containing the present value of SSQ(I,J) which is then passed.

(5) expression SIN(I)+J+Q

The present value of I, J and Q are used to evaluate the expression which is then stored in a temporary location. The address of this location is then passed.

(6) <u>constant</u> 7.9E-5

The address of a temporary location containing this value is passed.

(7) <u>dollar sign operation</u>

```
$I
$SSQ(12,14)
$SSQ(I,J)
$SSQ
$SIN(I)*J+Q
$7.9E-5
```

The address of a temporary location containing the machine address of each of these parameters is passed.

1.3.3. Fortran Library of Programs

1.3.3.1. Placing Programs in the Library

As outlined in the preceding section, the evaluator, upon recognizing a user call to the program library, requests an assembly language program called LOADIT to perform this operation. The information that is automatically passed to LOADIT includes the called subprogram name, the number of arguments in the call, the location and value of these arguments and also whether the location of each argument contains its actual value or its address (this refers to the dollar sign operator as discussed in section 1.3.2).

In the present version of COMS, the library is stored on a file on disc. The name given this file is COMSLIB and it is here that LOADIT expects to find library programs. Each time the COMS program is run this file must be attached to the job (the full set of control cards needed to run a COMS job is shown in Appendix G).

Thus before COMS is used at all, the library routines needed (either now and/or for future COMS programs) must be placed on COMSLIB. The creation of COMSLIB using the CATALOG control card under Scope 3.4 is shown in Figure 1.8. Also shown are the control card set-ups for both placing additional programs on the file and purging the entire file when it is not in use (a basic knowledge of Scope control cards is assumed).

FIGURE 1.8

CONTROL CARD SET-UP FOR

(1) CREATION OF COMSLIB FILE

(2) ADDITIONS TO COMSLIB FILE

(3) DELETION OF COMSLIB FILE

(1) CREATION OF COMSLIB FILE

```
JOB CARD
REQUEST,LGO,*PF.
RUN(S)
CATALOG(LGO,COMSLIB,ID=COMSPROG,RP=30,XR=A)
END OF RECORD
[ SUBPROGRAM TO BE PLACED IN LIBRARY
END OF FILE
```

(2) ADDITIONS TO COMSLIB FILE

This is done in two stages to allow testing of the new file for any errors that might have occurred during the cataloging process. The user is also cautioned to closely check the program itself since any errors in the coding of a program already added to the file means the entire file will have to be purged and recreated to dispose of this program (there is no easy method by which the program in error can be deleted from the rest of the file).

Stage 1:

```
JOB CARD
REQUEST,LGO,*PF.
ATTACH(X,COMSLIB,ID=COMSPROG,RP=30,PW=A)
RUN(S)
COPYBF(X,LGO)
CATALOG(LGO,COMSLIB,ID=COMSPROG,RP=30,XR=A)
END OF RECORD
[ SUBPROGRAM TO BE ADDED TO LIBRARY
END OF FILE
```

After testing the newly created file and finding no errors, the old cycle may be purged.

Stage 2:

```
JOB CARD
ATTACH(X,COMSLIB,ID=COMSPROG,RP=30,PW=A)
PURGE(X,COMSLIB,ID=COMSPROG,LC=1,PW=A)
END OF FILE
```

(3) DELETION OF COMSLIB FILE

```
JOB CARD
ATTACH(X,COMSLIB,ID=COMSPROG,PW=A)
PURGE(X,COMSLIB,ID=COMSPROG)
END OF FILE
```

1.3.3.2. Efficient Program Organization

A systems programmer in setting up a COMS implementation may wish to place utility routines in the library which would be useful not only to a particular problem area but to all problem areas in general (i.e. the library would always have these routines placed in it no matter what COMS application was involved). Examples of these include input-output routines involving printing, punching, plotting and CRT displays, routines to generate, edit or correct large data decks, or routines to control the external storage of information (i.e. on tape, disc, cards, drum, etc).

To make these routines flexible and easily used by both experienced and inexperienced programmers, certain methods by which FORTRAN subprograms can collect directive information from outside themselves are discussed in this section. Use of these methods is not directly related to COMS, but application programs employing them will help to make COMS a more versatile system.

Actually, the methods involved may not only be used in general routines as such, but rather in any program where the collection of information may vary from minimal (that information absolutely essential for correct execution) to maximal (settings for all optional parameters). Thus, the beginning user will find much less mandatory information required and the advanced user will find the ability to exert much more control over the program.

SET-RESET METHOD

The SET-RESET method of program organization (discussed by Gammill [3]) involves the use of NAMELIST input to direct a program which is to be executed several successive times using various settings of control parameters for each execution. NAMELIST input-output is included in some FORTRAN compilers (the FORTRAN-RUN compiler on the CDC 6400 allows it) but is not a part of ANSI (American National Standards Institute) FORTRAN. It involves the use of an internal symbol dictionary to allow the unformatted input and output of specified variables and arrays.

In the SET-RESET method, NAMELIST I/O is specifically applied to the modification of default values of a large set of independent variables and control parameters since only a subset (including none) of NAMELIST variables need be read in at any particular time. To the inexperienced programmer this provides a mechanism for obtaining default results of a program with no input operations necessary, while the same program in the hands of an experienced user can provide access to all internal control parameters needed.

The SET-RESET method involves the inclusion of a logical control parameter among the other program variables to make it possible to reset any, all or none of the initial settings of these variables at the beginning of each execution of the program. A Fortran program using the SET-RESET method is shown in Figure 1.9. The data cards used will cause the program to be successively executed three times before stopping. The values of the NAMELIST input variables for each pass are as follows:

PASS	I	J	RESET
1	5	5782	•FALSE•
2	2400	5782	•TRUE•
3	1	1	•FALSE•

The point of the above mentioned example is that if the program variables are not reset, their initial values used for any execution are those left from the previous execution. This allows the user working on a particuar problem to make small changes in a few independent variables which results in a slow step by step progression through the problem. After all, a typical researcher doesn't make massive changes in input data to get to a solution, but rather changes a few things "here" and a few things "there" to see if the result holds more promise.

Using the SET-RESET method encourages programmers to write programs which are extremely data directed. Applications of the method hold for any program which has many modes of operation or some uncertainty as to the best set-

FIGURE 1.9

THE SET-RESET METHOD OF PROGRAM ORGANIZATION

```
C       PROGRAM SET-RESET
C       DECLARATIONS
        NAMELIST/INPUT/RESET,I,J/OUTPUT/K
        LOGICAL RESET
C       INITIALIZE NAMELIST VARIABLES TO DEFAULT VALUES
1       I=1
        J=1
        RESET=.FALSE.
C       RESET DETERMINES IF A NAMELIST READ IS TO BE DONE
2       IF (RESET) GO TO 1
C       READ NEXT NAMELIST INPUT STRING
        READ(5,INPUT)
C       TEST FOR END OF FILE CONDITION
        IF (EOF,5) 3,4
C       WRITE OUT NAMELIST INPUT STRING
4       WRITE(6,INPUT)
C       ANY CODE USING VALUES OF I&J IS INSERTED HERE
C       A POSSIBLE EXAMPLE IS THE FOLLOWING
        K=I*J
        WRITE(6,OUTPUT)
C       END OF INSERTED CODE
C       EXECUTE THE PROGRAM AGAIN
        GO TO 2
C       STOP THE PROGRAM
3       STOP
        END
```

DATA CARDS USED:

```
        $INPUT I=5, J=5782 $
        $INPUT I=2400, RESET=.T. $
        $INPUT  $
```

ting of various independent parameters. The user is allowed a "good guess" default value for different parameters until the "true" value can be obtained through calculation.

A convenient implementation of the method is as a separate subroutine that could be placed in the COMS library and linked to any program needing this type of organization. The following section describes two more methods of program organization which also adapt library programs for more versatile and flexible use under COMS control.

METHOD OF VARIABLE LENGTH ARGUMENT LISTS

The purpose of this method of program organization is to relieve the application program user of having to remember all the arguments a particular subprogram requires. For example, if a library subprogram requires the passage of sixteen arguments for its operation but some of these need be changed only under certain circumstances, it would be nice for the inexperienced user (under normal operation of the subprogram) to leave these alone and have the subprogram itself take care of them. This possibility exists in the subprogram shown in Figure 1.10. In this example, it is only necessary to pass three arguments to the subroutine for its proper execution. All the rest of the arguments are optional to the programmer and are automatically initialized inside the subroutine.

FIGURE 1.10

VARIABLE LENGTH ARGUMENT LISTS

```
      SUBROUTINE EG(ARRAY,ISIZE,JSIZE,OPARG1,OPARG2,
     *OPARG3,....)
C     OPARG REPRESENTS OPTIONAL ARGUMENT
      DIMENSION ARRAY(ISIZE,JSIZE)
C     INITIALIZE DEFAULT VALUES OF OPTIONAL
C     ARGUMENTS BY EITHER ASSIGNMENT OR
C     DATA STATEMENT
      DATA ARG2,ARG3/0.0,5.0/
      ARG1=10.0
        .
        .
C     ANY MORE INITIALIZATIONS TAKE PLACE HERE
        .
        .
C     FIND OUT HOW MANY ARGUMENTS THIS SUBROUTINE WAS CALLED
C     WITH
      CALL NUMP(NARG)
C     IN THIS PARTICULAR SUBROUTINE THREE ARGUMENTS ARE
C     MANDATORY
      IF (NARG.GT.2) GO TO 500
C     WRITE(6,100)
100   FORMAT('TOO FEW ARGUMENTS IN CALL TO EG')
      RETURN
500   NARG=NARG-2
C     RESET VALUES OF SPECIFIED ARGUMENTS
      GO TO (1,2,3,....) NARG
        .
        .
C     ONE WAY OF RESETTING IS AS FOLLOWS
        .
4     IF (OPARG3.EQ.0) GO TO 3
      ARG3=OPARG3
3     ARG2=OPARG2
2     ARG1=OPARG1
1     CONTINUE
        .
C     BODY OF SUBROUTINE HERE
        .
      END
```

Note: NUMP is an assembly language routine which counts the number of arguments in the call to the routine in which it is placed. This count is then stored in NARG.

Two methods of initialization are shown. One is by the DATA statement (for ARG2 and ARG3) and the other by an assignment statement (for ARG1). In the former case an argument if changed by a specification in the argument list will keep this value for all subsequent calls to the program during execution. In the latter case, the argument is reassigned its initial value for every call and is only changed if the proper optional argument is provided.

Thus for subroutine EG, the inexperienced user can get away with a three argument call, while the experienced user can specify as many of the optional arguments as desired.

An extension of the variable length argument list method described above uses NAMELIST input to read in values of particular optional arguments. This permits the user to

(1) not have to specify the previous values of say fifteen arguments in order to change the sixteenth argument to a new value

(2) not have to specify argument values in any particular order since NAMELIST input accepts these in any order.

A program displaying this use of NAMELIST input is shown in Figure 1.11. It is basically the same as that in Figure 1.10 except that when the first optional argument OPARG1 is

FIGURE 1.11

NAMELIST ARGUMENT TRANSMISSION

```
      SUBROUTINE EG2(ARRAY,ISIZE,JSIZE,OPARG1,OPARG2,
     *OPARG3,....)
C     OPARG REPRESENTS OPTIONAL ARGUMENT
C     DECLARATIONS
      DIMENSION ARRAY(ISIZE,JSIZE)
      LOGICAL OPARG3,ARG3
      NAMELIST/INPUT/NTAPE,ARG4,ARG5,ARG6
C     INITIALIZE DEFAULT VALUES
      DATA ARG2,ARG3,ARG4,ARG5,ARG6,/0.0,
     *5.0,.FALSE.,7.69,10.1/
      DATA NTAPE/6/
      ARG1=10.0
          .
          .
          .
          .
C     ANY MORE INITIALIZATIONS TAKE PLACE HERE
          .
          .
          .
          .
C     FIND OUT HOW MANY ARGUMENTS THIS
C     SUBROUTINE WAS CALLED WITH
      CALL NUMP(NARG)
C     THREE ARGUMENTS ARE MANDATORY FOR THIS ROUTINE
      IF (NARG.GT.2) GO TO 500
C     WRITE OUT ERROR MESSAGE
      WRITE(6,100)
100   FORMAT('TOO FEW ARGUMENTS IN CALL TO EG')
      RETURN
500   NARG=NARG-2
C     RESET VALUES OF SPECIFIED ARGUMENTS
      GO TO (1,2,3,....) NARG
          .
          .
          .
          .
C     ONE WAY OF RESETTING IS AS FOLLOWS
          .
          .
          .
          .
4     IF (OPARG3) 5,3
5     ARG3=OPARG3
3     ARG2=OPARG2
2     ARG1=OPARG1
```

continued......

```
C       IF REQUIRED PERFORM A NAMELIST READ
        IF (OPARG1.GE.0.0) GO TO 6
        READ(5,INPUT)
6       CONTINUE
           .
           .
           .
           .
C       BODY OF PROGRAM
           .
           .
           .
           .
        END
```

negative a NAMELIST read is performed to obtain new values of specified optional arguments. Here, again all arguments are initialized to default values for the inexperienced users' sake.

Using this method of argument transmission does have disadvantages over the previous variable length argument list method. These are found in the inefficiency caused by the slower processes of manipulation of character strings and dictionary look-up of variable names involved in NAMELIST reads. However the method really shows its usefulness when long arguments lists are at stake.

The methods mentioned above were discussed in the hope that further interpretations of the material will promote more useful application programs for both inexperienced and experienced users.

1.3.4. The Associative Memory

The associative memory is the final element of COMS to be discussed. It was already introduced in section 1.3.1.4 where the STRAN statements for the storage and retrieval of ordered n-tuples of symbols (strings of characters) were described.

The main purpose of the associative memory is to permit the COMS programmer (i.e. the systems programmer) to store and retrieve factual information about COMS and the program library. This information can then be used in the translation of command languages, developed for the inexperienced computer user, into formal internal commands which control the operations of COMS. In other words, it is the associative memory that serves as the communicator between the vast computational facilities of a computer and the inexperienced computer user. In very simple COMS implementations, a satisfactory system can be set up without the use of the associative memory. Here the user is assumed to already have the basic information needed and hence no outside help (the associative memory) is needed. A software system such as COMS that is developed entirely without an equivalent type of associative memory mechanism will find itself restricted to a particular class of users - those that want to take the time and effort learning the intricate programming details involved in running the system.

1.3.4.1. Use of the Set Theoretic Language (STL)

Originally the associative memory was developed for the specific application of storing and retrieving sentences in finite set theory. It turned out that although this was only one of a myriad of potential uses for the associative memory, it proved extremely fruitful in the development and interpretation of command languages for inexperienced users. The reason for this was due to the inherent logic in finite set theory which could easily be used to produce simple deductive processes. These processes provided COMS with an "intelligence" to translate a simple command language statement given by the inexperienced user into useful programming actions. Thus the rigorous formalities of current programming languages could be left up to COMS while the user could concentrate more specifically on the particular problem being solved.

To adapt sentences of finite set theory for the ordered n-tuple form used by the associative memory, the Set Theoretic Language (STL) was developed by Gammill [3]. This language is simply an encoding of sentences of finite set theory in a particular form easily adaptable to the associative memory mechanism. This is shown in Figure 1.12(a). Finite set theory defines properties and relations between sets. This is also true of STL but to become meaningful the individual letters representing sets are expanded to more

FIGURE 1.12(a)

SET THEORETIC LANGUAGE

	Set Theoretic Language	Language of Finite Set Theory
(1)	(a,b)	$b \in a$
(2)	(d,e,f)	$\langle e,f \rangle \in d$
(3)	(g,h,j,k)	$\langle h,j,k \rangle \in g$
(4)	($\subset$,j,k)	$j \subset k$

$\in$ is a member relation

$\subset$ is a subset

FIGURE 1.12(b)

Expanded Set Theoretic Language N-Tuples

(1) (MAN,NORMAN)

(2) (FATHER OF,NORMAN,ERIC)

(3) (CHAIN OF&AND ,GRANDFATHER OF,FATHER OF,PARENT OF)

(4) (SUBSET OF,FATHER OF,PARENT OF)

expressive English words with symbols such as $\subset$ being replaced by "SUBSET OF". The result as one possibility is the four STL n-tuples dealing with human relationships as shown in Figure 1.12(b).

In general, there are three types of symbols permitted in STL. One is to represent the domain of the problem space being considered. In Figure 1.12(b), this domain is shown to be that of people such as NORMAN or ERIC. The second is either a property of the domain such as MAN or a relation of the domain such as FATHER OF or GRANDFATHER OF. The third expresses a property or relation on the properties or relations already existing. These latter symbols are termed "primitives" and are used to define relationships occurring throughout the total associative structure. Examples of these are SUBSET OF or INVERSE OF.

The ability to state properties and relations of the properties and relations defined for a problem space makes STL a very powerful fact retrieval language. This ability implies that symbols appearing in the first position of an STL sentence (n-tuples) may also appear elsewhere. An example of this is (CHAIN OF&AND, SUBSET OF, SUBSET OF) which using the primitive CHAIN OF&AND states that the relation SUBSET OF is a CHAIN OF the relation SUBSET OF and the relation SUBSET OF. Another example, using the primitive INVERSE OF, is (INVERSE OF, INVERSE OF, INVERSE OF) which states that the relation INVERSE OF is the INVERSE OF the relation INVERSE OF.

Any symbol appearing in the first position of an STL sentence is either a property or relation. In 2-tuples such as (BOY,ERIC) it is always a property, otherwise it is a relation as in the 3-tuples (PARENT OF, MARVIN, NORMAN).

It is not difficult to see that a complicated set of relationships for a problem space can be quickly built up using the simple sentences of STL. The process of model building using STL becomes one of identifying the relevant domain (for example humans) and the relations and properties involved in this domain. Once a set of primitive relations and properties have been worked out, deductions concerning relationships between elements of the domain can be drawn from the model and thence information can be retrieved which was not actually entered beforehand.

There is a large variety of primitive relationships available for the STL model builder. The set of these is by no means complete but the ones shown in Figure 1.13 have proven useful in previous work on model building. Once a model is constructed for a particular problem space, the properties and relations of that model can be stored in the associative memory. STRAN procedural definitions (i.e. rule sets) can then be written for each of the primitives desired. Using these definitions COMS can in turn produce deductions

FIGURE 1.13

A SET OF USEFUL PRIMITIVES

(1) (CHAIN OF&AND,A,B,C)

(2) (SUBSET OF,A,B)

(3) (DISJOINT FROM,A,B)

(4) (LEFT INTERSECTION OF&AND,A,B,C)

(5) (RIGHT INTERSECTION OF&AND,A,B,C)

(6) (INVERSE OF,A,B)

(7) (INTERSECTION OF&AND,A,B,C)

(8) (UNION OF&AND,A,B,C,)

(9) (LEFT HALF OF,A,B)

(10) (RIGHT HALF OF,A,B)

EXAMPLES OF THE ABOVE PRIMITIVES USING HUMAN RELATIONSHIPS

(1) (CHAIN OF&AND,GRANDPARENT OF,PARENT OF,PARENT OF)

(CHAIN OF&AND,UNCLE OF,BROTHER OF,PARENT OF)

(2) (SUBSET OF,FATHER,MAN)

(SUBSET OF,HUSBAND,MAN)

(3) (DISJOINT FROM,CHILD,ADULT)

(DISJOINT FROM,UNMARRIED,MARRIED)

(4) (LEFT INTERSECTION OF&AND,WIFE OF,MARRIED TO,WOMAN)

(LEFT INTERSECITON OF&AND,FATHER OF,PARENT OF,MAN)

(5) (RIGHT INTERSECTION OF&AND,WIFE OF,WOMAN,MARRIED TO)

continued .....

(RIGHT INTERSECTION OF&AND,FATHER OF,PARENT OF,MAN)

(6) (INVERSE OF,PARENT OF,OFFSPRING OF)

(INVERSE OF,HUSBAND OF,WIFE OF)

(7) (INTERSECTION OF&AND,GIRL,CHILD,FEMALE PERSON)

(INTERSECTION OF&AND,BACHELOR,UNMARRIED,MAN)

(8) (UNION OF&AND,PARENT,FATHER,MOTHER)

(UNION OF&AND,SIBLING,BROTHER,SISTER)

(9) (LEFT HALF OF,WIFE,WIFE OF)

(LEFT HALF OF,CHILD,CHILD OF)

(10) (RIGHT HALF OF,OFFSPRING,MOTHER OF)

(RIGHT HALF OF,OFFSPRING,FATHER OF)

Note: the correct way to translate an STL sentence into English is as follows:

(SUBSET OF,HUSBAND,MAN)

the translation is: HUSBAND is the SUBSET OF MAN.

For a 4-tuple such as (CHAIN OF&AND,GRANDPARENT OF, PARENT OF,PARENT OF) the translation is: GRANDPARENT OF is the CHAIN OF PARENT OF and PARENT OF.

from the model required by the inexperienced user.

1.3.4.2. An Example Deduction

If the relations (INVERSE OF, ABOVE, BELOW) and (ABOVE, LAMP, TABLE) are stored in the associative memory, deductions made by STRAN rules for the primitive relation INVERSE OF will entail searching for all the ordered triples that have as their first element INVERSE OF. Upon finding the sentence (INVERSE OF, ABOVE, BELOW), the STRAN rules will search for all the ordered triples beginning with ABOVE. When the triple (ABOVE, LAMP, TABLE) is found the automatic deduction (BELOW, TABLE, LAMP) is made and stored in the memory. Thus information is deduced and stored that was not previously available.

Appendix C contains three STRAN programs dealing with human relationships which use the associative memory. Program (3) shows two STRAN rule sets called ASSERT and ANSWER that store and retrieve ordered n-tuples respectively. Program (4) translates simple English sentences into STL and stores relevant information obtained from these in the associative memory. Finally program (5) uses the information previously stored by programs (3) and (4) and makes deductions leading to new information which in turn is stored away.

1.3.5. Conclusions

The software package introduced in this chapter describes a system consisting of a number of elements, each capable of performing certain tasks with regard to the programming difficulties inherent in any particular study area. COMS is a flexible system having as a major attribute extendability of use through modification of rule sets and additions to both the library and the facts stored in the associative memory.

The descriptions and examples given really only touch on the possible systems one can develop using COMS. The flexibility involved in being able to use some or all of COMS elements to provide different programming environments is really one of its great features. Its true strength however will be shown as more and more sophisticated users produce more and more sophisticated implementations through the development of a hierarchy of models, each able to override the statements of those below.

It is easily seen that provided with a proper set of grammar rules used to generate sequences of evaluatable statements as the result of simple commands and provided with the proper set of well written application programs, COMS through its ability to make logical deductions of new information, could assist in a great variety of computer programming applications in any field where research investigations require the use of a computer. Section 2.3 of chapter 2, as an example of a particular COMS application, discusses the development of an operating system command language using COMS, for use in a parallel programming environment.

At this point mention should be made of the disadvantages in using the COMS system. The major disadvantage here is in the massive amount of execution time needed for the operation of each of COMS program elements. Of course it would be very hard to develop a system such as COMS and provide the flexibility it does without using a great deal of computer time to do this. Overcoming long execution times would entail the rewriting of some of the COMS Fortran subroutines in assembly language. One place where this might be done is in the evaluator program which is well defined for the particular job it is doing and is not likely to be changed for different applications of the COMS system.

A second disadvantage in the use of COMS is the need for a relatively large machine to provide its memory requirements. There are several ways to alleviate this problem however, including a garbage collection routine to clear all unwanted storage locations originally allocated by COMS and allowing only those rules and n-tuples which are absolutely necessary for a particular COMS implementation to be stored in memory.

On the whole COMS presents a system that lends itself to change and emphasizes simplicity and flexibility over efficiency. The most exciting possibilities for its future use lie in the extension of its modelling capabilities since the design of the system was specifically created for this purpose.

CHAPTER 2

COMS AND CONTROL STRUCTURES

2.1. Introduction

Although the major part of this project centers around the introduction and implementation of the COMS system at McMaster, a small amount of research was also carried out by this author into both the possible incorporation of particular control structures into COMS and conversely, the possible use COMS might have in the field of control structures. More specifically, both the advantages of using coroutines instead of subroutines in the COMS library, and the development of a command language for the simulation of a parallel processing environment are discussed.

The field of control structures in general refers to the programming environments or operations which specify the sequencing and interpretation rules for programs and parts of programs. Included in this field are such controls as sequential processing, subroutines, parallel processing, coroutines, recursion, conditional and unconditional operations, iteration, continuous evaluation, and monitoring. Using the communication management system to develop models for some or all of the above controls would allow investigation of the processes involved and although simulation would have to be carried out in current sequential processing environments, systems could be constructed to allow the user to formulate new control structures not before conceived. The goal would be a better understanding of control structures leading (with particular respect to COMS) to a more powerful facility for the inexperienced computer user.

It is beyond the scope of this project to provide an implementation of the above discussion, but the ideas are presented for possible future research.

2.2. COMS and Coroutines

One factor on which to base the efficiency of the communication management system is the way it handles the external application programs found in the program library. Since COMS was specifically designed to provide easy user access to such a set of programs, any method of improving this accessibility would seem to this author to increase overall efficiency.

An important aspect involved in accessibility is found not only in the actual loading and execution of desired routines, but also in the transference of data by the COMS system to and from these routines. It would seem in the present version of COMS that due to the loss of evaluator-program library communication via NAMELIST-input (executed by the application routines as discussed in section 1.3.3.2), the system is not as flexible as it might be. Thus any method of improvement with regard to the external data communication of the program library will certainly benefit the COMS user.

More specifically, if an application program in the library is so structured that its paths of execution are entirely dependent on the results produced from a previous call to it, alot of unnecessary information will have to be passed to this program to have it run correctly (by this is meant both the variable settings resulting from the previous call, and the state of processing within the routine indicating where the current call is to continue processing). The reader will immediately say that this concept is certainly not particular to programs in the COMS library but is present in many subprogram applications in general use. This of course is very true, but due to the significantly greater amount of processing time involved with argument transference in the COMS system (as compared with the execution time of typical compiled code for program-subprogram communication in other programming languages), any method used to avoid unnecessary subprogram communication in COMS will certainly help the system's efficiency.

A possible solution to the above problem lies in the replacement of particular subroutines in the program library by coroutines. The word "coroutine" was coined by Melvin E. Conway in 1958 after he had developed the concept and applied it to the construction of an assembly language. Independent studies of coroutines were also carried out concurrently by Joel Erdwinn and J. Merner, but the first published explanation did not appear until 1963 when Conway wrote an article for the Communications of the ACM on the design of a seperable transition-diagram compiler [4]. The coroutine concept has not been widely discussed since its initial introduction, but its usefulness in particular program applications can easily be demonstrated through examples given later in the discussion. Before further discussion on the incorporation of coroutines into the COMS system, a brief introductory description of their major features will be given.

In contrast to the unsymmetric relationship between a main program and a subroutine, there is complete symmetry between coroutines. Every subroutine has a return address which is saved while the subroutine is being performed, and which is different each time the subroutine is called. When the subroutine is not being performed, no return address needs to be saved. Thus this makes a subroutine subordinate to its main program. If, however, the main program and the subroutine work as a team of programs where the main program calls the subroutine when it is needed and the subroutine calls the main program when it is needed, the result is a set of coroutines, neither subordinate to the other. When control passes from one coroutine to another, the coroutine which is being entered takes up where it last left off, and the address at which the other coroutine transferred control, plus one, is saved as the return address to that coroutine. This type of linkage is termed "bilateral".

Coroutines come under the classification of control structures (as described by D. A. Fisher [5]), their principle advantage as such being that each of several processes can be described as a principle routine with minimal concern for other processes.

To implement the facilities provided by coroutines in a high level language such as FORTRAN (that is, to retain the state of processing within a subroutine so that processing can continue from that point at the next call), the only mechanism that can be used is a "switch" which selects the proper point of re-entry specified by a label attached to each of the desired entry points.

This is the type of program the COMS library can really do without. The reader at this point may think that routines such as this will not appear very often in regular programming practice. The fact of the matter is that they do, simply because they are frequently based on input and output operations and these nobody will argue appear very frequently. To illustrate this point, take a situation where COMS is being used to study the lexical scan techniques used by different programming language compilers. Involved in this study is the development of a simple command language to load and execute sections of coding from different compilers and record data (say execution times) on the efficiency of each scan executed. Assume that for a particular programming language, part of the lexical scan process involves reading characters one at a time from an input card (starting at column 1 of card 1 say) and pairing off any occurrence of two adjacent asterisks, replacing these by the single character "↑". Also, any characters encountered between two ↑'s are output to the next print line with the symbol "≡" used to indicate the termination of that line. Thus for example take the input string

DECLARE...**DATA ITIME/5/**ASSIGN...**Y=5.2**

The part of the lexical scan described above will first form the new string

DECLARE...↑DATA ITIME/5/↑ASSIGN...↑Y=5.2↑

and then output the following two lines

DATA ITIME/5/≡
Y=5.2≡

The program to accomplish this task has been written with both main routine - subroutine and coroutine - coroutine (i.e. bilateral) linkage techniques. A block diagram displaying the two linkages is shown in Figure 2.1(a) and 2.1(b). Figure 2.2 shows the flowcharts of the read subroutine RDCARD (used to read single characters from a card) and the scanning subroutine SQUISH (used to replace ** with ↑). The writing subroutine WRITE which is called by SQUISH for its output operations is shown in Figure 2.3. Both SQUISH and WRITE have been rewritten as coroutines in Figures 2.4 and 2.5 respectively. Examination of the subroutines SQUISH and WRITE reveals that a switch is necessary to describe the execution path each time either of these routines are called. The need for this switch is a direct result of the entry point of the subroutine always remaining the same (of course different entry points could be used but then a switch in the main program would be needed to select the proper call).

The coroutine approach to the problem accomplishes the switching of entry points implicitly by use of the calling sequence. That is, coroutines SQUISH and WRITE are connected such that SQUISH runs for a while until it encounters a write operation which means it needs coroutine WRITE. Control is transferred to WRITE until this coroutine finds it needs another character. SQUISH is called and is entered <u>at the place where it last left off</u>. The point here is that by careful positioning of calls to other coroutines,

FIGURE 2.1(a)

MAIN ROUTINE - SUBROUTINE LINKAGE

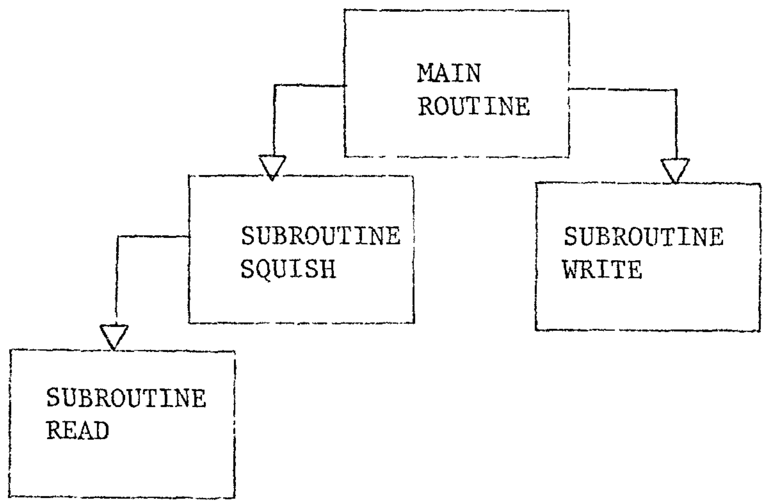


FIGURE 2.1(b)

COROUTINE - COROUTINE LINKAGE

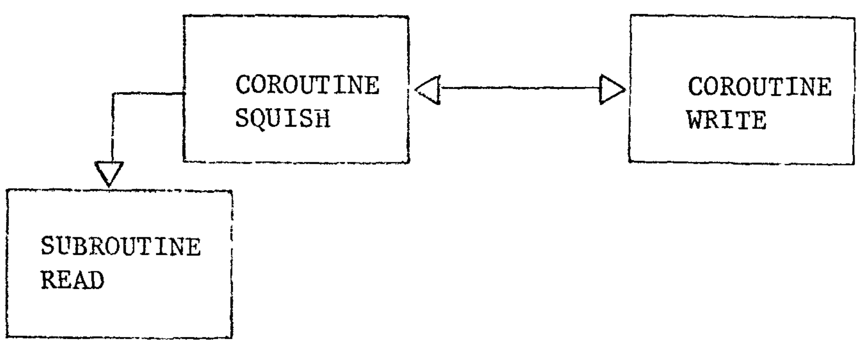
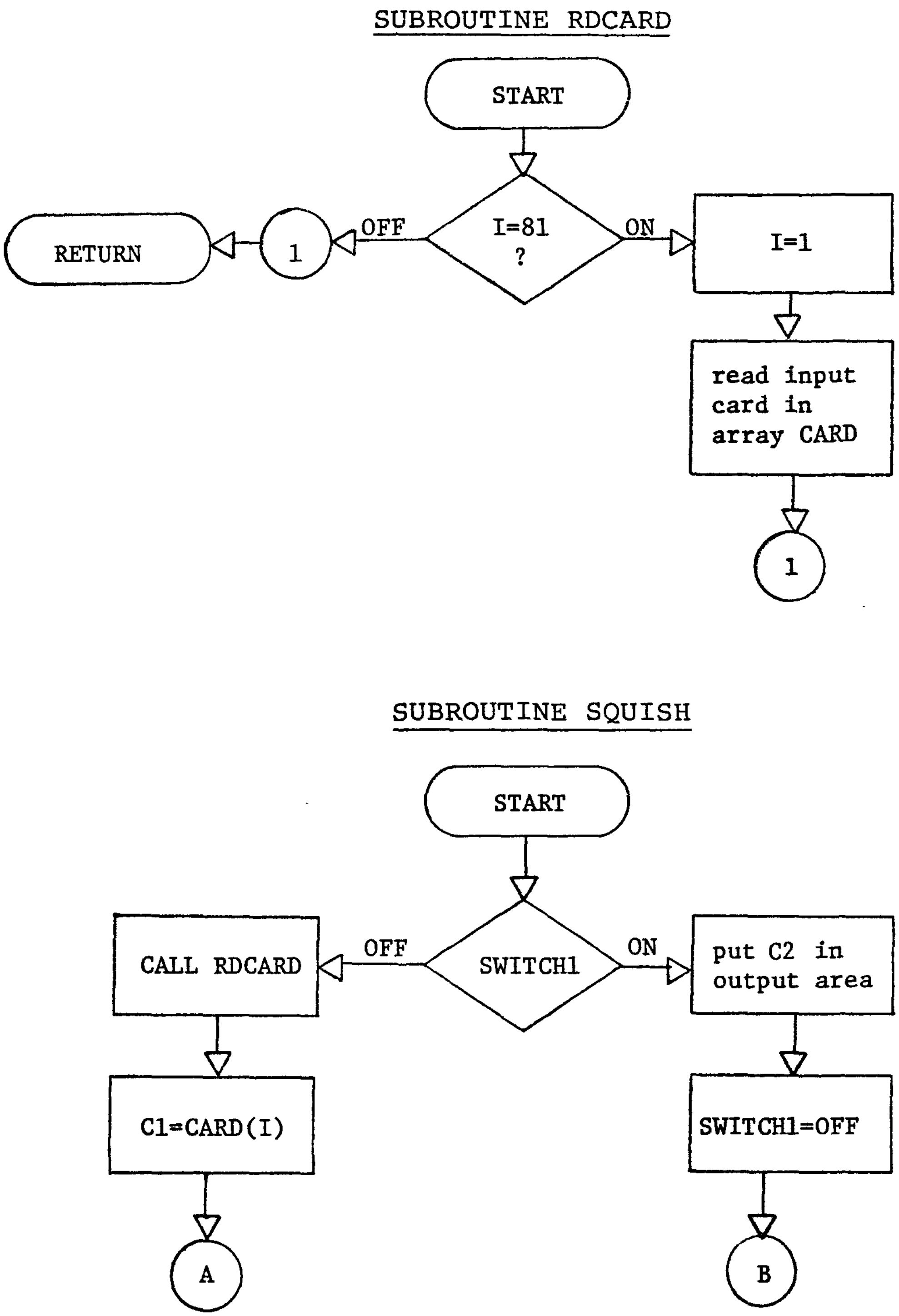


FIGURE 2.2

FLOWCHART OF SQUISH, RDCARD SUBROUTINES



Note: I is initialized to 81, SWITCH1 is initialized to OFF, main program calls subroutine WRITE to start off.

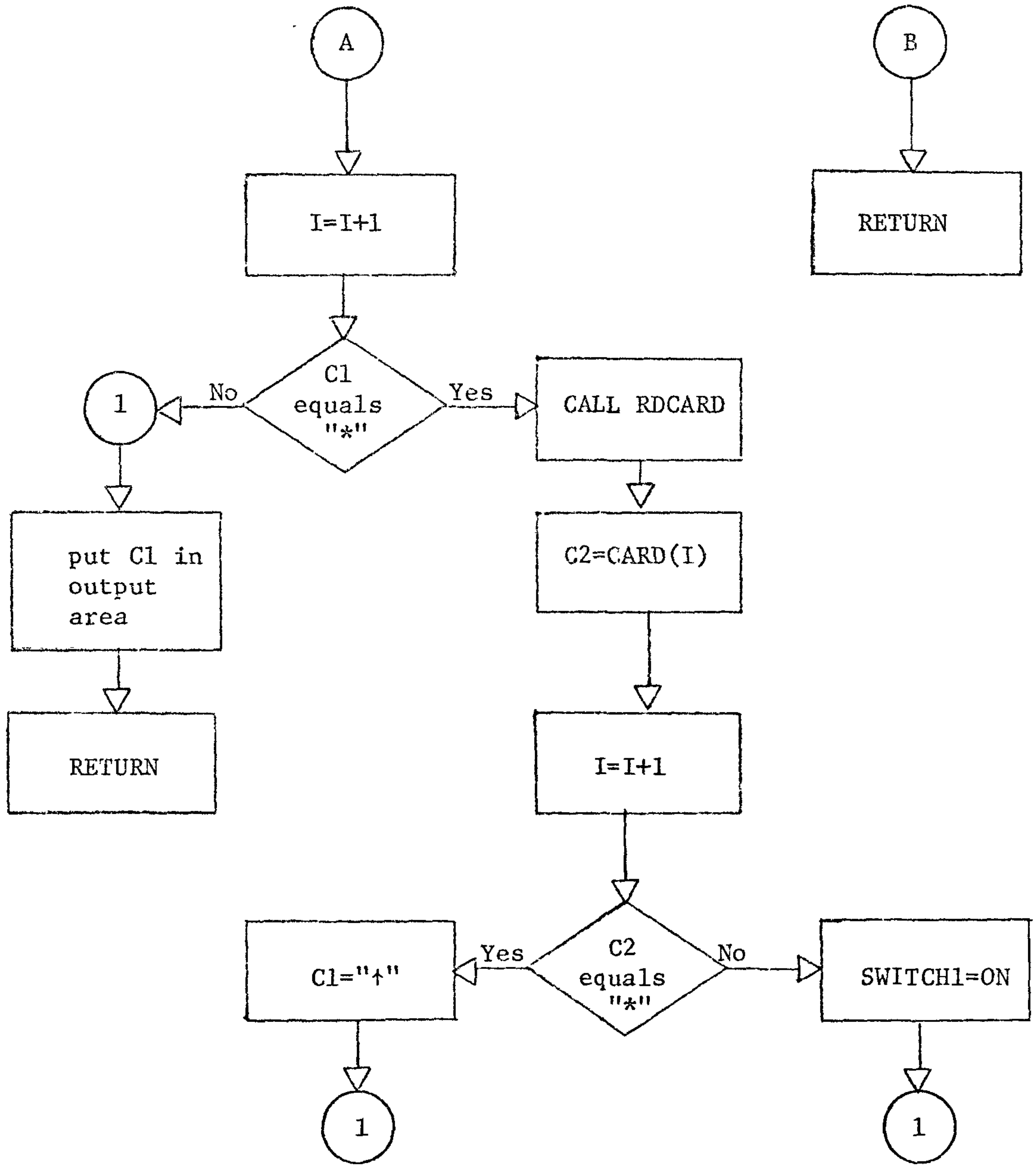
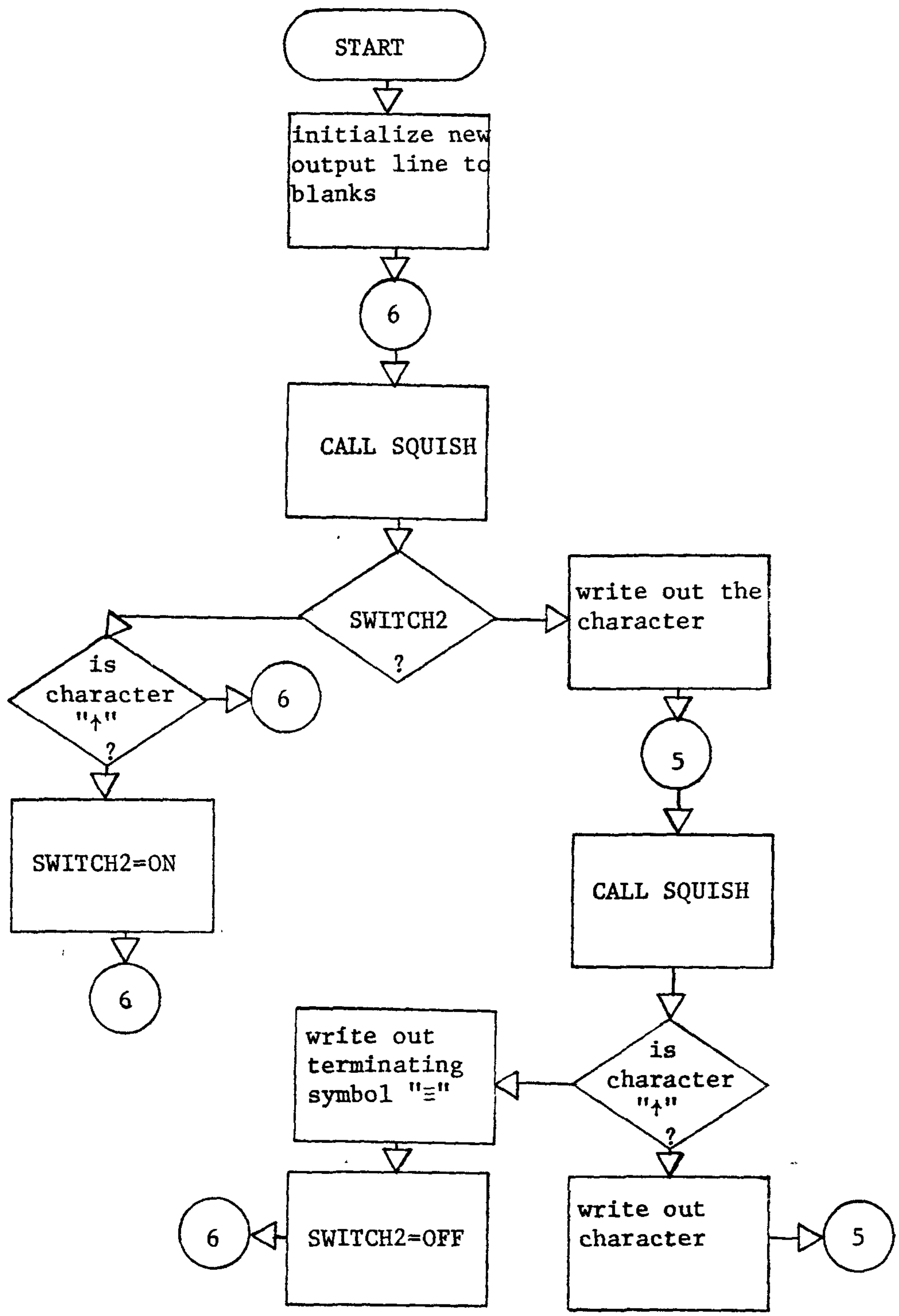

A
B
I=I+1
RETURN
C1 equals "*"
No
Yes
1
CALL RDCARD
put C1 in output area
C2=CARD(I)
RETURN
I=I+1
C2 equals "*"
Yes
No
C1="↑"
SWITCH1=ON
1
1

FIGURE 2.3

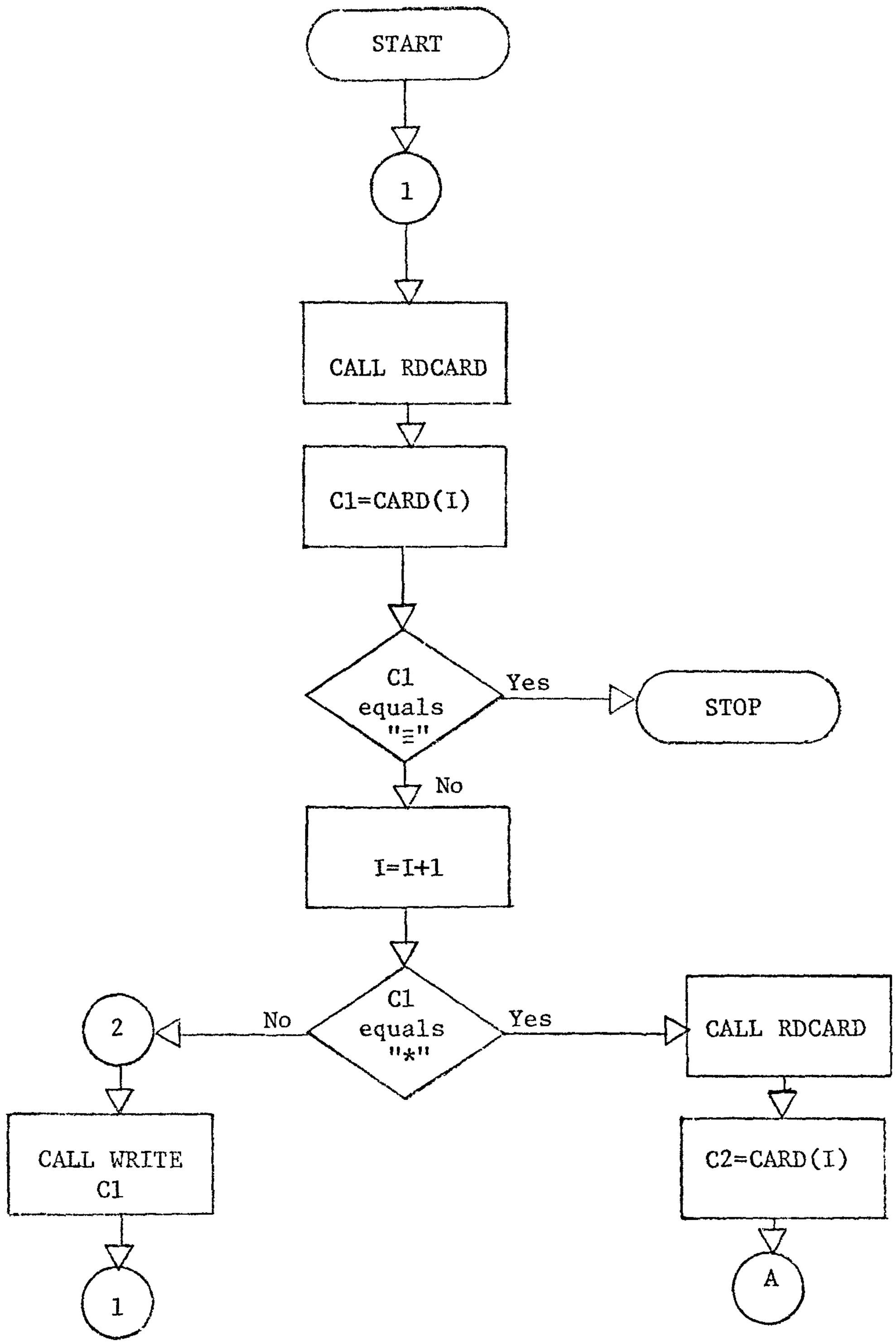
SUBROUTINE WRITE



Note: SWITCH2 is initialized to OFF

FIGURE 2.4

COROUTINE SQUISH



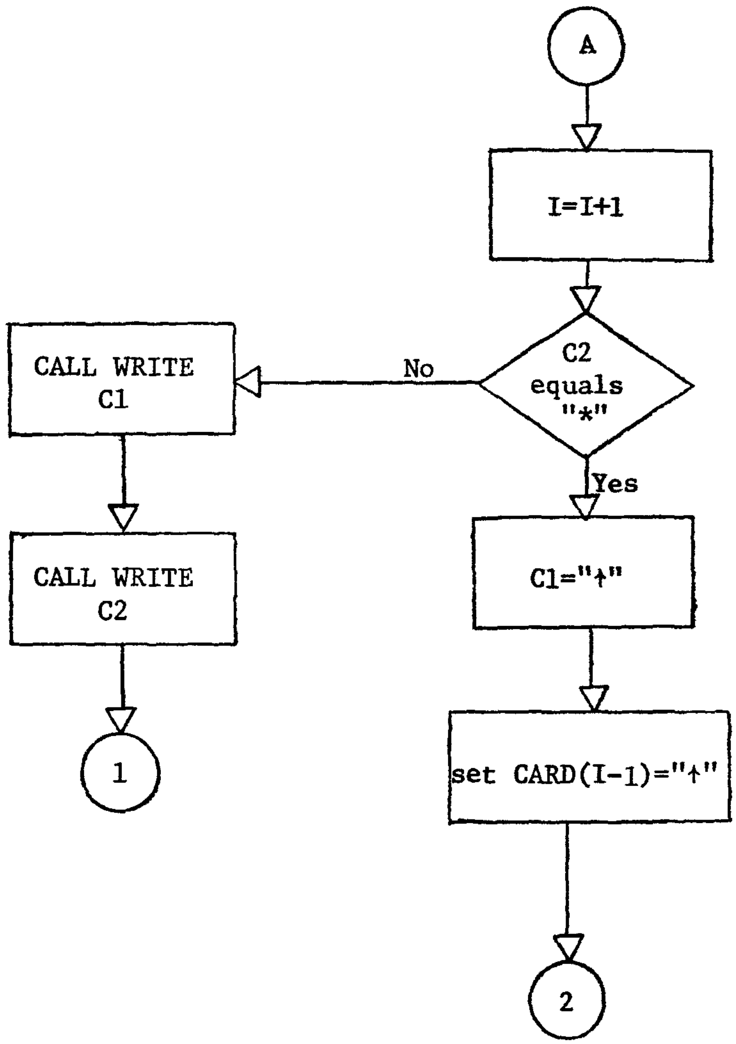
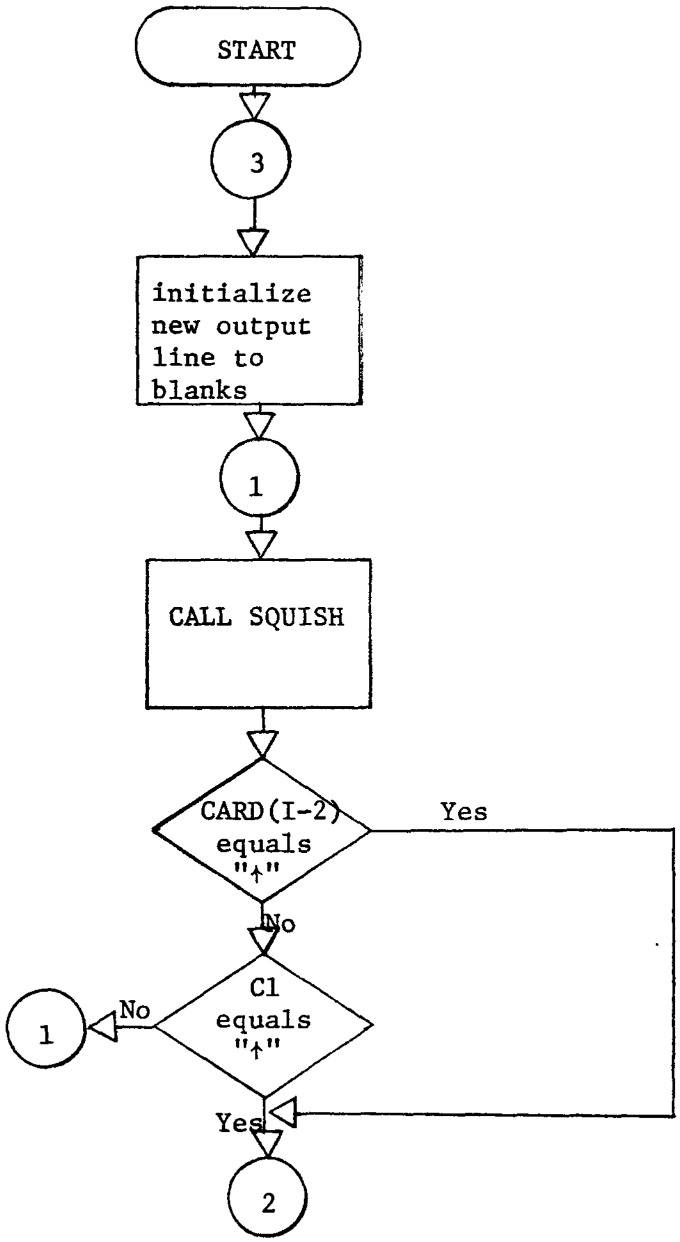
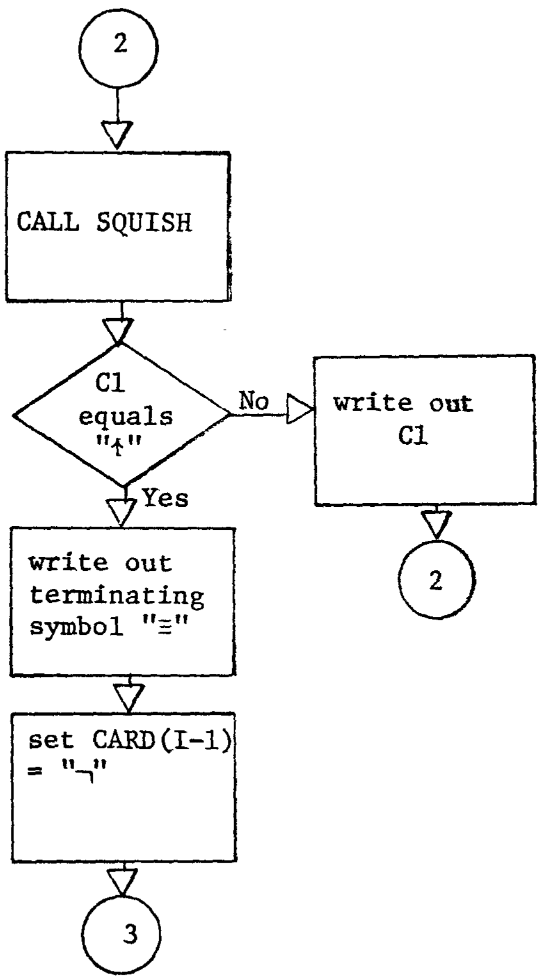

A
I=I+1
C2
equals
"*"
No
Yes
CALL WRITE
C1
CALL WRITE
C2
1
C1="↑"
set CARD(I-1)="↑"
2

FIGURE 2.5

COROUTINE WRITE




2
CALL SQUISH
C1
equals
"↑"
No
write out
C1
2
Yes
write out
terminating
symbol "≡"
set CARD(I-1)
= "¬"
3

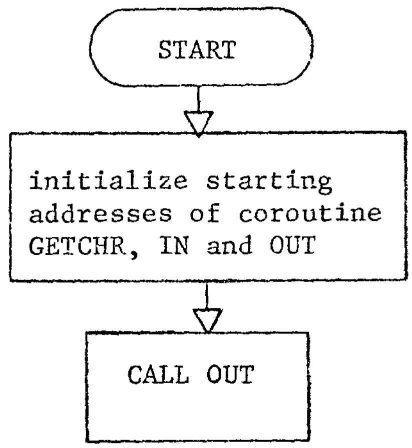
all switching processes are eliminated (this leads to the important relation coroutines have to multiple pass algorithms as discussed in the next section). An implementation of the above example is shown in Appendix H. The programming is done in Fortran except for an assembly language routine called COR which provides the bilateral linkage needed for proper coroutine execution.

It is not difficult to find short, simple examples of coroutines which illustrate the importance of the idea, but the most useful coroutine applications (for example a lexical scanner and a syntax analyser acting as coroutines) are usually quite lengthy.

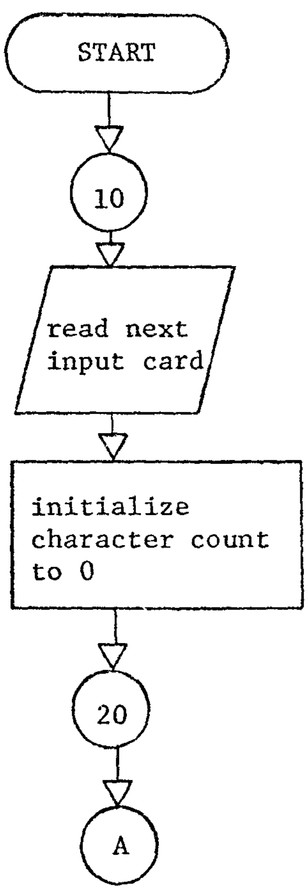
For the interested reader, a further example is shown in Figures 2.6 and 2.7. Three coroutines are involved here-namely GETCHR, IN and OUT. Again, input - output operations are involved in the translation of a "coded" sequence of alphabetic characters terminated by a period. The "code" involves the following: if the next character of the input string (read from left to right) is a digit, say n, it indicates (n+1) repetitions of the following character, whether the following character is a digit or not. A non-digit simply denotes itself. The program output consists of the sequence indicated in this manner and separated into groups of three characters each (where the last group may have less than three characters). For example the input

FIGURE 2.6

MAIN PROGRAM



COROUTINE GETCHR



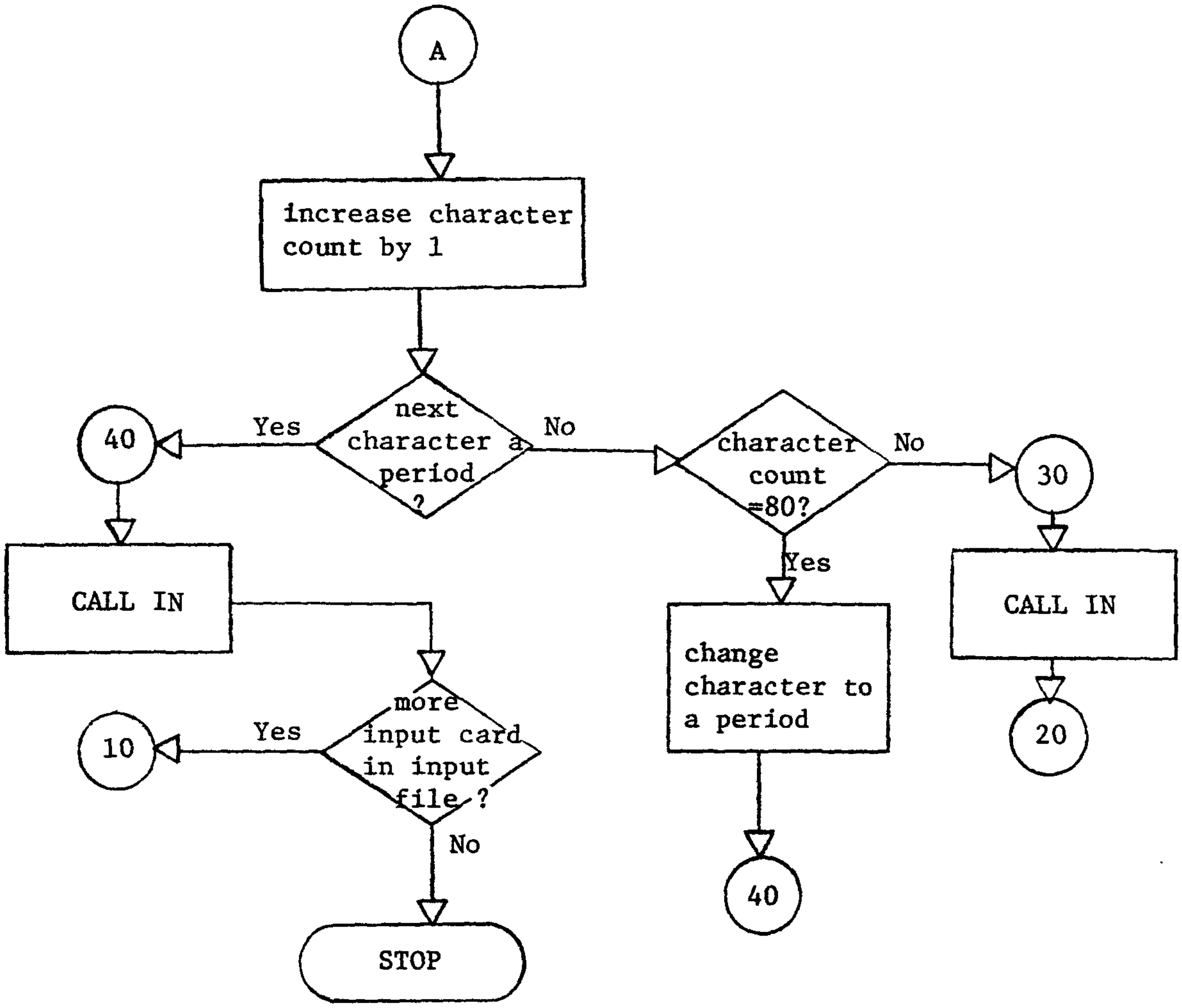
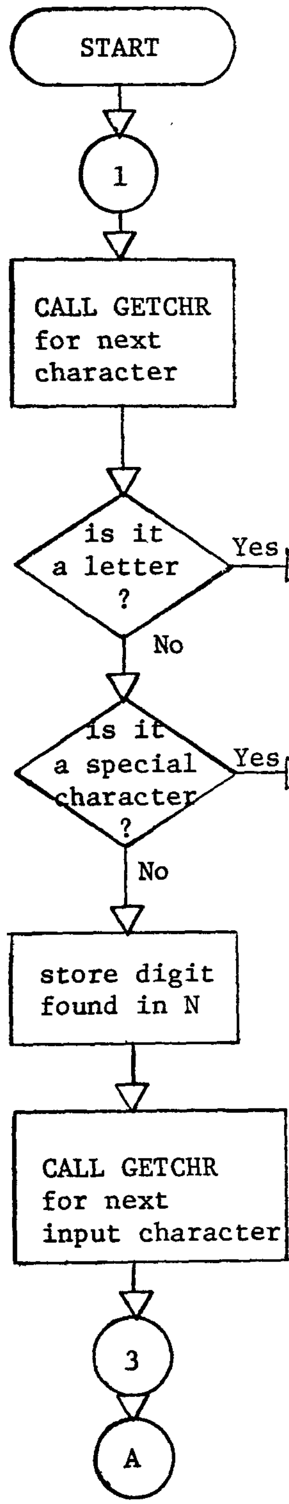
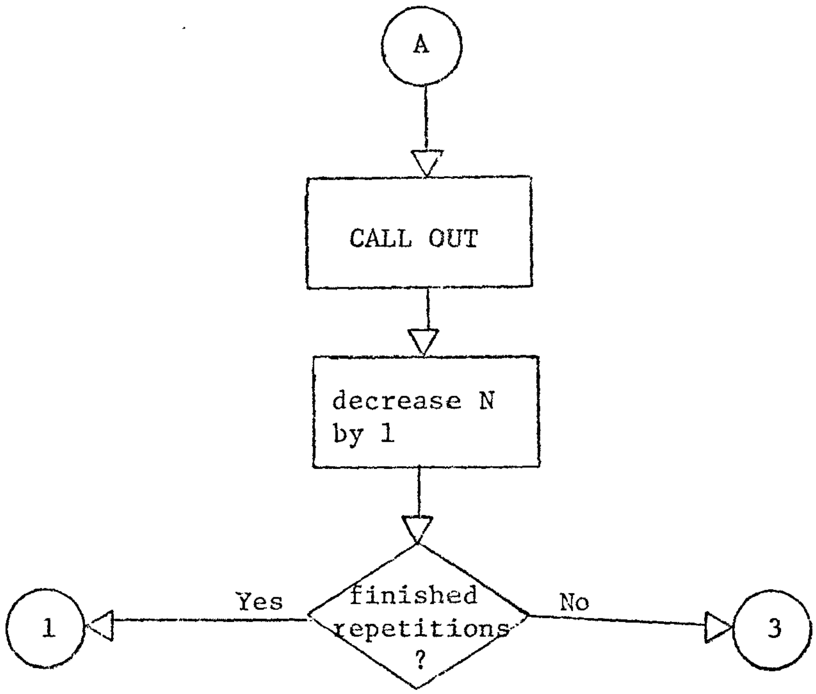

A
increase character count by 1
next character a period ?
Yes
No
40
CALL IN
character count =80?
No
Yes
30
CALL IN
20
change character to a period
40
more input card in input file ?
Yes
10
No
STOP

FIGURE 2.7

COROUTINE IN

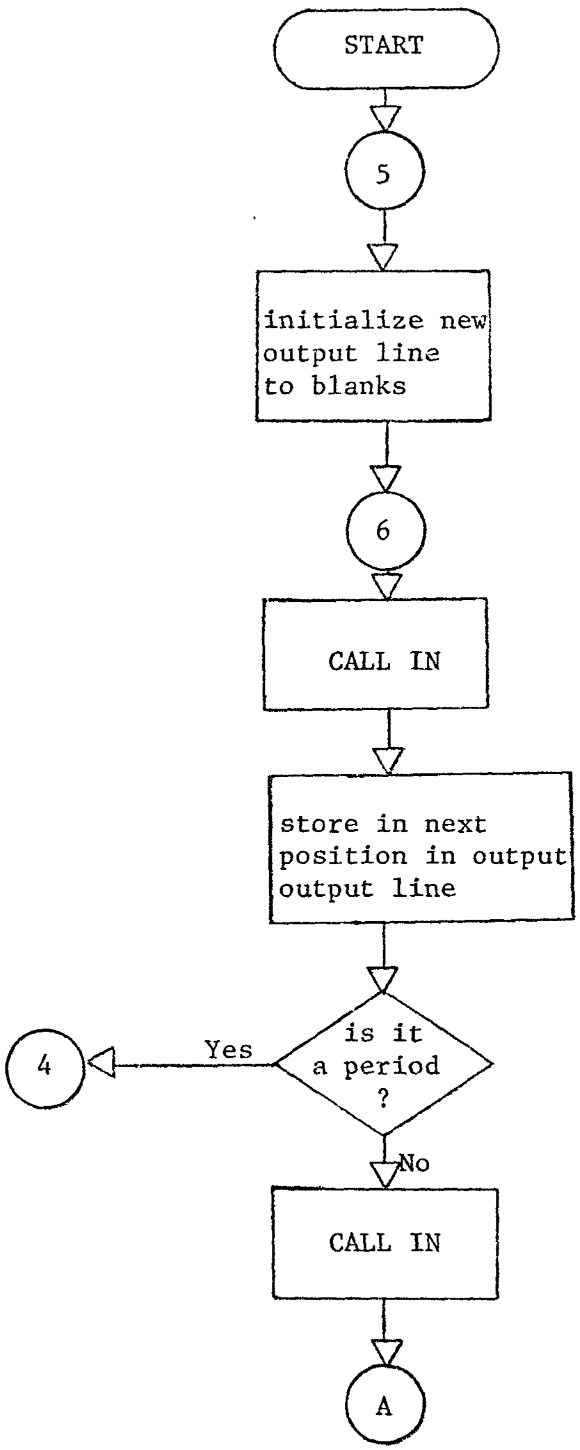


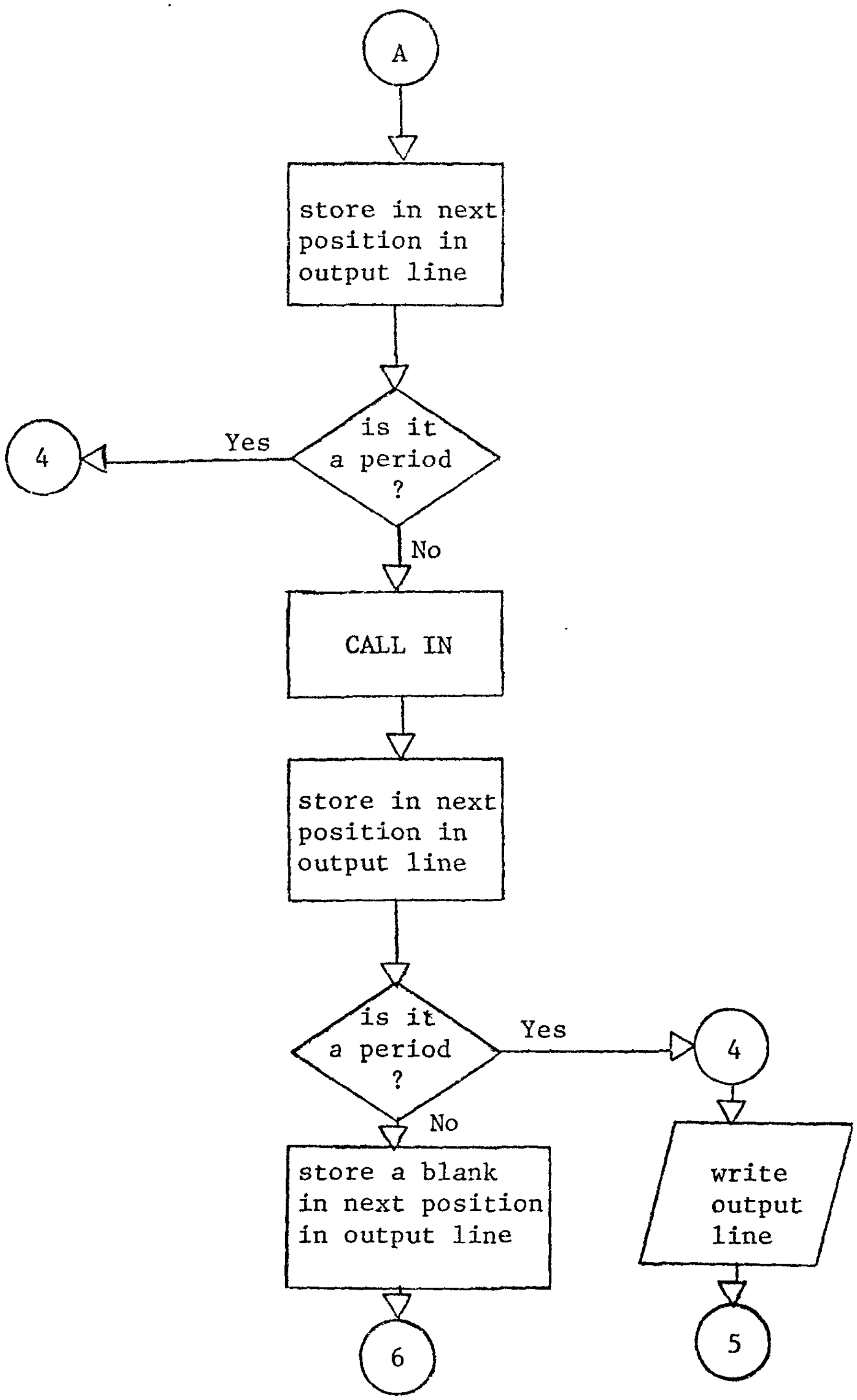


continued.........

FIGURE 2.7 (continued)

COROUTINE OUT




A
store in next position in output line
is it a period ?
Yes
4
No
CALL IN
store in next position in output line
is it a period ?
Yes
4
No
store a blank in next position in output line
write output line
6
5

string

A2B5E3426.

should be translated by the program into

ABB BEE EEE E44 446 66.

To accomplish this task coroutine GETCHR is used to read in one input card at a time and send individual characters to coroutine IN. The job is complete if no more input cards can be found in the input file. An error check is also made for a missing period (the string terminating symbol) by not allowing the character count on any one card to exceed 80. If the character count equals 80 and a period is still not found, the 80th character is set to a period and the next input card is read in (if one exists). Coroutine IN checks whether each character is a letter, special character or digit. Letters and special characters are immediately passed to coroutine OUT for placement in the output line, whereas digits initiate a looping process which sends the required number of repetitions of the following character to OUT (this is done one at a time). Coroutine OUT stores groups of three characters separated by a blank in the output line. Printing of this line is not done until a period is encountered.

The reader should note that the calls form one coroutine to another have been carefully placed for implicit recognition of the required actions to be taken by the program. Of course writing coroutines (in Fortran at least) is a little more involved than writing subroutines, but to reiterate, in longer more complex applications the extra time is well worth it.

2.2.1. Coroutines and Multiple-Pass Algorithms

It is important at this point to assert the relationship of coroutines to multiple-pass algorithms, and the effect this relationship can have on the COMS library. With regard to the second example of section 2.2.1 involving code translation, the process used could have been accomplished in two distinct passes rather than just one. This would entail using coroutine IN by itself to write the required number of character repetitions from the input string onto (say) magnetic tape, rewinding the tape, and using coroutine OUT by itself to read these characters from the tape and write them out in groups of three.

The point is that a process done by say n coroutines can often be transformed into an n-pass process and conversely, an n-pass process can often be transformed into a single pass process using n coroutines (an exception to this type of transformation involves forward referencing where one pass cannot proceed without information returned from a later pass). Assuming no forward references are needed, Figure 2.8 illustrates the coroutine - multiple-pass relationship. If coroutines A, B, C and D of Figure 2.8(b) are substituted for the respective passes A, B, C, D of Figure 2.8(a) the result is as follows. Coroutine A will jump to B when pass A would have written an item of output on tape 1; coroutine B will jump to A when pass B would have read an item of input from tape 1, and B will jump to C when pass B would have written an item of output on tape 2; etc. Thus, what previously took four passes to accomplish now only takes one.

In most cases, the COMS user can take distinct advantages of the coroutine - multiple-pass algorithm relationship in that any group of programs which are to be used in the COMS library and which depend on multiple-pass algorithms to produce their results can be rewritten as coroutines and used in a single-pass fashion. The major advantage here is in the time saved in not having to transfer data back and forth between the evaluator and the library (of course there is also the possibility of using secondary storage to hold the necessary data since this would also result in some time saved). The disadvantage in using coroutines to eliminate the above transferral of data stems from the resulting additional memory requirements. Specifically, enough fast core memory is needed to simultaneously store all the programs involved in the process. This problem

FIGURE 2.8(a)

MULTIPLE-PASS ALGORITHM

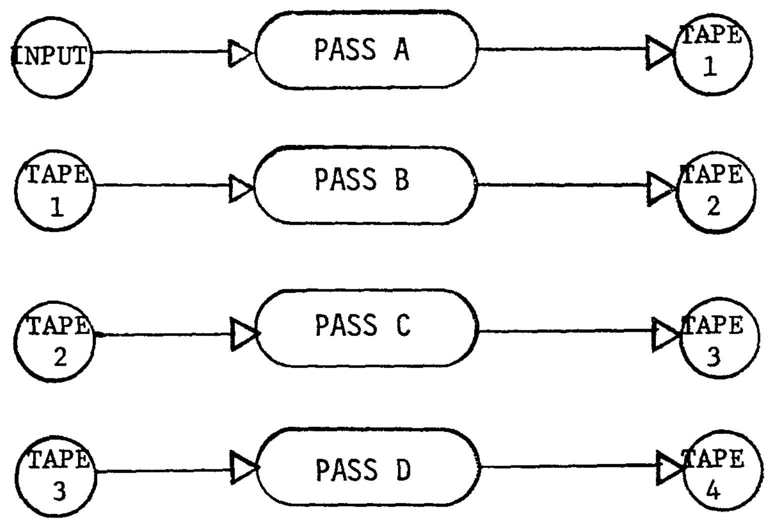
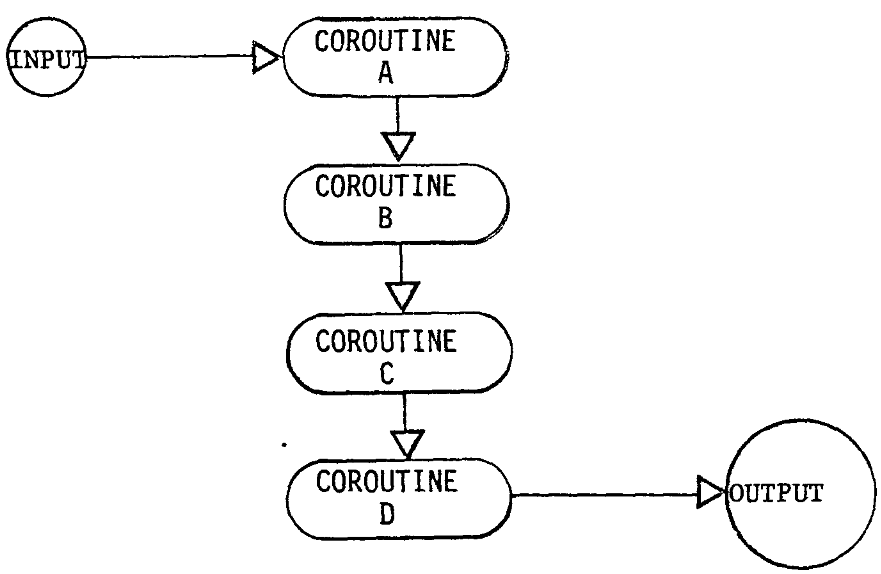


FIGURE 2.8(b)

ONE-PASS ALGORITHM



is partially remedied by reducing the original number of passes involved until the memory core limit is reached. This involves for a four pass algorithm say, writing only the number of coroutines that will fit into available core. This may mean that the four pass process is only reduced to say a three pass process, but with COMS the time saved will still prove advantageous.

Having seen what a particular control structure can do for COMS, the next section describes briefly what COMS can do for control structures.

2.3. Soapsuds

An interesting application of the communication management system to the field of control structures is in the development of command languages for the implementation of a simulated parallel processing environment. The most common type of parallel system configuration is the multi-processor, i.e. several central processors with a shared storage. Soapsuds[1] is an assembly language program written for the CDC 6600 which simulates such a configuration provid-

---

[1]Soapsuds is an offshoot of "WATCHER", a former debugging aid for the CDC 6600 which simulated the running of the 6600 central processor. Soapsuds, like Watcher, uses the program to be simulated as data, analyzing the instructions and performing the operations they request.

ing for up to a maximum of sixty central processing units, each with its own location counter, operating asynchronously, from a common memory. Thus a possible use of COMS with respect to the Soapsuds program is in the development of a parallel operating system[2] command language to perform such tasks as (1) re-distributing processors, as they become available, to various tasks attempting to run in the simulated parallel environment; (2) conversely, handling the assignment of tasks to processors; and (3) accepting and managing the I/O for all processors. The type of questions that experimentation with such an operating system will answer include (1) how to route the CPUs between several jobs in a minimum of time; (2) how to establish a job mix that keeps all CPUs occupied; (3) how to define and implement priority and (4) how to optimize throughput.

Linking COMS to Soapsuds should help answer these questions since the flexibility inherent in the COMS system can be used to quickly deduce the best way of approaching any of the above problems given a choice of possibilities. In other words, the details of the simulation will be left to Soapsuds, while commands developed for COMS will stem from both the descriptions of the general features available (i.e. what Soapsuds

---

[2]Such an operating system has been partically developed by E. Draughton as part of an experimentation with Soapsuds [6].

can actually do) and the results recorded on the performance of the system in a variety of program applications.

Performance characteristics are easily obtained through the trapping, tracing and checking options available in Soapsuds. "Trap" options are used for such things as counting the frequency of opcodes, loads or stores, turning on or off other available options at particular places in the program being executed, and checking the values of special machine locations at particular instants. "Trace" options are a special form of trap option where a message is printed out describing the required tracing procedure. "Checking" options check whether a specified location bears a particular relation to another specified location. Also available are timing options which keep track of system time, program time, idle time and tracing time. Finally, at the end of a program simulation, Soapsuds prints out the present status of all processors, and the running and idle times of the same. Other performance characteristics regarding the efficiency of programs executed in parallel as compared to serially executed programs is available (according to the authors of Soapsuds) from the following calculation:

$$E_n = \frac{T_1}{n*T_n}$$

where E_n = efficiency of n CPUs

T_1 = time required for a serial machine to perform the program

n = number of CPUs

T_n = time required for n CPUs to perform the program.

Incorporating the Soapsuds program into the COMS system for use in the development of an operating system command language can be accomplished by writing Fortran routines (to be placed in the COMS library) to provide the necessary controls needed in the operating system. These routines could then make calls to Soapsuds to perform the required simulations, and the resulting information (as described in the previous paragraph) could then be recorded in the associative memory. Deductive processes using the associative memory data might then lead to further improvements in the system.

A command language developed through COMS for the control of parallel processing operations would be very useful in the actual implementation of an operating system for a real multi-processor computer (including one that if and when developed has sixty central processing units), since the various techniques for the organization and control of such a multi-processor system will most certainly become apparent.

REFERENCES

[1] Yngve, V.H.: "An Introduction to COMIT Programming", MIT Press, Cambridge, Mass., 1962.

[2] McCracken, D.D.: "A Guide to FORTRAN IV Programming", John Wiley & Sons, Inc., New York, 1968.

[3] Gammill, R.C.: "COMS": Communication Management System", Ph.D. Thesis, University of Colorado, May, 1969.

[4] Conway, M.E.: "Design of a Seperable Transition Diagram Compiler", Commun. ACM, July, 1963 pp 396-401.

[5] Fisher, D.A.: "Control Structures For Programming Languages", Ph.D. Thesis, Carnegie-Mellon University, May, 1970.

[6] Draughon E., Grishman R., Schwartz J., Stein A.: "Programming Considerations For Parallel Computers", New York University, Courant Institute of Mathematical Sciences, November, 1967.

[7] Draughon, E., Schwartz J., Stein A.: "Individual and Multi-Processing Performance Characteristics of Programs on Large Parallel Computers", New York University, Courant Institute of Mathematical Sciences, April, 1970.

[8] Wegner, P.: "Programming Languages, Information Structures and Machine Organization", McGraw-Hill, New York, 1968.

[9] Lorin, H.: "Parallelism in Hardware and Software: Real and Apparent Concurrency", Prentice-Hall Inc., 1972.

[10] Knuth D.E.: "The Art of Computer Programming", Vol. 1, Addison-Wesley, 1969.

[11] Mauer,D.W.: "Programming: An Introduction to Computer Languages and Techniques", Holden-Day, Inc., 1969.

[12] Johnson L.R.: "System Structure In Data, Programs, and Computers", Prentice Hall, Inc., 1970.

[13] Gries D.: "Compiler Construction For Digital Computers", John Wiley & Sons, Inc., 1971.

APPENDIX A

STRAN SYNTAX

The following shows the complete STRAN syntax in BNF for the present version of the interpreter:

```
    <STRAN RULE>::= (<RULE NAME>(<TYPE 1 BODY>|
                    <TYPE 2 BODY>|<TYPE 3 BODY>)
    <TYPE 1 BODY>::=<RULE NAME>{,<RULE NAME>}*)
    <TYPE 2 BODY>::=<LITERAL PATTERN>)
    <TYPE 3 BODY>::=<RULE BODY>)<GO-TO SECTION>
     <RULE NAME>::=<NAME>
<LITERAL PATTERN>::={'<CHARACTER STRING}⊕'
     <RULE BODY>::=<LHS>=<RHS>|<LHS>|=<RHS>
  <GO-TO SECTION>::=<RULE NAME>|<RULE NAME>,<RULE NAME>
          <NAME>::=<LETTER><ALPHANUM CH>*
                    (maximum of ten characters)
   <ALPHANUM CH>::=<DIGIT>|<LETTER>|<ZERO>
         <DIGIT>::=1|2|3|4|5|6|7|8|9
        <LETTER>::=A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|
                    T|U|V|W|X|Y|Z
          <ZERO>::=0
           <LHS>::={<VARIABLE NAME>/<DECOMP PAT>/|
                    +F<DIGIT>/<FIND PAT>/|
                    +A<DIGIT>/<ACCESS PAT>/}⊕
```

<VARIABLE NAME>::=<NAME>

<DECOMP PAT>::=<DECOMP OP>{+<DECOMP OP>}$^{\oplus}$

<DECOMP OP>::=$|$<DIGIT>|$<LITERAL>|<VARIABLE NAME>|
↓<DIGIT>|<LITERAL>|·<DIGIT>

<LITERAL>::='<CHARACTER STRING>'

<CHARACTER STRING>::=(any sequence of one or more basic 6-bit display BCD characters)

<FIND PAT>::=<FIND OP>{+<FIND OP>}$^{\oplus}$
(a maximum of 4 find operators is allowed in one find pattern)

<FIND OP>::=$|<LITERAL>|<DIGIT>

<ACCESS PAT>::=<DIGIT>{+<DIGIT>}$^{\oplus}$
(a maximum of 3 digits is allowed in an access pattern)

<RHS>::={{*|,}* <VARIABLE NAME>/<COMP PAT>/|
+S/<STORE PAT>/}$^{\oplus}$

<COMP PAT>::=<COMP OP>{+<COMP OP>}$^{\oplus}$

<COMP OP>::=<DIGIT>|<LITERAL>|↓<DIGIT>|
≡L<NUMB>,<DIGIT>|≡R<NUMB>,<DIGIT>

<NUMB>::=<DIGIT>|<DIGIT><DIGIT>
(numb has a maximum value of 80)

<STORE PAT>::=<STORE OP>{+<STORE OP>}$^{\oplus}$
(maximum of 4 store operators in a store pattern)

<STORE OP>::=<LITERAL>|<DIGIT>

Note:

* indicates 0 or more repetitions

⊕ indicates 1 or more repetitions

{} indicates multiple construct repetitions

APPENDIX B

STRAN Decomposition and Composition Operators

Decomposition Operators:

1) $ - dollar sign matches any arbitrary character string including the null string.

2) literal - string of characters surrounded by single quote marks matches only an exact occurrence of the contained string of characters.

3) storage name - this storage location must contain a sequence of literals separated by single quotes and terminated by two right parentheses. The first of these literals to produce a satisfactory match is used. An example of the contents of the storage location is:

'THE'A'AN'))

4) $n - dollar sign followed by an integer matches the first n characters of the remaining string

5) $'character string' - dollar sign followed by a literal, matches as many repetitions of the

literal as can be found in consecutive order. This does <u>not</u> include matching the null string - i.e. there must be at least <u>one</u> occurrence of the literal present.

6) ↓n — downward arrow followed by an integer indicates the next character will come from column n. Thus this operator matches characters through column n-1 if the present position in the character string lies before column n. If the present position lies past column n, the operator matches the null string and backs up so that the next character will come from column n.

7) ·n — period followed by an integer, causes the contents of pseudo-register n to be inserted (at execution time) in the operator string for this operator.

Note (1) If a literal occurs in the left or rightmost position of a decomposition operator string, the left or rightmost portion of the string to be matched must duplicate the literal exactly.

Note (2) The elements of a match are placed in successive pseudo-registers where they remain until wiped out by the left side of some later rule.

Composition Operators:

1) integer from 1 to 9 - these refer to the nine pseudo-registers.

2) literal - string of characters surrounded by single quote marks.

3) ↓n - downward arrow followed by an integer, causes composition to continue at column n, either by adding blanks or truncating.

4) ≡Ln,m - equivalence followed by an L followed by two integers separated by a comma, causes the leftmost m characters of the string in pseudo-register n to be concatenated to the result. If there are less than m characters, the string is padded on the right with blanks.

5) ≡Rn,m - exactly the same as (4) except the rightmost m characters are used. If not enough characters are available padding with blanks is on the left.

APPENDIX C

Sample STRAN Programs

The sample programs presented in this appendix show how each of the software elements of COMS may be used to perform various operations including arithmetic calculations, pattern matching of character strings, storage and retrieval of information and referencing the program library. Examples will show how combinations of the interpreter, evaluator and associative memory programs can be used to perform different tasks.

Program execution for these examples was carried out on the CDC 6400 computer at McMaster.

PROGRAM 1

A simple STRAN rule set to read in character strings and send them to the evaluator is shown. Input information to the evaluator is terminated in each case by a semicolon, whereas comments used to describe the actions being performed are separated from the evaluator input by use of a period placed in column one of the comment card. Lines beginning with INPUT... are echos of the input cards read by the interpreter. Output from the evaluator is shown directly beneath each echoed line.

The reader should note the use of the dollar sign operator preceding a variable name (for example $X) to obtain the relative machine address of the variable.

Errors have purposely been placed in two input strings to show evaluator response under these conditions.

Sample calculations of formulae using some of the normal built-in arithmetic functions of Fortran are also given.

```
BEGIN READING RULES.

INPUT...(CALCUL(READ,EVAL))
INPUT...(READ(*INPUT/≠.≠+$/=*INPUT/2/)READ,END)
INPUT...(EVAL(INPUT/$+≠;≠+$/=,OUTPUT/1/)PRINT,END)
INPUT...(PRINT(OUTPUT/$/=*OUT/1/)CALCUL)
INPUT...(CALCUL)
```

```
BEGIN INTERPRETING RULES.

INPUT....    THE FOLLOWING EXAMPLES SHOW POSSIBLE INPUT STRINGS TO THE INTERPRETER
   THE FOLLOWING EXAMPLES SHOW POSSIBLE INPUT STRINGS TO THE INTERPRETER
INPUT....    (CALLING LIBRARY PROGRAMS IS DEALT WITH IN PROGRAM 6)
   (CALLING LIBRARY PROGRAMS IS DEALT WITH IN PROGRAM 6)
INPUT....

INPUT....

INPUT....

INPUT...Q=29.62:
Q=2.962000000000E+01$
INPUT....

INPUT...Z=1056;
Z=1056$
INPUT....

INPUT...X=4:
X=4$
INPUT....

INPUT...ABCDEFGHIJ=2348.62;
ABCDEFGHIJ=2.348620000000E+03$
INPUT....

INPUT...INTEGER IARRAY(5,2,2,3,2,1);
INTEGER IARRAY(5,2,2,3,2,1)
INPUT....

INPUT...REAL ARRAY(X,X,X,1);
REAL ARRAY(4,4,4,1)
INPUT....

INPUT...IARRAY(1,1,1,1,1,1)=Z;
IARRAY(1,1,1,1,1,1)=1056$
INPUT....
```

```
INPUT...Q:
2.962000000000E+01
INPUT....

INPUT...Z;
1056
INPUT....

INPUT...$Q:
13515
INPUT....

INPUT...$Z;
13515
INPUT....

INPUT...$X:
13517
INPUT....

INPUT...ARRAY;
12046
INPUT....

INPUT...$ARRAY;
12046
INPUT....

INPUT...IARRAY(1,1,1,1,1,1);
1056
INPUT....

INPUT...(700-925)/4;
-56
INPUT....

INPUT...2.963*2.0;
5.926000000000E+00
INPUT....
```

```
INPUT...SIN(1.7)*SQRT(12.8*16.923)-25.6+COS(0);
-1.00048499424 2E+01
INPUT....

INPUT....

INPUT....   THE FOLLOWING TWO STATEMENTS ARE PURPOSELY IN ERROR
   THE FOLLOWING TWO STATEMENTS ARE PURPOSELY IN ERROR
INPUT....

INPUT....

INPUT...2.1+6.2+ZQRPT;
ERROR IN NUMERIC STORAGE OR RETRIEVAL.
THE VARIABLE ZQRPT        HAS BEEN ASSIGNED A VALUE OF ZERO.
8.30000000000E+00
INPUT....

INPUT....

INPUT...IARRAY(6,8,5,4,3,8,3,5,2)=1;
ERROR IN INDICES.
ERROR IN NUMERIC STORAGE OR RETRIEVAL.
IARRAY(6,8,5,4,3,8,3,5,2)=1$
INPUT....

INPUT....

INPUT....

INPUT....

INPUT....   EXAMPLES OF CALCULATIONS
   EXAMPLES OF CALCULATIONS
INPUT....

INPUT...A=2.0;
A=2.00000000000E+00$
INPUT....
```

```
INPUT...B=4.0;
B=4.000000000000E+00$
INPUT....

INPUT...C=1.0;
C=1.000000000000E+00$
INPUT....

INPUT...DETERM=B**2-4.0*A*C;
DETERM=8.000000000000E+00$
INPUT....

INPUT...IDETERM=B**2-4*A*C;
IDETERM=8.000000000000E+00$
INPUT....

INPUT...ROOT1=(-B+SQRT(B**2-4.0*A*C))/2.0*A;
ROOT1=-1.171572875254E+00$
INPUT....

INPUT...ROOT1=(-B-SQRT(B**2-4.0*A*C))/2.0*A;
ROOT1=-6.828427124746E+00$
INPUT....

INPUT...H=1.0;
H=1.000000000000E+00$
INPUT....

INPUT...P=1.0;
P=1.000000000000E+00$
INPUT....

INPUT...A=3.14159;
A=3.141590000000E+00$
INPUT....

INPUT...E=27;
F=27$
INPUT....
```

```
INPUT...X=(E*H*P)/(SIN(A)*((H**4/16.0)+(H**2)*(P**2)));
X=9.576372621722E+06$
INPUT....

INPUT...PI=A;
PI=3.141590000000E+00$
INPUT....

INPUT...R=2.3;
R=2.300000000000E+00$
INPUT....

INPUT...CIRCAREA=2.0*P*R*SIN(PI/P);
CIRCAREA=1.220651307802E-05$
INPUT....
INPUT...REAL ARR1(4);
REAL ARR1(4)
INPUT....

INPUT...ARR1(1)=300.0;
ARR1(1)=3.000000000000E+02$
INPUT....

INPUT...ARR1(2)=400.0;
ARR1(2)=4.000000000000E+02$
INPUT....

INPUT...ARR1(3)=500.0;
ARR1(3)=5.000000000000E+02$
INPUT....

INPUT...S=(ARR1(1)+ARR1(2)+ARR1(3))/2.0;
S=6.000000000000E+02$
INPUT....

INPUT...TRIAREA=SQRT(S*(S-ARR1(1))*(S-ARR1(2))*(S-ARR1(3))):
TRIAREA=6.000000000000E+04$
INPUT....

INPUT....    END OF EXAMPLES
   END OF EXAMPLES
```

PROGRAM 2

Shown in this program are many of the STRAN pattern matching operators available in the interpreter. Fortran programs are read in as input data and each statement is analyzed with respect to its type. The operations performed include:

(1) all blanks except those in FORMAT statements are removed

(2) declaration statements including REAL, INTEGER, DIMENSION, COMMON, LOGICAL and DATA are headed with the string 'DECL...'

(3) assignment statements are headed with the string 'ASN...'

(4) DO statements are broken up into component parts, each separated by a comma (for example DO 5 I=1,10 would be changed to DO,5,I,1,10,1)

(5) statement labels are replaced by blanks

(6) continuation cards indicated by a character placed in column 6 are concatenated to the card immediately preceding the first continuation card.

(7) if more than one statement is found on a card (i.e. by use of the dollar sign which is permitted in many Fortran compilers) the statements

are separated and re-examined individually.

(8) once a statement has been examined, and the necessary operations performed, it is output to the print line.

The above operations are not necessarily meant to represent those processes which occur in the lexical and syntactic scans of a true Fortran compiler. Rather, they are used to demonstrate the many and varied pattern matching operations that one might want to perform on strings of characters in different situations.

There are two output listings for the STRAN program given. The first uses the default (ECHO) pseudo operator to output all lines read by the interpreter. This may become confusing as the interpreter output for particular input card may not appear until after a second input card has been read in and echoed out. Thus for clarity sake, a second output of the program is shown with no echoing of input cards (this was done with the use of the (NOECHO) pseudo operator).

```
BEGIN READING RULES.

INPUT...(FORTRAN(INIT,READ,LOOP))
INPUT...(INIT(=CARDNO/≠0≠/LOC/≠0≠/DOPD/≠≠/)END)
INPUT...(LOOP(GETLINE,COMPILE))
INPUT...(READ(*INPUT/≠C≠+$/=*INPUT/↓10+1+2/)READ,STEP)
INPUT...(STEP(CARDNO/$/=,CARDNO/≠1+≠+1/)PRINT)
INPUT...(PRINT(CARDNO/$/INPUT/$72+$/=*OUTPUT/=R1,4+↓10+2+3/INPUT/2/SERIAL/3/)END)
INPUT...(GETLINE(INPUT/↓6+≠ ≠+$≠ ≠+$/=LABEL/1/LINE/4/)GET,ERROR)
INPUT...(GET(PACK,ENDTEST))
INPUT...(PACK(LINE/≠FORMAT≠+$≠ ≠+≠(≠+$/=LINE/1+3+4/)END,PACK1)
INPUT...(PACK1(LINE/$+≠ ≠+$≠ ≠+$/=LINE/1+4/)PACK1,END)
INPUT...(ENDTEST(LINE/≠END≠/)END,DOLLARCHK)
INPUT...(DOLLARCHK(LINE/$+≠$≠+$/=LINE/1/INPUT/↓7+3/)END,CONTINUE)
INPUT...(CONTINUE(READ,ADD))
INPUT...(ADD(INPUT/↓6+≠ ≠+$/)END,ADD1)
INPUT...(ADD1(INPUT/$6+$≠ ≠+$/LINE/$/=LINE/4+3/)GET,ERROR)
INPUT...(ERROR(=*OUTPUT/≠PRECEDING CARD CANNOT BE HANDLED.≠/)END)
INPUT...*    COMPILATION RULES.
INPUT...(COMPILE(LINE/DECLOPS+$/=LINE/≠DECL...≠+1+2/)OUT,EQTST)
INPUT...(EQTST(LINE/$+≠=≠+$/)DOTEST,OTHER)
INPUT...(DOTEST(LINE/≠DO≠+DIGIT+$+LETTER+$+≠=≠+$/=DOLAB/2+3/VARB/4+5/RT/7/)DO1,ASN)
INPUT...(DO1(RT/$+≠(≠+$/)ASN,DO2)
INPUT...(DO2(RT/$1+$+≠,≠+$1+$/=ISTRT/1+2/ISTOP/4+5/ISTEP/≠1≠/)DO3,ASN)
INPUT...(DO3(ISTOP/$1+$+≠,≠+$1+$/=ISTOP/1+2/ISTEP/4+5/)DO4)
INPUT...(DO4(STEPLOC,DO5))
INPUT...(STEPLOC(END))
INPUT...(DO5(DOLAB/$/VARB/$/ISTRT/$/ISTOP/$/ISTEP/$/)DO6)
INPUT...(DO6(=LINE/≠DO,≠+1+≠,≠+2+≠,≠+3+≠,≠+4+≠,≠+5/)OUT)
INPUT...(OUT(LINE/$/=*OUTPUT/↓30+1/)LOOP)
INPUT...(ASN(LINE/$/=LINE/≠ASN...≠+1/)OUT)
INPUT...(OTHER(LINE/≠END≠/=*OUTPUT/↓30+1/)END,OUT)
INPUT...(DIGIT(≠0≠1≠2≠3≠4≠5≠6≠7≠8≠9≠))
INPUT...(LETTER(≠A≠B≠C≠D≠E≠F≠G≠H≠I≠J≠K≠L≠M≠N≠O≠P≠Q≠R≠S≠T≠U≠V≠W≠X≠Y≠7≠))
INPUT...(DECLOPS(≠REAL≠INTEGER≠DIMENSION≠COMMON≠LOGICAL≠DATA≠))
INPUT...(FORTRAN)
```

BEGIN INTERPRETING RULES.

```
INPUT...      SUBROUTINE NUMBER(CHAR,NBEG,ICHAR)
INPUT...      COMMON/CNTROL/ECHO,TRACE,JUNK(11),INTAP
                                 SUBROUTINENUMBER(CHAR,NBEG,ICHAR)
INPUT...      COMMON/ALGEBR/NAMES(5),SOURCE(100)
                                 DECL...COMMON/CNTROL/ECHO,TRACE,JUNK(11),INTAP
INPUT...      DIMENSION FLOT(100)
                                 DECL...COMMON/ALGEBR/NAMES(5),SOURCE(100)
INPUT...      INTEGER SHIER,SOURCE,TEST,CHAR(2),SAVE
                                 DECL...DIMENSIONFLOT(100)
INPUT...      LOGICAL FLAG,NUMBSW
                                 DECL...INTEGERSHIER,SOURCE,TEST,CHAR(2),SAVE
INPUT...      INTEGER FNAM
                                 DECL...LOGICALFLAG,NUMBSW
INPUT...      INTEGER FNAME(5)
                                 DECL...INTEGERFNAM
INPUT...      REAL X,M,U,I
                                 DECL...INTEGERFNAME(5)
INPUT...      DATA FNAME/3HMOD,5HFLOAT,3HFIX,
                                 DECL...REALX,M,U,I
INPUT...     1 3HABS,3HSIN/
INPUT...      NCHAR = ICHAR - NBEG $ IF(.NOT.NUMBSW) GO TO 3
                                 DECL...DATAFNAME/3HMOD,5HFLOAT,3HFIX,3HABS,3HSIN/
                                 ASN...NCHAR=ICHAR-NBEG
INPUT...      CALL GETNUM(SOURCE(L),CHAR(NBEG),NCHAR,1)
                                 IF(.NOT.NUMBSW)GOTO3
INPUT...      SHIER(L)=0
```

```
                                        CALLGETNUM(SOURCE(L),CHAR(NBEG),NCHAR,1)
INPUT...     1  CALL GETNUM(FLOT(L),CHAR(NBEG),NCHAR,3)

                                        ASN...SHIER(L)=0
INPUT...        SHIER(L)=1

                                        CALLGETNUM(FLOT(L),CHAR(NBEG),NCHAR,3)
INPUT...     2  L=L+1           $        FLAG=.FALSE.     $ RETURN

                                        ASN...SHIER(L)=1
                                        ASN...L=L+1
                                        ASN...FLAG=.FALSE.
INPUT...     3  CALL  PACK(CHAR(NBEG),FNAM,NCHAR)

                                        RETURN
INPUT...        DO 5  NF=1,12
                                        CALLPACK(CHAR(NBEG),FNAM,NCHAR)
INPUT...        DO 5  IJK=1,45,3

                                        DO,5,NF,1,12,1
INPUT...        IF(FNAM.NE.FNAME(NF))  GO TO 5

                                        DO,5,IJK,1,45,3
INPUT...        IF(NF.NE.1) GO          TO 4 $ SHIER(L)=5 $ SOURCE(L)=6

                                        IF(FNAM.NE.FNAME(NF))GOTO5
                                        IF(NF.NE.1)GOTO4
                                        ASN...SHIER(L)=5
INPUT...        SOURCE(L)  = NF         + 1

                                        ASN...SOURCE(L)=6
INPUT...                 GO                   TO                2

                                        ASN...SOURCE(L)=NF+1
INPUT...5       CONTINUE

                                        GOTO2
INPUT...C
         C
```

```
INPUT...C    NOW TREAT SUBSCRIPTED AND UNSUBSCRIPTED VARIABLES
           C    NOW TREAT SUBSCRIPTED AND UNSUBSCRIPTED VARIABLES
INPUT...C
           C
INPUT...        IF(TEST.NE.1H() GO TO 6
                              CONTINUE
INPUT...        MM = MM + 1
                              IF(TEST.NE.1H()GOTO6
INPUT...        IF(MM.GT.5) GO TO 777
                              ASN...MM=MM+1
INPUT...8       NAMES(MM) = FNAM
                              IF(MM.GT.5)GOTO777
INPUT...        SOURCE(L) = 13 + MM
                              ASN...NAMES(MM)=FNAM
INPUT...        SHIER(L) = 6
                              ASN...SOURCE(L)=13+MM
INPUT...        GO TO 2
                              ASN...SHIER(L)=6
INPUT...    6   CALL GETNL(CHAR(NBEG),NCHAR,SOURCE(L),FLOT(L))
                              GOTO2
INPUT...        IF(SHIER(L).GE.-2) GO TO 2
                              CALLGETNL(CHAR(NBEG),NCHAR,SOURCE(L),FLOT(L))
INPUT...        WRITE(NOUTAP,7) FNAM
                              IF(SHIER(L).GE.-2)GOTO2
INPUT...    7   FORMAT(*THE VARIABLE *A10* HAS BEEN ASSIGNED*)
                              WRITE(NOUTAP,7)FNAM
```

```
INPUT...        SOURCE(L) = 0
                                     FORMAT(*THE VARIABLE *A10* HAS BEEN ASSIGNED*)
INPUT...        SHIER(L) = 0
                                     ASN...SOURCE(L)=0
INPUT...        GO TO 2
                                     ASN...SHIER(L)=0
INPUT...   777  WRITE(NOUTAP,778)
                                     GOTO2
INPUT...   778  FORMAT(*0ERROR, ALGEBRAIC EXPRESSION*)
                                     WRITE(NOUTAP,778)
INPUT...        MM = 5
                                     FORMAT(*0ERROR, ALGEBRAIC EXPRESSION*)
INPUT...        GO TO 8
                                     ASN...MM=5
INPUT...        END
                                     GOTO8
                                     END
```

BEGIN INTERPRETING RULES.

```
SUBROUTINENUMBER(CHAR,NBEG,ICHAR)
DECL...COMMON/CNTROL/ECHO,TRACE,JUNK(11),INTAP
DECL...COMMON/ALGEBR/NAMES(5),SOURCE(100)
DECL...DIMENSIONFLOT(100)
DECL...INTEGERSHIER,SOURCE,TEST,CHAR(2),SAVE
DECL...LOGICALFLAG,NUMBSW
DECL...INTEGERFNAM
DECL...INTEGERFNAME(5)
DECL...REALX,M,U,I
DECL...DATAFNAME/3HMOD,5HFLOAT,3HFIX,3HABS,3HSIN/
ASN...NCHAR=ICHAR-NBEG
IF(.NOT.NUMBSW)GOTO3
CALLGETNUM(SOURCE(L),CHAR(NBEG),NCHAR,1)
ASN...SHIER(L)=0
CALLGETNUM(FLOT(L),CHAR(NBEG),NCHAR,3)
ASN...SHIER(L)=1
ASN...L=L+1
ASN...FLAG=.FALSE.
RETURN
```

```
      CALLPACK(CHAR(NBEG),FNAM,NCHAR)
      DO,5,NF,1,12,1
      DO,5,IJK,1,45,3
      IF(FNAM.NE.FNAME(NF))GOTO5
      IF(NF.NE.1)GOTO4
      ASN...SHIER(L)=5
      ASN...SOURCE(L)=6
      ASN...SOURCE(L)=NF+1
      GOTO2
C
C     NOW TREAT SUBSCRIPTED AND UNSUBSCRIPTED VARIABLES
C
      CONTINUE
      IF(TEST.NE.1H()GOTO6
      ASN...MM=MM+1
      IF(MM.GT.5)GOTO777
      ASN...NAMES(MM)=FNAM
      ASN...SOURCE(L)=13+MM
      ASN...SHIER(L)=6
      GOTO2
      CALLGETNL(CHAR(NBEG),NCHAR,SOURCE(L),FLOT(L))
```

```
IF(SHIER(L).GE.-2)GOTO2
WRITE(NOUTAP,7)FNAM
FORMAT(*THE VARIABLE *A10* HAS BEEN ASSIGNED*)
ASN...SOURCE(L)=0
ASN...SHIER(L)=0
GOTO2
WRITE(NOUTAP,778)
FORMAT(*0ERROR, ALGEBRAIC EXPRESSION*)
ASN...MM=5
GOTO8
END
```

PROGRAM 3

Two programs are shown here which carry out elementary operations for a fact retrieval system using the Set Theoretic Language. The first program, ASSERT, is used to store 2-tuples, 3-tuples and 4-tuples of information in the COMS associative memory. If an n-tuple has previously been stored, no action is taken by the program. However an n-tuple not previously encountered is stored in the memory and output to the print line.

The second program, ANSWER, is used to retrieve stored n-tuples from the associative memory. If all the components of the n-tuple being sought are known, the program will output either that the n-tuple is true or, that the truth or falsehood of the n-tuple is unknown, depending of course on whether the n-tuple has been previously stored or not. If some of the n-tuple components are not known then the program either outputs a statement saying that n-tuple sought has no answers (meaning it does not currently exist in the associative memory at all), or provides a list of all the answers found. Sample output of the ANSWER program is shown below:

N-TUPLE	CLOSED OR OPEN	PREVIOUSLY STORED?	OUTPUT OF PROGRAM ANSWER
(A,B,C)	CLOSED	YES	(A,B,C) IS TRUE

N-TUPLE	CLOSED OR OPEN	PREVIOUSLY STORED?	OUTPUT OF PROGRAM ANSWER
(A,B,C)	CLOSED	NO	THE TRUTH OR FALSEHOOD OR (A,B,C) IS UNKNOWN
(A,$,C)	OPEN	YES	(A,$,B) HAS THESE ANSWERS: . . . (list of answers) . . .
(A,$,C)	OPEN	NO	THERE ARE NO ANSWERS TO (A,$,C)

(The dollar sign character is used to indicate the particular component being sought).

The reader will find a list of the n-tuples being stored by ASSERT following the program listing. Recall that lines starting with INPUT... are echos of input cards being read. Each time an n-tuple is stored, it is also output. Following this, use is made of the ANSWER program to retrieve some of the stored information.

```
BEGIN READING RULES.

INPUT...* RULES FOR ASSERT.
INPUT...(ASSERT(READ,STORE,ATST))
INPUT...(READ(*INPUT/≠.≠+$/=*INPUT/2/)READ,END)
INPUT...(ATST(INPUT/$+≠) ≠+$/)END,ASSERT)
INPUT...(STORE(INPUT/$+≠ (≠+?+≠)≠+$/=OUT/3/INPUT/5/)STOR,END)
INPUT...(STOR(S5,STORE) )
INPUT...(S5(OUT/$+≠,≠+$+≠,≠+$+≠,≠+$+≠,≠+$/)END,S4)
INPUT...(S4(OUT/$+≠,≠+$+≠,≠+$+≠,≠+$/=+S/1+3+5+7/*OUT/≠ (≠+1+2+3+4+5+6+7+≠)≠/)END,S3)
INPUT...(S3(OUT/$+≠,≠+$+≠,≠+$/=+S/1+3+5/*OUT/≠ (≠+1+2+3+4+5+≠)≠/)END,S2)
INPUT...(S2(OUT/$+≠,≠+$/=+S/1+3/*OUT/≠ (≠+1+2+3+≠)≠/)END,S1)
INPUT...(S1(OUT/$/=OUT/1+≠))≠/)OUT) THIS RULE ALLOWS A GOTO ON A RULE NAME
INPUT...* RULES FOR ANSWER. USES RULES FROM ASSERT.
INPUT...(ANSWER(READ,FIND,NTST))
INPUT...(NTST(INPUT/$+≠) ≠+$/)END,ANSWER)
INPUT...(FIND(INPUT/$+≠(≠+$+≠)≠+$/=*OUT/≠ ≠/OUT/3/INPUT/5/)FND,END)
INPUT...(FND(F5,FIND))
INPUT...(F5(OUT/$+≠,≠+$+≠,≠+$+≠,≠+$+≠,≠+$/)END,F4)
INPUT...(F4(OUT/$+≠,≠+$+≠,≠+$+≠,≠+$/)AN4,F3)
INPUT...(F3(OUT/$+≠,≠+$+≠,≠+$/)AN3,F2)
INPUT...(F2(OUT/$+≠,≠+$/)AN2,S1)
INPUT...(AN4(+F1/1+3+5+7/)FOUND,FAIL)
INPUT...(AN3(+F1/1+3+5/)FOUND,FAIL)
INPUT...(AN2(+F1/1+3/)FOUND,FAIL)
INPUT...(FAIL(OUT/$+≠$≠+$/=*OUT/≠THERE ARE NO ANSWERS TO (≠+1+2+3+≠).≠/)END,FCLSD)
INPUT...(FCLSD(OUT/$/=*OUT/≠THE TRUTH OR FALSEHOOD OF (≠+1+≠) IS UNKNOWN.≠/)END)
INPUT...(FOUND(OUT/$+≠$≠+$/=*OUTPUT/≠(≠+1+2+3+≠) HAS THESE ANSWERS.≠/)FND3,CLOSED)
INPUT...(FND3(OUT/$+≠$≠+$+≠$≠+$+≠$≠+$/)AC3,FND2)
INPUT...(FND2(OUT/$+≠$≠+$+≠$≠+$/)AC2,FND1)
INPUT...(FND1(OUT/$+≠$≠+$/)AC1,CLOSED)
INPUT...(AC3(+A1/2+4+6/=*OUT/≠(≠+1+2+3+4+5+6+7+≠)≠/)AC3,FND)
INPUT...(AC2(+A1/2+4/=*OUT/≠(≠+1+2+3+4+5+≠)≠/)AC2,END)
INPUT...(AC1(+A1/2/=*OUT/≠(≠+1+2+3+≠)≠/)AC1,END)
INPUT...(CLOSED(OUT/$/=*OUT/≠(≠+1+≠) IS TRUE.≠/)END)
INPUT...(ASSERT)
```

BEGIN INTERPRETING RULES.

```
INPUT...(INVERSE OF,INVERSE OF,INVERSE OF)
 (INVERSE OF,INVERSE OF,INVERSE OF)
INPUT...(INVERSE OF,DISJOINT FROM,DISJOINT FROM)
 (INVERSE OF,DISJOINT FROM,DISJOINT FROM)
INPUT...(INVERSE OF,IDENTICAL TO,IDENTICAL TO)
 (INVERSE OF,IDENTICAL TO,IDENTICAL TO)
INPUT...(CHAIN OF+AND,DISJOINT FROM,SUBSET OF,DISJOINT FROM)
 (CHAIN OF+AND,DISJOINT FROM,SUBSET OF,DISJOINT FROM)
INPUT...(CHAIN OF+AND,SUBSET OF,SUBSET OF,SUBSET OF)
 (CHAIN OF+AND,SUBSET OF,SUBSET OF,SUBSET OF)
INPUT...(CHAIN OF+AND,COMES FROM,COMES FROM,IN)
 (CHAIN OF+AND,COMES FROM,COMES FROM,IN)
INPUT...(CHAIN OF+AND,IN,IN,IN)
 (CHAIN OF+AND,IN,IN,IN)
INPUT...(INVERSE OF,PARENT OF,OFFSPRING OF)
 (INVERSE OF,PARENT OF,OFFSPRING OF)
INPUT...(INVERSE OF,SIBLING OF,SIBLING OF)
 (INVERSE OF,SIBLING OF,SIBLING OF)
INPUT...(INVERSE OF,MARRIED TO,MARRIED TO)
 (INVERSE OF,MARRIED TO,MARRIED TO)
INPUT...(INVERSE OF,HUSBAND OF,WIFE OF)
 (INVERSE OF,HUSBAND OF,WIFE OF)
INPUT...(INVERSE OF,COUSIN OF,COUSIN OF)
 (INVERSE OF,COUSIN OF,COUSIN OF)
INPUT...(CHAIN OF+AND,GRANDPARENT OF,PARENT OF,PARENT OF)
 (CHAIN OF+AND,GRANDPARENT OF,PARENT OF,PARENT OF)
INPUT...(CHAIN OF+AND,GRANDFATHER OF,FATHER OF,PARENT OF)
 (CHAIN OF+AND,GRANDFATHER OF,FATHER OF,PARENT OF)
INPUT...(CHAIN OF+AND,GRANDMOTHER OF,MOTHER OF,PARENT OF)
 (CHAIN OF+AND,GRANDMOTHER OF,MOTHER OF,PARENT OF)
INPUT...(CHAIN OF+AND,GRANDDAUGHTER OF,DAUGHTER OF,OFFSPRING OF)
 (CHAIN OF+AND,GRANDDAUGHTER OF,DAUGHTER OF,OFFSPRING OF)
INPUT...(CHAIN OF+AND,GRANDCHILD OF,CHILD OF,OFFSPRING OF)
 (CHAIN OF+AND,GRANDCHILD OF,CHILD OF,OFFSPRING OF)
INPUT...(CHAIN OF+AND,UNCLE OF,BROTHER OF,PARENT OF)
 (CHAIN OF+AND,UNCLE OF,BROTHER OF,PARENT OF)
INPUT...(CHAIN OF+AND,AUNT OF,SISTER OF,PARENT OF)
 (CHAIN OF+AND,AUNT OF,SISTER OF,PARENT OF)
```

```
INPUT...(CHAIN OF+AND,NIECE OF,DAUGHTER OF,SIBLING OF)
 (CHAIN OF+AND,NIECE OF,DAUGHTER OF,SIBLING OF)
INPUT...(CHAIN OF+AND,NEWPHEW OF,SON OF,SIBLING OF)
 (CHAIN OF+AND,NEWPHEW OF,SON OF,SIBLING OF)
INPUT...(CHAIN OF+AND,COUSIN OF,OFFSPRING OF,UNCLE OF)
 (CHAIN OF+AND,COUSIN OF,OFFSPRING OF,UNCLE OF)
INPUT...(CHAIN OF+AND,COUSIN OF,OFFSPRING OF,AUNT OF)
 (CHAIN OF+AND,COUSIN OF,OFFSPRING OF,AUNT OF)
INPUT...(CHAIN OF+AND,COUSIN OF,NIECE OF,PARENT OF)
 (CHAIN OF+AND,COUSIN OF,NIECE OF,PARENT OF)
INPUT...(CHAIN OF+AND,COUSIN OF,SIBLING OF,COUSIN OF)
 (CHAIN OF+AND,COUSIN OF,SIBLING OF,COUSIN OF)
INPUT...(CHAIN OF+AND,MOTHER IN LAW OF,MOTHER OF,MARRIED TO)
 (CHAIN OF+AND,MOTHER IN LAW OF,MOTHER OF,MARRIED TO)
INPUT...(CHAIN OF+AND,FATHER IN LAW OF,FATHER OF,MARRIED TO)
 (CHAIN OF+AND,FATHER IN LAW OF,FATHER OF,MARRIED TO)
INPUT...(CHAIN OF+AND,BROTHER IN LAW OF,HUSBAND OF,SIBLING OF)
 (CHAIN OF+AND,BROTHER IN LAW OF,HUSBAND OF,SIBLING OF)
INPUT...(CHAIN OF+AND,BROTHER IN LAW OF,BROTHER OF,MARRIED TO)
 (CHAIN OF+AND,BROTHER IN LAW OF,BROTHER OF,MARRIED TO)
INPUT...(CHAIN OF+AND,SISTER IN LAW OF,WIFE OF,SIBLING OF)
 (CHAIN OF+AND,SISTER IN LAW OF,WIFE OF,SIBLING OF)
INPUT...(CHAIN OF+AND,SISTER IN LAW OF,SISTER OF,MARRIED TO)
 (CHAIN OF+AND,SISTER IN LAW OF,SISTER OF,MARRIED TO)
INPUT...(CHAIN OF+AND,SON IN LAW OF,HUSBAND OF,DAUGHTER OF)
 (CHAIN OF+AND,SON IN LAW OF,HUSBAND OF,DAUGHTER OF)
INPUT...(CHAIN OF+AND,DAUGHTER IN LAW OF,WIFE OF,SON OF)
 (CHAIN OF+AND,DAUGHTER IN LAW OF,WIFE OF,SON OF)
INPUT...(UNION OF+AND,MARRIED,HUSBAND,WIFE)
 (UNION OF+AND,MARRIED,HUSBAND,WIFE)
INPUT...(UNION OF+AND,PARENT,FATHER,MOTHER)
 (UNION OF+AND,PARENT,FATHER,MOTHER)
INPUT...(UNION OF+AND,GRANDPARENT,GRANDFATHER,GRANDMOTHER)
 (UNION OF+AND,GRANDPARENT,GRANDFATHER,GRANDMOTHER)
INPUT...(UNION OF+AND,OFFSPRING,SON,DAUGHTER)
 (UNION OF+AND,OFFSPRING,SON,DAUGHTER)
INPUT...(UNION OF+AND,SIBLING,BROTHER,SISTER)
 (UNION OF+AND,SIBLING,BROTHER,SISTER)
INPUT...(UNION OF+AND,PERSON,MALE PERSON,FEMALE PERSON)
 (UNION OF+AND,PERSON,MALE PERSON,FEMALE PERSON)
```

```
INPUT...(UNION OF+AND,CHILD,BOY,GIRL)
 (UNION OF+AND,CHILD,BOY,GIRL)
INPUT...(INTERSECTION OF+AND,BOY,CHILD,MALE PERSON)
 (INTERSECTION OF+AND,BOY,CHILD,MALE PERSON)
INPUT...(INTERSECTION OF+AND,MALE PERSON,MALE,PERSON)
 (INTERSECTION OF+AND,MALE PERSON,MALE,PERSON)
INPUT...(LEFT INTERSECTION OF+AND,CHILD OF,OFFSPRING OF,CHILD)
 (LEFT INTERSECTION OF+AND,CHILD OF,OFFSPRING OF,CHILD)
INPUT...(LEFT INTERSECTION OF+AND,WIFE OF,MARRIED TO,WOMAN)
 (LEFT INTERSECTION OF+AND,WIFE OF,MARRIED TO,WOMAN)
INPUT...(LEFT INTERSECTION OF+AND,SON OF,OFFSPRING OF,MALE)
 (LEFT INTERSECTION OF+AND,SON OF,OFFSPRING OF,MALE)
INPUT...(LEFT INTERSECTION OF+AND,DAUGHTER OF,OFFSPRING OF,FEMALE)
 (LEFT INTERSECTION OF+AND,DAUGHTER OF,OFFSPRING OF,FEMALE)
INPUT...(LEFT INTERSECTION OF+AND,FATHER OF,PARENT OF,MAN)
 (LEFT INTERSECTION OF+AND,FATHER OF,PARENT OF,MAN)
INPUT...(LEFT INTERSECTION OF+AND,MOTHER OF,PARENT OF,WOMAN)
 (LEFT INTERSECTION OF+AND,MOTHER OF,PARENT OF,WOMAN)
INPUT...(LEFT INTERSECTION OF+AND,BROTHER OF,SIBLING OF,MALE)
 (LEFT INTERSECTION OF+AND,BROTHER OF,SIBLING OF,MALE)
INPUT...(LEFT INTERSECTION OF+AND,GRANDFATHER OF,GRANDPARENT OF,MAN)
 (LEFT INTERSECTION OF+AND,GRANDFATHER OF,GRANDPARENT OF,MAN)
INPUT...(LEFT INTERSECTION OF+AND,GRANDMOTHER OF,GRANDPARENT OF,WOMAN)
 (LEFT INTERSECTION OF+AND,GRANDMOTHER OF,GRANDPARENT OF,WOMAN)
INPUT...(LEFT HALF OF,FATHER,FATHER OF)
 (LEFT HALF OF,FATHER,FATHER OF)
INPUT...(LEFT HALF OF,MOTHER,MOTHER OF)
 (LEFT HALF OF,MOTHER,MOTHER OF)
INPUT...(LEFT HALF OF,SON,SON OF)
 (LEFT HALF OF,SON,SON OF)
INPUT...(LEFT HALF OF,DAUGHTER,DAUGHTER OF)
 (LEFT HALF OF,DAUGHTER,DAUGHTER OF)
INPUT...(LEFT HALF OF,PARENT,PARENT OF)
 (LEFT HALF OF,PARENT,PARENT OF)
INPUT...(LEFT HALF OF,CHILD,CHILD OF)
 (LEFT HALF OF.CHILD.CHILD OF)
INPUT...(LEFT HALF OF,COUSIN,COUSIN OF)
 (LEFT HALF OF,COUSIN,COUSIN OF)
INPUT...(LEFT HALF OF,BROTHER,BROTHER OF)
 (LEFT HALF OF,BROTHER,BROTHER OF)
```

```
INPUT...(LEFT HALF OF,SISTER,SISTER OF)
 (LEFT HALF OF,SISTER,SISTER OF)
INPUT...(LEFT HALF OF,HUSBAND,HUSBAND OF)
 (LEFT HALF OF,HUSBAND,HUSBAND OF)
INPUT...(LEFT HALF OF,WIFE,WIFE OF)
 (LEFT HALF OF,WIFE,WIFE OF)
INPUT...(LEFT HALF OF,SIBLING,SIBLING OF)
 (LEFT HALF OF,SIBLING,SIBLING OF)
INPUT...(LEFT HALF OF,GRANDFATHER,GRANDFATHER OF)
 (LEFT HALF OF,GRANDFATHER,GRANDFATHER OF)
INPUT...(LEFT HALF OF,GRANDDAUGHTER.GRANDDAUGHTER OF)
 (LEFT HALF OF,GRANDDAUGHTER,GRANDDAUGHTER OF)
INPUT...(LEFT HALF OF,GRANDMOTHER,GRANDMOTHER OF)
 (LEFT HALF OF,GRANDMOTHER,GRANDMOTHER OF)
INPUT...(LEFT HALF OF,GRANDSON,GRANDSON OF)
 (LEFT HALF OF,GRANDSON,GRANDSON OF)
INPUT...(LEFT HALF OF,MARRIED,MARRIED TO)
 (LEFT HALF OF,MARRIED,MARRIED TO)
INPUT...(LEFT HALF OF,OFFSPRING,OFFSPRING OF)
 (LEFT HALF OF,OFFSPRING,OFFSPRING OF)
INPUT...(LEFT HALF OF,MOTHER IN LAW,MOTHER IN LAW OF)
 (LEFT HALF OF,MOTHER IN LAW,MOTHER IN LAW OF)
INPUT...(LEFT HALF OF,FATHER IN LAW,FATHER IN LAW OF)
 (LEFT HALF OF,FATHER IN LAW,FATHER IN LAW OF)
INPUT...(LEFT HALF OF,BROTHER IN LAW,BROTHER IN LAW OF)
 (LEFT HALF OF,BROTHER IN LAW,BROTHER IN LAW OF)
INPUT...(LEFT HALF OF,SISTER IN LAW,SISTER IN LAW OF)
 (LEFT HALF OF,SISTER IN LAW,SISTER IN LAW OF)
INPUT...(LEFT HALF OF.SON IN LAW,SON IN LAW OF)
 (LEFT HALF OF,SON IN LAW,SON IN LAW OF)
INPUT...(LEFT HALF OF,DAUGHTER IN LAW,DAUGHTER IN LAW OF)
 (LEFT HALF OF,DAUGHTER IN LAW,DAUGHTER IN LAW OF)
INPUT...(RIGHT HALF OF,OFFSPRING,MOTHER OF)
 (RIGHT HALF OF,OFFSPRING,MOTHER OF)
INPUT...(RIGHT HALF OF,OFFSPRING,FATHER OF)
 (RIGHT HALF OF,OFFSPRING,FATHER OF)
INPUT...(SUBSET OF,FATHER,MAN)
 (SUBSET OF,FATHER,MAN)
INPUT...(SUBSET OF,MOTHER,WOMAN)
 (SUBSET OF,MOTHER,WOMAN)
```

```
INPUT...(SUBSET OF,HUSBAND,MAN)
 (SUBSET OF,HUSBAND,MAN)
INPUT...(SUBSET OF,WIFE,WOMAN)
 (SUBSET OF,WIFE,WOMAN)
INPUT...(SUBSET OF,GRANDPARENT,PARENT)
 (SUBSET OF,GRANDPARENT,PARENT)
INPUT...(SUBSET OF,PARENT,ADULT)
 (SUBSET OF,PARENT,ADULT)
INPUT...(SUBSET OF,CHILD,OFFSPRING)
 (SUBSET OF,CHILD,OFFSPRING)
INPUT...(SUBSET OF,CHILD,UNMARRIED)
 (SUBSET OF,CHILD,UNMARRIED)
INPUT...(SUBSET OF,MARRIED,PERSON)
 (SUBSET OF,MARRIED,PERSON)
INPUT...(SUBSET OF,CHILD,PERSON)
 (SUBSET OF,CHILD,PERSON)
INPUT...(SUBSET OF,MAN,PERSON)
 (SUBSET OF,MAN,PERSON)
INPUT...(SUBSET OF,WOMAN,PERSON)
 (SUBSET OF,WOMAN,PERSON)
INPUT...(DISJOINT FROM,PLANT,ANIMAL)
 (DISJOINT FROM,PLANT,ANIMAL)
INPUT...(DISJOINT FROM,PERSON,PLANT)
 (DISJOINT FROM,PERSON,PLANT)
INPUT...(DISJOINT FROM,UNMARRIED,MARRIED)
 (DISJOINT FROM,UNMARRIED,MARRIED)
INPUT...(DISJOINT FROM,MALE,FEMALE)
 (DISJOINT FROM,MALE,FEMALE)
INPUT...(DISJOINT FROM,CHILD,ADULT)
 (DISJOINT FROM,CHILD,ADULT)
INPUT...(ANSWER)
INPUT....

INPUT....


INPUT....

INPUT....    THE FOLLOWING LINES ARE QUESTIONS PUT TO THE ROUTINE ≠ANSWER≠
    THE FOLLOWING LINES ARE QUESTIONS PUT TO THE ROUTINE ≠ANSWER≠
INPUT....

INPUT....
```

```
INPUT....

INPUT....

INPUT....

INPUT....

INPUT...(CHAIN OF+AND,$,$,$)

(CHAIN OF+AND,$,$,$) HAS THESE ANSWERS.
(CHAIN OF+AND,DISJOINT FROM,SUBSET OF,DISJOINT FROM)
(CHAIN OF+AND,DAUGHTER IN LAW OF,WIFE OF,SON OF)
(CHAIN OF+AND,SON IN LAW OF,HUSBAND OF,SON OF)
(CHAIN OF+AND,SISTER IN LAW OF,SISTER OF,MARRIED TO)
(CHAIN OF+AND,SISTER IN LAW OF,WIFE OF,MARRIED TO)
(CHAIN OF+AND,BROTHER IN LAW OF,BROTHER OF,MARRIED TO)
(CHAIN OF+AND,BROTHER IN LAW OF,HUSBAND OF,MARRIED TO)
(CHAIN OF+AND,FATHER IN LAW OF,FATHER OF,MARRIED TO)
(CHAIN OF+AND,MOTHER IN LAW OF,MOTHER OF,MARRIED TO)
(CHAIN OF+AND,COUSIN OF,SIBLING OF,MARRIED TO)
(CHAIN OF+AND,COUSIN OF,NIECE OF,MARRIED TO)
(CHAIN OF+AND,COUSIN OF,OFFSPRING OF,MARRIED TO)
(CHAIN OF+AND,COUSIN OF,OFFSPRING OF,MARRIED TO)
(CHAIN OF+AND,NEWPHEW OF,SON OF,MARRIED TO)
(CHAIN OF+AND,NIECE OF,DAUGHTER OF,MARRIED TO)
(CHAIN OF+AND,AUNT OF,SISTER OF,MARRIED TO)
(CHAIN OF+AND,UNCLE OF,BROTHER OF,MARRIED TO)
(CHAIN OF+AND,GRANDCHILD OF,CHILD OF,MARRIED TO)
(CHAIN OF+AND,GRANDDAUGHTER OF,DAUGHTER OF,MARRIED TO)
(CHAIN OF+AND,GRANDMOTHER OF,MOTHER OF,MARRIED TO)
(CHAIN OF+AND,GRANDFATHER OF,FATHER OF,MARRIED TO)
(CHAIN OF+AND,GRANDPARENT OF,PARENT OF,MARRIED TO)
(CHAIN OF+AND,IN,IN,IN)
(CHAIN OF+AND,COMES FROM,COMES FROM,IN)
(CHAIN OF+AND,SUBSET OF,SUBSET OF,IN)
INPUT...(DISJOINT FROM,PLANT,ANIMAL)(BOY,ERIC)(INVERSE OF,$,$)($,CHILD,$)($,TREE,$))

(DISJOINT FROM,PLANT,ANIMAL) IS TRUE.
```

```
THE TRUTH OR FALSEHOOD OF (BOY,ERIC) IS UNKNOWN.

(INVERSE OF,$,$) HAS THESE ANSWERS.
(INVERSE OF,INVERSE OF,INVERSE OF)
(INVERSE OF,COUSIN OF,COUSIN OF)
(INVERSE OF,HUSBAND OF,WIFE OF)
(INVERSE OF,MARRIED TO,MARRIED TO)
(INVERSE OF,SIBLING OF,SIBLING OF)
(INVERSE OF,PARENT OF,OFFSPRING OF)
(INVERSE OF,IDENTICAL TO,IDENTICAL TO)
(INVERSE OF,DISJOINT FROM,DISJOINT FROM)

($,CHILD,$) HAS THESE ANSWERS.
(LEFT HALF OF,CHILD,CHILD OF)
(DISJOINT FROM,CHILD,ADULT)
(SUBSET OF,CHILD,PERSON)
(SUBSET OF,CHILD,UNMARRIED)
(SUBSET OF,CHILD,OFFSPRING)

THERE ARE NO ANSWERS TO ($,TREE,$).
```

PROGRAM 4

Here again, use of the associative memory is made to store information obtained through the parsing of English sentences into facts encoded as sentences of the Set Theoretic Language. The particular example given consists of sentences describing relationships in a particular family. As each sentence is broken down into STL form, the information is stored in the associative memory just as if the set of n-tuples formed had been input to the ASSERT program.

The STRAN rule set involved in this program only recognizes a few structural English words and hasn't the need to know the syntactic categories of all the words in a sentence. The result is that sentences can be parsed which contain words unknown to the parser and thus many diverse facts written in English can be stored in the associative memory without having to write STL n-tuples for any of them.

The parser recognizes proper names as those words having a preceeding asterisk. Each sentence is printed out with the resulting n-tuples following. Once an n-tuple has been stored, it is not printed out if encountered again.

```
BEGIN READING RULES.

INPUT...*  RULES FOR PARSE. STORES RESULTS IN ASSOCIATIVE MEMORY.  USES ASSERT.
INPUT...ECHO
INPUT...(PARSE(READ,BEGIN,STORE.PTEST))
INPUT...(PTEST(INPUT/$+≠)≠+$/)END.PARSE)
INPUT...(BEGIN(INPUT/$+≠.  ≠+$/=*OUTPUT/≠ ≠/*SENTENCE/1+≠.≠/INPUT/3/)B1,BEGUN)
INPUT...(BEGUN(INPUT/$+≠*≠*$+≠.≠+$/=*OUTPUT/≠ ≠/*SENTENCE/2+3+4/INPUT/5/)DBL,END)
INPUT...(B1(SENTENCE/≠ ≠+$/=SENTENCE/2/)B1,DBL)
INPUT...(DBL(SENTENCE/$+≠  ≠+$/=SENTENCE/1+≠ ≠+3/)DBL,RULE1)
INPUT...(RULE1(SENTENCE/$+≠ IS ≠+$+≠.≠/=NP/1+≠ ≠/VP/≠ ≠+3/)VPD1,ACTV)
INPUT...(VPD1(VP/DETS+$/=VP/≠ ≠+2/VPTYP/≠NOUN,≠/)RULE2,VPD3)
INPUT...(DETS(≠ A ≠ AN ≠))
INPUT...(VPD3(VP/≠ THE ≠+$/=VP/≠ ≠+2/VPTYP/≠FUNCTION,≠/)RULE2,VPD4)
INPUT...(VPD4(=VPTYP/≠ADJECTIVE,≠/)RULE2)
INPUT...(RULE2(NP/≠*HE ≠+$/SUBJ/$/=OUT/≠MALE,≠+3/TMP/≠S2,PREPS))≠/)TMP,RULE3)
INPUT...(RULE3(NP/≠*SHE ≠+$/SUBJ/$/=OUT/≠FEMALE,≠+3/TMP/≠S2,PREPS))≠/)TMP,RULE4)
INPUT...(RULE4(NP/≠*IT ≠+$/SUBJ/$/=OUT/≠NEUTER,≠+3/TMP/≠S2,PREPS))≠/)TMP,RULE5)
INPUT...(RULE5(NP/D1+$+≠ ≠/=OUT/≠NOUN,≠+2/SUBJ/2/TMP/≠S2,RSUBS))≠/)TMP,RULE6)
INPUT...(D1(≠*ANY ≠*EVERY ≠))
INPUT...(RULE6(NP/D2+$+≠ ≠/=OUT/≠ADJECTIVE,≠+2/SUBJ/2/TMP/≠S2,RSUBS))≠/)TMP,RULE7)
INPUT...(D2(≠*ANYTHING ≠*EVERYTHING ≠))
INPUT...(RULE7(NP/≠*NO ≠+$+≠ ≠/=OUT/≠NOUN,≠+2/SUBJ/2/TMP/≠S2,RDISJ))≠/)TMP,RULE8)
INPUT...(RULE8(NP/≠*NOTHING ≠+$+≠ ≠/=OUT/≠ADJECTIVE,≠+2/SUBJ/2/)RULE8A,RULE9)
INPUT...(RULE8A(=TMP/≠S2,RDISJ))≠/)TMP)
INPUT...(RULE9(NP/$+PREF+$+≠ ≠/=SUBJ/3/)PREPS,GSUBJ)
INPUT...(GSUBJ(NP/$+≠ ≠/=SUBJ/1/)PREPS)
INPUT...(RSUBS(SUBJ/$/VP/≠ ≠+$/=RSS/≠(SUBSET OF,≠+1+≠,≠+3+≠)≠/)RVPT)
INPUT...(RDISJ(SUBJ/$/VP/≠ ≠+$/=RSS/≠(DISJOINT FROM,≠+1+≠,≠+3+≠)≠/)RVPT)
INPUT...(PREPS(VP/$+PREPOSITNS+$/=RELTN/1+2/OBJ/3/)3T3,TEST)
INPUT...(PREPOSITNS(≠ OF ≠ IN ≠ TO ≠ THAN ≠ FROM ≠ AT ≠ FOR ≠ BETWEEN ≠))
INPUT...(2TUP(VP/≠ ≠+$+≠,≠+$/=VP/4/)2TA,2T1)
INPUT...(2T1(VP/≠ AND ≠+$/=VP/≠ ≠+2/)2T3,2T2)
INPUT...(2T2(VP/≠ ≠+$+≠ ≠+$/=VP/≠ ≠+4/)2TA,2T3)
INPUT...(2T3(VP/≠ ≠+$/SUBJ/$/=RSS/≠(≠+2+≠,≠+3+≠)≠/)RVPT)
INPUT...(2TA(SUBJ/$/=OUT/≠ADJECTIVE,≠+2/RSS/2+≠,≠+1/TMP/≠S2,2TB,S2,2TUP))≠/)TMP)
INPUT...(2TB(RSS/$/=OUT/1/)END)
INPUT...(SPSHL(≠S OF ≠S IN ≠S TO ≠S FROM ≠S AT ≠S FOR ≠))
INPUT...(ACTV(SENTENCE/$+≠ ≠+$+SPSHL+$+≠.≠/=NP/1+2/OUTPUT/3/VP/4+5/)A1,UNABLE)
INPUT...(A1(NP/$/OUTPUT/$+≠ ≠+$/=NP/1+2+≠ ≠/OUTPUT/4/)A1,A2)
```

```
INPUT...(A2(OUTPUT/$/VP/$/=VP/≠ ≠+1+2/VPTYP/≠VERB PHRASE,≠/)RULE2)
INPUT...(PREF(≠THE RELATION ≠THE PROPERTY ≠THE WORD ≠))
INPUT...(ALST(≠ AND THE PROPERTY ≠ AND THE RELATION ≠))
INPUT...(3T3(OBJ/PREF+$/=OBJ/2/)3T4)
INPUT...(3T4(OBJ/≠ITSELF≠+$/SUBJ/$/=OBJ/3+2/)3T5)
INPUT...(3T5(OBJ/$+≠ AND ITSELF≠/SUBJ/$/RELTN/$+≠ ≠/=OBJ/1+≠,≠+3/RELTN/4+≠+AND ≠/)3T6)
INPUT...(3T6(OBJ/$+ALST+$/RELTN/$+≠ ≠/=OBJ/1+≠,≠+3/RELTN/4+≠+AND ≠/)3T7)
INPUT...(3T7(OBJ/≠THE ≠+$/=OUT/≠UNIQUE,≠+2/OBJ/2/TMP/≠S2,3T8))≠/)TMP,3T8)
INPUT...(3T8(OBJ/≠*≠+$/=OUT/≠PROPER NAME,≠+1+2/TMP/≠S2,3TUP))≠/)TMP,3T9)
INPUT...(3T9(OBJ/$+PREPOSITNS+$/=OUTPUT/2/OBJ/1+≠,≠+3/)3TN,3TUP)
INPUT...(3TN(OUTPUT/≠ ≠+$/RELTN/$+≠ ≠/=RELTN/3+≠+≠+2/)3TUP)
INPUT...(3TUP(VPTYP/≠NOUN,≠/RELTN/≠ ≠+$+≠ ≠/=OUT/≠NOUN ROOT,≠+3/TMP/≠S2,3TF))≠/)TMP,3T1)
INPUT...(3T1(VPTYP/≠VERB≠+$/RELTN/≠ ≠+$+≠ ≠/=OUT/≠VERB ROOT,≠+4/TMP/≠S2,3TF))≠/)TMP,3T2)
INPUT...(3T2(VPTYP/≠FUNCTION,≠/RELTN/≠ ≠+$+≠ ≠/=OUT/1+3/TMP/≠S2,SUPER))≠/)TMP,3TF)
INPUT...(SUPER(RELTN/≠ ≠+$+≠ ≠/=OUT/≠NOUN ROOT,≠+2/TMP/≠S2,3TF))≠/)TMP,3TF)
INPUT...(3TF(RELTN/≠ ≠+$+≠ ≠/SUBJ/$/OBJ/$/=RSS/≠(≠+2+≠,≠+4+≠,≠+5+≠)≠/)PNAM)
INPUT...(TEST(SUBJ/≠*THE ≠+$/VP/≠ ≠+$/=SUBJ/4/VP/≠ ≠+2/VPTYP/≠FUNCTION,≠/)PREPS,2TUP)
INPUT...(RVPT(VPTYP/≠ADJ+$/VP/≠ ≠+$+≠ ≠+$/=OUT/≠ADJECTIVE PHRASE,≠+4+5+6/)RVPA,RVP1)
INPUT...(RVPA(=TMP/≠S2,PNAM))≠/)TMP)
INPUT...(RVP1(VPTYP/$/VP/≠ ≠+$/=OUT/1+3/TMP/≠S2,PNAM))≠/)TMP,PNAM)
INPUT...(PNAM(SUBJ/≠*≠+$/=OUT/≠PROPER NAME,≠+1+2/TMP/≠S2,PRINT))≠/)TMP,PRINT)
INPUT...(PRINT(RSS/$+≠(≠+$+≠)≠+$/=OUT/3/TMP/≠S5,BEGIN))≠/)TMP,BEGIN)
INPUT...(UNABLE(SENTENCE/$/=*OUTPUT/≠UNABLE TO PARSE ≠+1/)BEGIN)
INPUT...* RULES FOR ASSERT.
INPUT...(ASSERT(READ,STORE,ATST))
INPUT...(READ(*INPUT/≠.≠+$/=*INPUT/2/)READ,END)
INPUT...(ATST(INPUT/$+≠)≠+$/)END,ASSERT)
INPUT...(STORE(INPUT/$+≠(≠+$+≠)≠+$/=OUT/3/INPUT/5/)STOR,END)
INPUT...(STOR(S5,STORE))
INPUT...(S5(OUT/$+≠,≠+$+≠,≠+$+≠,≠+$+≠,≠+$/)END,S4)
INPUT...(S4(OUT/$+≠,≠+$+≠,≠+$+≠,≠+$/=+S/1+3+5+7/*OUT/≠ (≠+1+2+3+4+5+6+7+≠)≠/)END,S3)
INPUT...(S3(OUT/$+≠,≠+$+≠,≠+$/=+S/1+3+5/*OUT/≠ (≠+1+2+3+4+5+≠)≠/)END,S2)
INPUT...(S2(OUT/$+≠,≠+$/=+S/1+3/*OUT/≠ (≠+1+2+3+≠)≠/)END,S1)
INPUT...(S1(OUT/$/=OUT/1+≠))≠/)OUT) THIS RULE ALLOWS A GOTO ON A RULE NAME
INPUT...* RULES FOR ANSWER. USES RULES FROM ASSERT.
INPUT...(ANSWER(READ,FIND,NTST))
INPUT...(NTST(INPUT/$+≠)≠+$/)END,ANSWER)
INPUT...(FIND(INPUT/$+≠(≠+$+≠)≠+$/=*OUT/≠ ≠/OUT/3/INPUT/5/)FND,END)
INPUT...(FND(F5,FIND))
INPUT...(F5(OUT/$+≠,≠+$+≠,≠+$+≠,≠+$+≠,≠+$/)END,F4)
```

```
INPUT...(F4(OUT/$+≠,≠+$+≠,≠+$+≠,≠+$/)AN4,F3)
INPUT...(F3(OUT/$+≠,≠+$+≠,≠+$/)AN3,F2)
INPUT...(F2(OUT/$+≠,≠+$/)AN2,S1)
INPUT...(AN4(+F1/1+3+5+7/)FOUND,FAIL)
INPUT...(AN3(+F1/1+3+5/)FOUND,FAIL)
INPUT...(AN2(+F1/1+3/)FOUND,FAIL)
INPUT...(FAIL(OUT/$+≠$≠+$/=*OUT/≠THERE ARE NO ANSWERS TO (≠+1+2+3+≠).≠/)END,FCLSD)
INPUT...(FCLSD(OUT/$/=*OUT/≠THE TRUTH OR FALSEHOOD OF (≠+1+≠) IS UNKNOWN.≠/)END)
INPUT...(FOUND(OUT/$+≠$≠+$/=*OUTPUT/≠(≠+1+2+3+≠) HAS THESE ANSWERS.≠/)FND3,CLOSED)
INPUT...(FND3(OUT/$+≠$≠+$+≠$≠+$+≠$≠+$/)AC3,FND2)
INPUT...(FND2(OUT/$+≠$≠+$+≠$≠+$/)AC2,FND1)
INPUT...(FND1(OUT/$+≠$≠+$/)AC1,CLOSED)
INPUT...(AC3(+A1/2+4+6/=*OUT/≠(≠+1+2+3+4+5+6+7+≠)≠/)AC3,END)
INPUT...(AC2(+A1/2+4/=*OUT/≠(≠+1+2+3+4+5+≠)≠/)AC2,END)
INPUT...(AC1(+A1/2/=*OUT/≠(≠+1+2+3+≠)≠/)AC1,END)
INPUT...(CLOSED(OUT/$/=*OUT/≠(≠+1+≠) IS TRUE.≠/)END)
INPUT...(PARSE)
```

```
BEGIN INTERPRETING RULES.

INPUT.... ENGLISH LANGUAGE MODEL OF NORMAN≠S FAMILY.
 ENGLISH LANGUAGE MODEL OF NORMAN≠S FAMILY.
INPUT...*NORMAN IS THE FATHER OF *ERIC.

*NORMAN IS THE FATHER OF *ERIC.
 (PROPER NAME,*ERIC)
 (FUNCTION,FATHER OF)
 (NOUN ROOT,FATHER OF)
 (PROPER NAME,*NORMAN)
 (FATHER OF,*NORMAN,*ERIC)
INPUT...*MARVIN IS THE FATHER OF *NORMAN.

*MARVIN IS THE FATHER OF *NORMAN.
 (PROPER NAME,*MARVIN)
 (FATHER OF,*MARVIN,*NORMAN)
INPUT...*PAT IS A DAUGHTER OF *ALICE.

*PAT IS A DAUGHTER OF *ALICE.
 (PROPER NAME,*ALICE)
 (NOUN ROOT,DAUGHTER OF)
 (PROPER NAME,*PAT)
 (DAUGHTER OF,*PAT,*ALICE)
INPUT...*ERIC IS A LITTLE BOY.

*ERIC IS A LITTLE BOY.
 (ADJECTIVE,LITTLE)
 (LITTLE,*ERIC)
 (NOUN,BOY)
 (BOY,*ERIC)
INPUT...*BARTON IS A BROTHER OF *NORMAN.

*BARTON IS A BROTHER OF *NORMAN.
 (NOUN ROOT,BROTHER OF)
 (PROPER NAME,*BARTON)
 (BROTHER OF,*BARTON,*NORMAN)
INPUT...*MARVIN IS MARRIED TO *ALICE.  *NORMAN IS MARRIED TO *MADELAINE.

*MARVIN IS MARRIED TO *ALICE.
 (MARRIED TO,*MARVIN,*ALICE)
```

```
*NORMAN IS MARRIED TO *MADELAINE.
 (PROPER NAME,*MADELAINE)
 (MARRIED TO,*NORMAN,*MADELAINE)
INPUT...*BARTON IS MARRIED TO *MERLA.   *KEVIN IS A SON OF *BARTON.

*BARTON IS MARRIED TO *MERLA.
 (PROPER NAME,*MERLA)
 (MARRIED TO,*BARTON,*MERLA)

*KEVIN IS A SON OF *BARTON.
 (NOUN ROOT,SON OF)
 (PROPER NAME,*KEVIN)
 (SON OF,*KEVIN,*BARTON)
INPUT...*CHRISTOPHER IS A SON OF *NORMAN.  *HE IS A CHILD.

*CHRISTOPHER IS A SON OF *NORMAN.
 (PROPER NAME,*CHRISTOPHER)
 (SON OF,*CHRISTOPHER,*NORMAN)

*HE IS A CHILD.
 (MALE,*CHRISTOPHER)
 (NOUN,CHILD)
 (CHILD,*CHRISTOPHER)
INPUT...*MANDRES IS IN *FRANCE.  *MADELAINE COMES FROM *MANDRES.

*MANDRES IS IN *FRANCE.
 (PROPER NAME,*FRANCE)
 (PROPER NAME,*MANDRES)
 (IN,*MANDRES,*FRANCE)

*MADELAINE COMES FROM *MANDRES.
 (VERB ROOT,COMES FROM)
 (COMES FROM,*MADELAINE,*MANDRES)
INPUT...)))
```

PROGRAM 5

A final example of the use of the associative memory is presented here. Using the information stored previously by programs (3) and (4), program (5) attempts to make deductions leading to the production of n-tuples that are relevant to the overall model of family relationships but have not been introduced previously. The reader should examine the information presented in programs (3) and (4) before proceeding to this example. The program given here is split into sections, each dealing with a particular primitive (a discussion of primitives can be found in section 1.3.4.1) which has proven useful in defining the characteristics of other relations. Those presented here are CHAIN OF&AND,INVERSE OF,UNION OF&AND,LEFT HALF OF, RIGHT HALF OF, RIGHT INTERSECTION OF&AND and MINISET OF&AND. By using previously stored information it is shown that with the proper sequence of FIND and ACCESS operations, new information (in the form of n-tuples) which is relevant to the area of interest can easily be deduced.

Thus we have the idea whereby the inexperienced computer user can input simple command to COMS for performing certain desired computer operations. Using this information (the commands) COMS could be set up to properly deduce the correct instructions needed to accomplish the required tasks. For example, if the user was in doubt as to what partiular commands were available to carry out his desired tasks, instructions to COMS could result in the output of such information. As a second example, instructions for the generation of calls to the Fortran library could be made available so the user need not be concerned about the actual programming statements involved.

A important point to note about program (5) is that one must be careful in programming a STRAN rule set to perform deductions. Enough information may be present to allow the eventual deduction of incorrect information.

```
BEGIN READING RULES.

INPUT...* DEDUCTION RULES.  CALLED BY DEDUCE.
INPUT...(DEDUCE(CHAIN,INVERSE,UNION,LEHALF,RIHALF,RINT,MINISET))
INPUT...(CHAIN(+F1/≠CHAIN OF+AND≠+$+$+$/=*OUT/≠EXECUTING RULE OF CHAIN.≠/)RCH1,END)
INPUT...(RCH1(+A1/1+2+3/)RCH2,END)
INPUT...(RCH2(+F2/2+$+$/)RCH3.RCH1)
INPUT...(RCH3(+A2/4+5/)RCH4,RCH1)
INPUT...(RCH4(+F3/3+5+$/)RCH5,RCH3)
INPUT...(RCH5(+A3/6/=+S/1+4+6/*OUT/≠ (≠+1+≠,≠+4+≠,≠+6+≠)≠/)RCH5,RCH3)
INPUT...(INVERSE(+F1/≠INVERSE OF≠+$+$/=*OUT/≠EXECUTING RULE OF INVERSE.≠/)RIV1,END)
INPUT...(RIV1(+A1/3+4/)RIV2,END)
INPUT...(RIV2(+F2/3+$+$/)RIV3,RIV1)
INPUT...(RIV3(+A2/1+2/=+S/4+2+1/*OUT/≠ (≠+4+≠,≠+2+≠,≠+1+≠)≠/)RIV3,RIV1)
INPUT...(UNION(+F1/≠UNION OF+AND≠+$+$+$/=*OUT/≠EXECUTING RULE OF UNION.≠/)UN1,END)
INPUT...(UN1(+A1/1+2+3/)UN2,END)
INPUT...(UN2(=+S/≠SUBSET OF≠+2+1/*OUT/≠ (SUBSET OF,≠+2+≠,≠+1+≠)≠/)UN3)
INPUT...(UN3(=+S/≠SUBSET OF≠+3+1/*OUT/≠ (SUBSET OF,≠+3+≠,≠+1+≠)≠/)UN1)
INPUT...(LEHALF(+F1/≠LEFT HALF OF≠+$+$/=*OUT/≠EXECUTE LEFT HALF.≠/)LEH1,END)
INPUT...(LEH1(+A1/1+2/)LEH2,END)
INPUT...(LEH2(+F2/2+$+$/)LEH3,LEH1)
INPUT...(LEH3(+A2/3+4/=+S/1+3/*OUT/≠ (≠+1+≠,≠+3+≠)≠/)LEH3,LEH1)
INPUT...(RIHALF(+F1/≠RIGHT HALF OF≠+$+$/=*OUT/≠EXECUTING RIGHT HALF.≠/)RIH1,END)
INPUT...(RIH1(+A1/1+2/)RIH2,END)
INPUT...(RIH2(+F2/2+$+$/)RIH3,RIH1)
INPUT...(RIH3(+A2/3+4/=+S/1+4/*OUT/≠ (≠+1+≠,≠+4+≠)≠/)RIH3,RIH1)
INPUT...(RINT(+F1/≠RIGHT INTERSECTION OF+AND≠+$+$+$/=*OUT/≠EXECUTING RINT.≠/)RIN1,END)
INPUT...(RIN1(+A1/1+2+3/)RIN2,END)
INPUT...(RIN2(+F2/1+$+$/)RIN3,RIN5)
INPUT...(RIN3(+A2/4+5/=+S/3+5/*OUT/≠ (≠+3+≠,≠+5+≠)≠/)RIN4,RIN5)
INPUT...(RIN4(=+S/2+4+5/*OUT/≠ (≠+2+≠,≠+4+≠,≠+5+≠)≠/)RIN3)
INPUT...(RIN5(+F2/2+$+$/)RIN6,RIN1)
INPUT...(RIN6(+A2/4+5/)RIN7,RIN1)
INPUT...(RIN7(+F3/3+5/=+S/1+4+5/*OUT/≠ (≠+1+≠,≠+4+≠,≠+5+≠)≠/)RIN6)
INPUT...(MINISET(+F1/≠MINISET OF+AND≠+$+$+$/=*OUT/≠EXECUTING MINISET.≠/)MIN1,END)
INPUT...(MIN1(+A1/2+3+4/=+S/2+4/*OUT/≠ (≠+2+≠,≠+4+≠)≠/)MIN2,END)
INPUT...(MIN2(=+S/≠SUBSET OF≠+3+2/*OUT/≠(SUBSET OF,≠+3+≠,≠+2+≠)≠/)MIN1)
INPUT...(DEDUCE)
```

```
BEGIN INTERPRETING RULES.

EXECUTING RULE OF CHAIN.
 (DISJOINT FROM,WOMAN,PLANT)
 (DISJOINT FROM,MAN,PLANT)
 (DISJOINT FROM,CHILD,PLANT)
 (DISJOINT FROM,MARRIED,PLANT)
 (DISJOINT FROM,CHILD,MARRIED)
 (DISJOINT FROM,WIFE,PLANT)
 (DISJOINT FROM,HUSBAND,PLANT)
 (DISJOINT FROM,MOTHER,PLANT)
 (BROTHER IN LAW OF,*BARTON,*MADELAINE)
 (FATHER IN LAW OF,*MARVIN,*MADELAINE)
 (NEWPHEW OF,*KEVIN,*MERLA)
 (NEWPHEW OF,*CHRISTOPHER,*MADELAINE)
 (UNCLE OF,*BARTON,*MADELAINE)
 (GRANDFATHER OF,*MARVIN,*MADELAINE)
 (COMES FROM,*MADELAINE,*FRANCE)
EXECUTING RULE OF INVERSE.
 (INVERSE OF,WIFE OF,HUSBAND OF)
 (INVERSE OF,OFFSPRING OF,PARENT OF)
 (MARRIED TO,*ALICE,*MARVIN)
 (MARRIED TO,*MERLA,*BARTON)
 (MARRIED TO,*MADELAINE,*NORMAN)
 (DISJOINT FROM,ANIMAL,PLANT)
 (DISJOINT FROM,PLANT,MOTHER)
 (DISJOINT FROM,PLANT,HUSBAND)
 (DISJOINT FROM,PLANT,WIFE)
 (DISJOINT FROM,MARRIED,CHILD)
 (DISJOINT FROM,PLANT,MARRIED)
 (DISJOINT FROM,PLANT,CHILD)
 (DISJOINT FROM,PLANT,MAN)
 (DISJOINT FROM,PLANT,WOMAN)
 (DISJOINT FROM,ADULT,CHILD)
 (DISJOINT FROM,FEMALE,MALE)
 (DISJOINT FROM,MARRIED,UNMARRIED)
 (DISJOINT FROM,PLANT,PERSON)
EXECUTING RULE OF UNION.
 (SUBSET OF,HUSBAND,MARRIED)
 (SUBSET OF,WIFE,MARRIED)
```

```
(SUBSET OF,BOY,CHILD)
(SUBSET OF,CHILD,CHILD)
(SUBSET OF,MALE PERSON,PERSON)
(SUBSET OF,PERSON,PERSON)
(SUBSET OF,BROTHER,SIBLING)
(SUBSET OF,PERSON,SIBLING)
(SUBSET OF,SON,OFFSPRING)
(SUBSET OF,GRANDFATHER,GRANDPARENT)
(SUBSET OF,FATHER,PARENT)
(SUBSET OF,MOTHER,PARENT)
EXECUTE LEFT HALF.
(FATHER,*NORMAN)
(FATHER,*MARVIN)
(BROTHER IN LAW,*BARTON)
(FATHER IN LAW,*MARVIN)
(MARRIED,*MARVIN)
(MARRIED,*MADELAINE)
(MARRIED,*MERLA)
(MARRIED,*ALICE)
(MARRIED,*BARTON)
(MARRIED,*NORMAN)
(GRANDFATHER,*MARVIN)
(BROTHER,*BARTON)
(UNCLE,*BARTON)
(DAUGHTER,*PAT)
(SON,*KEVIN)
(SON,*CHRISTOPHER)
EXECUTING RIGHT HALF.
(OFFSPRING,*ERIC)
(OFFSPRING,*NORMAN)
```

PROGRAM 6

This final program shows calls made to four different programs located in the COMS library. Since this is accomplished by use of the evaluator, input strings (data to the STRAN program) are sent directly to the evaluator unless they are preceeded by a period in column one; in which case they are output as comments. The first two subroutines called are TEST1 and TEST2. These are used to show the varied type of arguments one may use in a call to a library program. The reader should be aware that the actual calling statement is not output by the interpreter until after the subroutine has completely been executed. Only the echoed input line (i.e. beginning with INPUT...) is seen before execution begins.

Subroutines TEST1 and TEST2 have been set up to simply output the information passed to them by the evaluator. This is to show the reader that argument transferrence is infact done correctly. All possible types of arguments have been used in the two calls. Note that the dollar sign character placed before a variable name does not pass the relative machine address of the variable as was shown in program (1). Instead, the address of a temporary location containing the value of the variable is passed, just as if the dollar sign had not been present. This overriding effect only occurs for variables in the argument list of a call statement made to the COMS library. (The reason for this is discussed in section 1.3.2.4). A listing of subroutines TEST1 and TEST2 is shown following the output to program (6).

A second example involving sorting routines is also provided. Here, the two subroutines BBSORT and SORT are called on to sort a group of numbers stored in array A. Again, the reader will note that the printing of the call statement is not done until the termination of execution.

The call to each subroutine passes only the number of numbers to be sorted (this is stored in the variable N). The numbers to be sorted are provided by the subroutine FRANDN which is called during execution of each of the sorting routines. This was purposely done to show how other routines may be loaded during the execution of COMS library routines. FRANDN generates N real numbers each having a value between 0 and 1 inclusive. Subroutine BBSORT performs a bubblesort of the numbers in array A and prints out the sorted result. Once this subroutine has been executed, a few elements of the array A are printed out to show correct argument transferrence back to the evaluator.

The same operations are carried out by subroutine SORT except that an interchange sort is done rather than a bubblesort. Listings of BBSORT and SORT are shown immediately following the listings of subroutines TEST1 and TEST2.

```
BEGIN READING RULES.

INPUT...(LIBCALL(READ.EVAL))
INPUT...(READ(*INPUT/≠.≠+$/=*INPUT/2/)READ,END)
INPUT...(EVAL(INPUT/$+≠;≠+$/=,OUTPUT/1/)PRINT,END)
INPUT...(PRINT(OUTPUT/$/=*OUT/1/)LIBCALL)
INPUT...(LIBCALL)
```

```
BEGIN INTERPRETING RULES.

INPUT...INTEGER ARR1(5);
INTEGER ARR1(5)
INPUT...REAL ARR2(2,2,2);
REAL ARR2(2,2,2)
INPUT...Y=2.3;
Y=2.30000000000E+00$
INPUT...I=2;
I=2$
INPUT...J=2;
J=2$
INPUT...K=2;
K=2$
INPUT...ARR1(1)=100;
ARR1(1)=100$
INPUT...ARR1(2)=200;
ARR1(2)=200$
INPUT...ARR1(3)=300;
ARR1(3)=300$
INPUT...ARR1(4)=ARR1(1);
ARR1(4)=100$
INPUT...ARR1(5)=500;
ARR1(5)=500$
INPUT...ARR2(1,1,1)=10.10;
ARR2(1,1,1)=1.0100000000 00E+01$
INPUT...ARR2(1,1,2)=20.20;
ARR2(1,1,2)=2.0200000000 00E+01$
INPUT...ARR2(1,2,1)=30.30;
ARR2(1,2,1)=3.0300000000 00E+01$
INPUT...ARR2(1,2,2)=40.40;
ARR2(1,2,2)=4.0400000000 00E+01$
INPUT...ARR2(2,1,1)=50.50;
ARR2(2,1,1)=5.0500000000 00E+01$
INPUT...ARR2(2,1,2)=60.60;
ARR2(2,1,2)=6.0600000000 00E+01$
INPUT...ARR2(2,2,1)=70.70;
ARR2(2,2,1)=7.0700000000 00E+01$
INPUT...ARR2(2,2,2)=80.80;
ARR2(2,2,2)=8.0800000000 00E+01$
```

```
INPUT...ARG1=2.87E-2;
ARG1=2.870000000000E-02$
INPUT...ARG2=3672.1;
ARG2=3.672100000000E+03$
INPUT...ARG3=ARG1;
ARG3=2.870000000000E-02$
INPUT...IARG4=225693;
IARG4=225693$
INPUT...CALL TEST1(ARG1,ARG2,ARG3,IARG4,273,Y*5.0/(SQRT(16.0)),ARG1*ARG2,ARR1);

SUBROUTINE TEST1 ENTERED

TO DEMONSTRATE CORRECT ARGUMENT TRANSFERRENCE, THIS SUBROUTINE WILL LIST ALL THE
ARGUMENT VALUES IT HAS RECEIVED FROM THE CALL

FIRST ARGUMENT=       .03

SECOND ARGUMENT=   3672.10

THIRD ARGUMENT= .0287

FOURTH ARGUMENT=    225693

FIFTH ARGUMENT=  273

SIXTH ARGUMENT= 2.88

SEVENTH ARGUMENT=  105.389270

THE EIGHTH ARGUMENT IS AN ARRAY NAME - THE ELEMENTS OF THIS ARRAY ARE AS FOLLOWS

 100
 200
 300
 100
 500
```

```
EXIT SUBROUTINE TEST1
CALL TEST1(ARG1,ARG2,ARG3,IARG4,273,Y*5.0/(SQRT(16.0)),ARG1*ARG2,ARR1)
INPUT...CALL TEST2(ARR2(1,2,1),ARR2(I,J,K),$ARG1,$ARR2(1,2,2),$ARR2(I,I,1),$ARR1):
SUBROUTINE TEST2 ENTERED

TO DEMONSTRATE CORRECT ARGUMENT TRANSFERRENCE, THIS SUBROUTINE WILL LIST ALL THE
ARGUMENT VALUES IT HAS RECEIVED FROM THE CALL

FIRST ARGUMENT= 30.30

SECOND ARGUMENT= 80.80

THIRD ARGUMENT= .0287

FOURTH ARGUMENT= 40.40

FIFTH ARGUMENT= 70.70

THE SIXTH ARGUMENT IS AN ARRAY NAME - THE ELEMENTS OF THIS ARRAY ARE AS FOLLOWS

 100
 200
 300
 100
 500

EXIT SUBROUTINE TEST2
CALL TEST2(ARR2(1,2,1),ARR2(I,J,K),$ARG1,$ARR2(1,2,2),$ARR2(I,I,1),$ARR1)
END OF FILE READ ON INPUT TAPE 5

               HADBOJB //// END OF LIST ////
```

```
BEGIN READING RULES.

INPUT...(LIBCALL(READ,EVAL))
INPUT...(READ(*INPUT/≠.≠+$/=*INPUT/2/)READ,END)
INPUT...(EVAL(INPUT/$+≠;≠+$/=,OUTPUT/1/)PRINT,END)
INPUT...(PRINT(OUTPUT/$/=*OUT/1/)LIBCALL)
INPUT...(LIBCALL)
```

```
BEGIN INTERPRETING RULES.

INPUT....

INPUT....

INPUT....   SORTING EXAMPLE USING ROUTINES IN THE COMS LIBRARY
   SORTING EXAMPLE USING ROUTINES IN THE COMS LIBRARY
INPUT....

INPUT....

INPUT....   THE NUMBERS TO BE SORTED ARE GENERATED BY A RANDOM NUMBER GENERATOR
   THE NUMBERS TO BE SORTED ARE GENERATED BY A RANDOM NUMBER GENERATOR
INPUT....   ROUTINE CALLED FROM BOTH -BBSORT- AND -SORT-
   ROUTINE CALLED FROM BOTH -BBSORT- AND -SORT-
INPUT....

INPUT....

INPUT....   ARRAY -A- WILL BE USED TO STORE THE NUMBERS BEFORE AND AFTER SORTING
   ARRAY -A- WILL BE USED TO STORE THE NUMBERS BEFORE AND AFTER SORTING
INPUT....

INPUT...REAL A(1001);
REAL A(1001)
INPUT....

INPUT....

INPUT....   -N- IS THE NUMBER OF NUMBERS TO BE SORTED
   -N- IS THE NUMBER OF NUMBERS TO BE SORTED
INPUT....

INPUT...N=25;
N=25$
INPUT....

INPUT....
```

```
INPUT....   CALL THE BUBBLESORT SUBROUTINE (FROM THE COMS LIBRARY) WHICH SORTS
   CALL THE BUBBLESORT SUBROUTINE (FROM THE COMS LIBRARY) WHICH SORTS
INPUT....   AND OUTPUTS THE NUMBERS
   AND OUTPUTS THE NUMBERS
INPUT....

INPUT...CALL BBSORT(A,N,-1);

THE SORTED NUMBERS ARE....
     .030   .054   .082   .099   .175   .301   .315   .322   .404   .493
     .507   .518   .586   .603   .688   .698   .748   .760   .795   .834
     .887   .907   .922   .959   .986
CALL BBSORT(A,N,-1)
INPUT....

INPUT....

INPUT....   THE VALUE OF A FEW ELEMENTS IS PRINTED TO SHOW THAT ARGUMENT
   THE VALUE OF A FEW ELEMENTS IS PRINTED TO SHOW THAT ARGUMENT
INPUT....   TRANSFERRENCE BACK TO THE EVALUATOR WAS SUCCESSFUL
   TRANSFERRENCE BACK TO THE EVALUATOR WAS SUCCESSFUL
INPUT....

INPUT...A(1);
2.9572833661 84E-02
INPUT....

INPUT...A(2);
5.3798700823 23E-02
INPUT....

INPUT...N=50;
N=50$
INPUT....

INPUT....

INPUT....   CALL THE INTERCHANGE SORT SUBROUTINE WHICH WILL GENERATE A NEW
   CALL THE INTERCHANGE SORT SUBROUTINE WHICH WILL GENERATE A NEW
```

```
INPUT....   SET OF NUMBERS TO BE SORTED
    SET OF NUMBERS TO BE SORTED
INPUT....

INPUT...CALL SORT(A,N);

THE SORTED NUMBERS ARE....
       .010   .116   .125   .125   .129   .129   .144   .149   .204   .310
       .310   .312   .327   .336   .339   .354   .368   .385   .392   .404
       .407   .436   .513   .535   .550   .572   .597   .599   .618   .650
       .653   .655   .658   .686   .688   .694   .713   .718   .756   .771
       .794   .820   .854   .899   .930   .958   .963   .966   .968   .969
CALL SORT(A,N)
INPUT....

INPUT...A(1);
9.878572120204E-03
INPUT....

INPUT...A(2);
1.156555548453E-01
INPUT....

INPUT...A(50);
9.688191751696E-01
```

```
                 SUBROUTINE TEST1(A,B,C,I,J,D,E,K)
000013           DIMENSION K(5)
           C
000013           WRITE(6,1)
000016     1     FORMAT(*0SUBROUTINE TEST1 ENTERED*//*0TO DEMONSTRATE CORRECT ARGUM
                1ENT TRANSFERRENCE, THIS SUBROUTINE WILL LIST ALL THE*/* ARGUMENT V
                1ALUES IT HAS RECEIVED FROM THE CALL*)
           C
000016           WRITE(6,2) A,B,C,I,J,D,E
000043     2     FORMAT(*0FIRST ARGUMENT=*,F10.2//*0SECOND ARGUMENT=*,F10.2//*0THIR
                1D ARGUMENT=*,F6.4//*0FOURTH ARGUMENT=*,I10//*0FIFTH ARGUMENT=*,
                1 I5//*0SIXTH ARGUMENT=*,F5.2//*0SEVENTH ARGUMENT=*,F12.6)
           C
000043           WRITE(6,3) (K(IK),IK=1,5)
000055     3     FORMAT(*0THE EIGHTH ARGUMENT IS AN ARRAY NAME - THE ELEMENTS OF TH
                1IS ARRAY ARE AS FOLLOWS*//5(I5/))
           C
000055           WRITE(6,4)
000061     4     FORMAT(1H0,*EXIT SUBROUTINE TEST1*)
           C
000061           RETURN
000062           END

UNUSED COMPILER SPACE
010200
```

```
                       SUBROUTINE TEST2(F,G,P,L,Q,M)
000011                 DIMENSION M(5)
              C
000011                 WRITE(6,1)
000014        1        FORMAT(*0SUBROUTINE TEST2 ENTERED*//*0TO DEMONSTRATE CORRECT ARGUM
                      1ENT TRANSFERRENCE. THIS SUBROUTINE WILL LIST ALL THE*/* ARGUMENT V
                      1ALUES IT HAS RECEIVED FROM THE CALL*)
              C
000014                 WRITE(6,2) F,G,P,L,Q
000035        2        FORMAT(*0FIRST ARGUMENT=*,F6.2//*0SECOND ARGUMENT=*,F6.2//*0THIRD
                      1ARGUMENT=*,F6.4//*0FOURTH ARGUMENT=*,F6.2//*0FIFTH ARGUMENT=*,
                      1 F6.2)
              C
000035                 WRITE(6,3) (M(IK),IK=1,5)
000055        3        FORMAT(*0THE SIXTH ARGUMENT IS AN ARRAY NAME - THE ELEMENTS OF THI
                      1S ARRAY ARE AS FOLLOWS*//5(I5/))
              C
000055                 WRITE(6,5)
000061        5        FORMAT(*0EXIT SUBROUTINE TEST2*)
              C
000061                 RETURN
000062                 END

UNUSED COMPILER SPACE
010200
```

```
                SUBROUTINE BBSORT (A,N,M)
          C
          C   THIS SUBROUTINE PERFORMS A BUBBLESORT ON A SET OF REAL NUMBERS
          C
000006          DIMENSION A(1001)
          C
          C  CALL THE RANDOM NUMBER GENERATOR
          C
000006          CALL FRANDN (A,N,0)
000007          M=-1
          C
          C  -1 FOR ASCENDING ORDER
          C
000012          I=1
          C
          C  FIRST ELEMENT OF ARRAY A
          C
000013       21 IF(A(I).GT.A(I+1)) GO TO 20
          C
          C  COMPARE FIRST ELEMENT WITH NEXT ONE IN ARRAY
          C
000020       24 I=I+1
          C
          C  IF THE SECOND IS GREATER THAN THE FIRST, INCREMENT AND GO ON
          C
000022          IF (I-N) 21,26,26
          C
          C  TEST FOR END OF THE ARRAY
          C
000024       20 X=A(I)
000026          A(I)= A(I+1)
000031          A(I+1)=X
          C
          C  INTERCHANGE ONE ELEMENT WITH ANOTHER
          C
000032          J=I
000033          GO TO 27
000034       23 IF (A(J).LT.A(J-1)) GO TO 22
```

```
         C
         C    START WORKING BACKWARDS FROM THIS POINT COMPARING EACH ELEMENT TO PREVIOUS
         C  ONE AND INTERCHANGE IF NECESSARY
         C
000040             GO TO 24
000041        22   Y=A(J)
000043             A(J)=A(J-1)
000046             A(J-1)=Y
000050             J=J-1
         C
         C   INTERCHANGE
         C
000052        27   IF (J.EQ.1) GO TO 24
         C
         C  TEST SO DONT GO BEYOND BEGINNING OF ARRAY WHEN WORKING BACKWORDS
         C
000054             GO TO 23
000055       25    WRITE(6,99)
000061       99    FORMAT(*0THE SORTED NUMBERS ARE....*)
         C
         C   WRITE OUT SORTED ARRAY
         C
000061             WRITE(6,101) (A(I), I=1,N)
000102       101   FORMAT (4X,10F6.3)
000102             RETURN
000103             END

UNUSED COMPILER SPACE
010200
```

```
                 SUBROUTINE SORT (A,N)
          C
          C   THIS SUBROUTINE PERFORMS AN EXCHANGE SORT ON A SET OF REAL NUMBERS
          C
000005           DIMENSION A(1001)
          C
          C  CALL THE RANDOM NUMBER GENERATOR
          C
000005           CALL FRANDN(A,N)
000006         1 I=1
          C
          C  INDEX OF FIRST ELEMENT
          C
000007           IFLAG=0
          C
          C  FLAG RESET
          C
000010         4 IF (A(I) .GT. A(I+1)) GO TO 3
          C
          C  COMPARISON
          C
000016         5 I=I+1
          C
          C  INCREMENT
          C
000020         2 IF (I-N) 4,6,6
          C
          C  TEST FOR END OF ARRAY
          C
000023         3 IFLAG=1
000024           X=A(I)
000026           A(I)=A(I+1)
000031           A(I+1)=X
          C
          C  INTERCHANGE
          C
000032           GO TO 5
000033         6 IF (IFLAG .NE. 0) GO TO 1
```

```
                C
                C   MEANS INTERCHANGE WAS MADE . MUST GO THROUGH ARRAY UNTIL IT IS NOT
                C
 000034               WRITE(6,99)
 000040         99    FORMAT(*0THE SORTED NUMBERS ARE....*)
                C
                C   WRITE OUT SORTED ARRAY
                C
 000040               WRITE (6,12) (A(I), I=1,N)
 000057           12  FORMAT (4X, 10F6.3)
 000057           14  CONTINUE
 000057               RETURN
 000060               END

 UNUSED COMPILER SPACE
 010300
```

APPENDIX D

STRAN Error Messages

ERROR MESSAGE	RESULTING ACTION
ERROR HAS OCCURRED IN INTERPRETATION OF ...	Interpretation of new rule name popped up from stack
VARIABLE NAMED ... IS NOT YET STORED	Interpretation of new rule name popped up from stack
ERROR IN EVALUATION OF ARITHMETIC EXPRESSION	Expression not evaluated - next section of originating rule body is interpreted
ERROR, ALGEBRAIC EXPRESSION CONTAINS MORE THAN 5 SUB-SCRIPTED VARIABLE NAMES	Excess variable names ig-nored - evaluator continues on
THE VARIABLE ... HAS BEEN ASSIGNED A VALUE ZERO	Does what it says - eval-uator continues on
ERROR IN NUMERIC STORAGE OR RETRIEVAL	Variable in question is either not stored or not retrieved - evaluator continues on
NUMERIC STORAGE HAS OVER-FLOWED	No further results from arithmetic expressions evaluated are stored by the evaluator - evaluator continues on
ERROR IN INDICES	Subscripted variable for which error occurred is ignored - evaluator con-tinues on
DICTIONARY FULL, EXECUTION TERMINATED	Interpreter is immediately halted

APPENDIX E

COMS REFERENCE MANUAL

The major program elements of COMS are:

1) The STRAN interpreter

2) The Evaluator

3) The Associative Memory

4) The Program Library

The original version of COMS was implemented in PL/1 for the IBM 360/65 computer. A second but incomplete implementation for the CDC 6600 was carried out at Colorado University and NCAR in 1970 using - FORTRAN IV - . This version was updated and reimplemented at McMaster University for the CDC 6400 under Scope 3.4 by MARC S. BADER as part of this M.Sc. project in 1972-73.

The present version does not have all of the features of the original version but this has not lessened any of COMS capabilities. The differences in the two versions are outlined in Chapter 1 of this report.

COMS is not written in ANSI FORTRAN (due to the use of such CDC FORTRAN statements as BUFFER IN and BUFFER OUT) and at present will not compile under the FORTRAN extended compiler (FTN) at McMaster due to the COMPASS[1] routine LOAD-IT which involves transferrence of subroutine arguments under control of the CDC RUN compiler.

Heavily commented routines of the original FORTRAN version were rigorously tested and found to be satisfactorily applicable to the present version. Lightly commented routines however, were found to be either in need of further updating (where this was true more comments were entered) or completely incomprehensible to this author due to the unavailability of the algorithms involved. Thus, these routines (mainly the ones dealing with the hash coded associative memory) were left alone. A list of updated and nonupdated routines can be found below.

Any sections of coding needing more detailed explanations than could be explicitly entered via comment statements have the message ***NOTE() preceding them which gives a number reference to a section in this appendix where further details are given.

Another FORTRAN version of COMS was prepared by this author for the specific use of debugging certain routines which the interested programmer may have trouble understanding. This program is called COMSTR and is available on punched cards. The output consists of the contents of var-

---

[1]COMPASS is the assembly language used in the CDC 6000 series computers.

iables and arrays during the actual execution of a typical COMS run. This may be compared to the DEBUG facility of FTN but has nowhere near the detail. For a complete execution breakdown of the COMS operation, the user is advised to first adapt COMS for FTN compilation (by rewriting the LOAD-IT compass routine) and then use the DEBUG facility of that compiler. The work for this was started by this author (as can be seen in the PRISM compass routines which are capable of running under either RUN or FTN) but time ran out before its completion.

UPDATED ROUTINES:

Stran, Load, Interp, Ibody, Ipatrn, Setdict, Initial, Push, Eval, Number, Floater, Fixer, Bugout, Getnum, Getchr, Execute, Rdcard, Wrcard, Locate, Index, Page, Pack, Unpack, Move, Prisms.

NON-UPDATED ROUTINES:

Find, Locsym, Cont, Lnbr, Lnbl, Id, Strind, St4ind, Npart, Nucell, Rcell, Indices, Alloc, Getnl.

ROUTINES ADDED:

Loadit

Function LOAD

Note (1)

The purpose of the function LOAD is to:

1) Read the rules, store them (packed) in the array STORE and store their lengths in the array LSTORE.

2) Return a value of 1, 2 or 3 to the calling program (subroutine STRAN)

   (1): restart the entire COMS program by entering subroutine STRAN at statement 100

   (2): stop execution of the entire program

   (3): call the interpreting subroutine INTERP to begin executing the rules

Note (2)

COM is an array of ten elements which is equivalenced to the variable BEGIN. This variable is the first of the common block RESRVD thus giving the following correspondence:

```
COM(1) ..... BEGIN
COM(2) ..... ECH
COM(3) ..... ECHOFF
          .
          .
          .
COM(10)..... READL
```

The contents of the variables BEGIN, ECH and so on are initialized in subroutine SETDICT.

Note (3)

The variable NAMLIM is initialized to 10 in the subroutine SETDICT. It applies to names of STRAN rules and names of variables used in these rules. If a name encountered is longer than ten characters (one 6000 series word), the characters after the tenth are ignored.

Note (4)

The first index of array STORE refers to the word numbers into which the rule has been packed. The second index corresponds to the index of LSTORE. Thus a rule and its length are referenced by the same number.

The array DICT serves as the dictionary of names of rules and variables. The position (i.e. index) in DICT assigned to a rule is then correspondingly given as the index of LSTORE and second index of STORE for that rule. To illustrate, examine the following STRAN rule:

(READ(INPUT/1/=OUTPUT/1/)END)bb......

The name of the rule (i.e. READ) is stored in the dictionary (a hash code is calculated for the position in the dictionary) as say DICT(57). Then the rest of the rule

(INPUT/1/=OUTPUT/1/)END)bb......

will be stored as

```
STORE(1,57)=(INPUT/1/=
STORE(2,57)=OUTPUT/1/)
```

```
STORE(3,57)=END)bbbbbb
              .
              .
              .
STORE(8,57)=bbbbbbbbbb
```

Thus LSTORE(57) will equal 74.

Note (5)

The following pseudo operator commands are available:

(RESTART) - restarts the entire COMS program

(DUMP) - used when an error condition has developed in COMS and a dump of the associative memory is required

(RETURN) - stops execution of the COMS program

(READLX) - sets the current rule name to END and initiates a NAMELIST read whereby a user may redefine the value of specified COMS variables (given in subroutine SETDICT) that were originally initialized by DATA statements during the compilation of COMS. On completion of the NAMELIST read, if the current command (i.e. the one just read in by NAMELIST) stored in the variable RNAME is still END, then the STRAN interpreter will continue in the rule reading mode. If the current command is not END then RNAME is re-examined for the interpretation of a new command

(ECHO) - causes an echo of each input card to be printed. Each line given by the echo command begins with the phrase

INPUT...

to distinguish it from the output lines of the interpreter

(NOECHO) - echoing is discontinued

(TRACE) - causes each rule currently being interpreted to be output in the form

INTERPRETING RULE...

and also causes the contents of each variable change during rule execution to be output in the form

VARIABLE (name) =

(NOTRACE) - turns off the (TRACE) command

(PUNCH) - causes punching of a card for each output line produced by a rule. That is, the output line produced by the (TRACE) command is also punched on cards (without the two words shown above)

(NOPUNCH) - punching is discontinued

The above mentioned pseudo operators can be more easily looked upon as a set of switches controlling certain operations of the interpreter. The initial or default settings (where applicable) are (NOTRACE), (ECHO) and (NOPUNCH).

Subroutine INTERP

Note (1)

The purpose of subroutine INTERP is to:

1) Obtain the current STRAN rule to be interpreted. This rule is identified by a rule name which has either been passed directly to this subroutine from the function LOAD or has been "popped up" from the pushdown stack of rule names.

   The current rule is unpacked into the array CHAR one character per word. All further references to the rule contents by other COMS routines are done using CHAR as the information source (CHAR is in common with these other routines).

2) Once the rule has been obtained (from the array STORE where it was originally placed by the function LOAD), the rule type is established as either a "push-down" rule or a string manipulation rule; the former containing only a list of rule names, the latter

a middle or "body" section.

(3) Place rule names on the push-down stack.

(4) Break down the "go-to" section of a rule to determine if a path exists for both the success and failure of the rule.

(5) Send the body of a rule (if one exists) to the function IBODY.

(6) Receive information (from the function IBODY) on how successfully a rule was interpreted by the rest of the COMS routines.

Note (2)

The variable BREAK is in the common block RESRVD with the variable PRNR immediately following it. BREAK and PRNR are initially set in subroutine SETDICT.

Note (3)

A STRAN rule can either succeed or fail depending on the outcome of the left hand side of the rule. If the left hand side fails (whether through a decomposition or an associative memory operation) then the right hand side of the rule (the part to the right of the equals sign) is not executed and control is passed to the second rule name in the go-to section (if only one rule name is present control is passed in any case). Otherwise control is passed to the first rule name. Keeping this in mind, one sees the following possibilities. Assuming RNAME (the present rule name) is not END, it can either be the first of the two go-to rule names (stored in TNAME) or the second of these (stored in FNAME) or it can be neither. This is illustrated below.

<u>CASE 1</u> Successful left hand side:

```
(RULE1(rule body)RULE2,RULE3)
   ↑               ↑     ↑
 RNAME           TNAME FNAME
```

RNAME is set to TNAME and control is passed to RNAME.

```
(RULE1(rule body)RULE1,RULE3)
```

Control is passed to the same rule for re-execution (RNAME doesn't change).

<u>CASE 2</u> Unsuccessful left hand side:

```
(RULE1(rule body)RULE2,RULE3)
```

RNAME is set to FNAME and control is passed to RNAME.

```
(RULE1(rule body)RULE2,RULE1)
```

Control is passed to the same rule for re-execution (RNAME doesn't change).

Function IBODY

Note (1)

The purpose of the function IBODY is to:

1) Examine the section of a STRAN rule between the second opening bracket of the entire rule (i.e. the opening bracket after the rule name) and the pair of terminating characters consisting of an oblique stroke / followed by a closing bracket.

2) Break this same section down into left and right sides and send these to the function IPATRN for further interpretation.

3) Return a value of 1, 2, 3 or 4 to the subroutine INTERP.

   (1): no errors have occurred

   (2): some operation on the left hand side (i.e. the side to the left of the equal sign) has failed

   (3): error has occurred in the storage of a variable

   (4): interpretation error has occurred

Note (2)

ISIDE=1 indicates we are on the right side of the equals sign in a rule body and are doing a composition operation (because the associative store operation [i.e. +S] is taken care of elsewhere in the coding). Thus one can assume the result of the composition will be placed in a variable not previously defined. This assumption may be false but will not upset anything since if the name is found by the function DEFINE to be already in the dictionary, it just redefines it, giving it the exact dictionary location it had before. This reasoning also holds true for IREAD = •TRUE• where a read input data operation takes place putting the result in a variable we assume has not previously been defined.

Note (3)

It should be understood by the reader that strings of characters (to be referenced by COMS variables) which either are read from the input file by the function IBODY or are formed by the function IPATRN as the result of a right hand side pattern operation are placed (packed ten characters per computer word) in the array STORE with the corresponding string length in array LSTORE.

When work is to be carried out on these strings, they are taken out of their packed form in STORE and placed in unpacked form (one character per computer word) in the array TEMP with their corresponding lengths in array LTEMP.

The array STORE (as noted in subroutine INTERP) is also used to hold the packed form of the STRAN rules when they are first read in by the function LOAD. When work is to be done on these, they are transferred in unpacked form to array CHAR with their corresponding lengths placed in array LCHAR.

Note (4)

There is a possibility of having a slash as a literal character being used in a composition operation as shown the following example

(RULE1(=COMP/1+'/'/)END)

Note (5)

If the function IPATRN returns a value of 1 (meaning no errors in pattern matching have occurred) the variable IOP becomes an indicator to the function IBODY telling it through the calculation of IOP=JOP+1 whether or not one of the three associative memory operators has been encountered.

Function IPATRN

Note (1)

The purpose of this function is to:

1) Examine the strings of pattern operators found between the two oblique strokes immediately following a variable name on either the left or right side of a rule body.

2) Check the syntax of these strings and perform the operations required through other COMS routines called by IPATRN.

3) Return a value of 1, 2, 3 or 4 to the function IBODY, each number having the same meaning as those returned by IBODY to the subroutine INTERP (see note (1) of subroutine INTERP).

The term "pattern operator" refers to the legal STRAN operators for pattern decomposition, composition or transformation. These include the following:

dollar sign	$
quote	'
letter(s)	A,B,C,...Z
number	0,1,2,...9
period	.
downward arrow	↓
equivalence L	≡L
equivalence R	≡R

A detailed description of the meaning of the above operators can be found in Appendix B.

Note (2)

If a plus sign is found the program must check whether or not it lies inside a literal string (i.e. between quotes). This checking is done beginning at statement label 35. If the plus sign is in a literal string, a further test is made for another plus sign. If one is not found, the end of pattern indicator is set.

Note (3)

At this point, if IPLUS=1, this indicates there were no characters between the two outer slashes; for example (R1(XYZ//)R2). This of course is an error and pattern matching is terminated with a return to function IBODY (via the statement GO TO 33). An error condition in this case is not flagged to IBODY but rather this section of the rule body is ignored and the next section is picked up to be processed.

Note (4)

The calculation of ICHAR gives a pointer to an element in the array IVAL. ICHAR will be a number from 1 to 63 inclusive, having a one to one correspondence with the 63 possible characters allowed under the current FORTRAN compiler (RUN) in use. For example if ICHAR contains the dollar sign character $ (i.e. left justified octal code 53), ICHAR will be equal to 43 (a summary of octal display codes for all the available FORTRAN characters can be found in the CDC RUN FORTRAN manual, Appendix A).

The array IVAL is arranged so that a lexical scan of pattern operators is accomplished by a reference to it. In other words the variable L (used in a group of computed go-to's) indicates to the program the section of coding which should be executed next depending on which pattern operator has been encountered.

The legality of this pattern operator is also examined and if an illegal operator is encountered (i.e. a character other than those mentioned in note (1) of function IPATRN), control is returned to the function IBODY which in turn gives control to subroutine INTERP. Here an error message is printed out and the next rule to be interpreted is picked up.

The array IVAL is initialized in subroutine SETDICT. An outline of its contents and corresponding character references is given below.

```
IVAL(1)= 1
    (2)= 1   L=1 =>  letter A→Z
    (3)= 1   (L is used as a computed go-to
      .      pointer indicating the particular
      .      pattern operator in use)
      .      (translate => as "indicating")
    (26)=1
```

```
IVAL(27)=2
    (28)=2
      .
      .        L=2   =>   number  0→9
      .
    (36)=2

    (37)=9
    (38)=9
      .
      .        L=9   =>    +,-,*,(,),/
      .
    (42)=9

    (43)=3     L=3   =>    $
    (44)=9     L=9   =>    =
    (45)=4     L=4   =>    blank
    (46)=9     L=9   =>    comma
    (47)=5     L=5   =>    period
    (48)=6     L=6   =>    ≡

    (49)=9
    (50)=9     L=9   =>    ],[,:
    (51)=9

    (52)=7     L=7   =>    quote

    (53)=9
      .
      .        L=9   =>    →,∧,∨,↑
      .
    (56)=9

    (57)=8     L=8   =>    ↓

    (58)=9
    (59)=9
      .
      .        L=9   =>    <,>,≤,≥,¬,;
      .
    (63)=9
```

<u>Note (5)</u>

The variable K is used as a pointer to the specific pattern matching operation to be performed. This in turn is determined by the two variables ISIDE and IOP. ISIDE determines whether we are dealing with the left side (ISIDE=0) or right side (ISIDE=1) of the rule body. IOP determines which associative memory operation is to be performed (if any). Both ISIDE and IOP are set in function IBODY. The following chart shows all the possible combinations of ISIDE and IOP with the resulting value of K.

ISIDE	IOP	K	OPERATION
0	0	1	Left side pattern match
0	1	2	Error (1)
0	2	3	Left side associative find
0	3	4	Left side associative access
1	0	5	Right side composition
1	1	6	Right side associative store
1	2	7	Error (2)
1	3	8	Error (3)

Error (1) IOP indicates an associative store operation but ISIDE indicates the left side of the rule body. Associative stores can only be done on the right side.

Error (2) & (3) IOP indicates associative access and find operations respectively, but ISIDE indicates the right side of the rule body. Associative finds and accesses can only be done on the left side.

K is used to determine the general pattern matching operation needed, while the variable L determines the particular operator within this operation that is presently in use. This is illustrated in the following example:

```
           K=1        K=2       K=3
        ┌───────┐ ┌───────┐ ┌───────┐
(R1(ABC/$+'D'+•5/+F1/1+$/=+S/2+3/)S,F)
        ↑ ↑    ↑       ↑ ↑     ↑ ↑
        | |    |       | |     | |
      L=3 |   L=5    L=2 |   L=2 |
        L=7            L=3     L=2
```

Note (6)

The following coding is broken up into sections, each dealing with particular decomposition or composition operators. At the end of each section of coding the program exits to one of four statement labels. These include 33, 76, 77 or 78. Statement 33 tests for the existence of any more operators in the string and if none are found the program returns to function IBODY . Statements 76, 77 and 78 perform "clean-up" operations (these are described in later notes).

Note (7)

The ↓n operator positions the output string pointer such that composition will continue at the column specified by the integer n.

Note (8)

For all associative memory operations (that is, FIND, ACCESS and STORE), the +F, +A or +S has already been picked up by the function IBODY. Also, IBODY picks up the pseudo register number following a +F or +A and stores it in the variable NTRACK. This means that the function IPATRN need only be concerned with the dollar signs, literals and pseudo register numbers found between the oblique strokes immediately following the +F, +A or +S.

Note (9)

From experimentation with the following section of code, it was found that the dot operation is not working the way the original COMS manual claims. The reason is probably due to a mistake in coding during the translation of the PL1 version of COMS to the Fortran version. At present the dot operator has the same effect as a dollar sign operator.

Note (10)

If the current decomposition operator is the last operator in the current pattern matching string (i.e. a terminating oblique stroke follows it) and if this operator is a dollar sign by itself (operators of more than one character begin with dollar signs - e.g. $'ABC' or $5), then the whole or remaining part of the string being operated on (this would depend on what decomposition operations were previously performed on this string) is moved into the current pseudo register WORK(1,NVARB) with its length placed in LWORK(NVARB).

Note (11)

Even though at this point the program knows no more operators exist; to be consistent with the rest of the program a test for more operators is again made at statement 33.

Note (12)

There are more decomposition operators to be examined and thus the result of the $ operator (i.e. just how many characters it matches) will not be known until the next operator in line is picked up and executed. In the meantime, the dollar sign character is stored in the current pseudo register to be replaced "next time around" by the actual character string it matches.

Note (13)

The pattern matching operator $ followed by a literal (i.e. a character string enclosed in quotes) will match consecutive occurrences of the literal in the input string. The four lines of coding starting at statement label 70 form a loop to accomplish this operation. The program exits from this loop when either of the following conditions occur:

1) All the characters in the input string have successfully been matched, as in the following example:

   /$+$'ABC'+$/

   operating on the input string ABCABCABC (the loop will be executed three times)

2) An exact occurrence of the required literal is not immediately found each time a match is attempted (i.e. each time through the loop), as in the following example:

   /$+$'ABC'+S/

   operating on the input strings

   DABC (failure during first loop)

   ABCDEF (failure during second loop)

   ABCABCABD (failure during third loop)

<u>Note (14)</u>

When a successful match is found the characters matched are moved into the next position (as determined by the variable J) in the current pseudo register (as determined by NVARB). Thus, /$+$'ABC'+$/ matching say ABCABC will result in

```
WORK(1,NVARB) = A
  "  (2,  "  ) = B
  "  (3,  "  ) = C
  "  (4,  "  ) = A
         .
         .
         .
       etc.
```

Note (15)

The variable NFND is initially set to a value one more than the length of the string being operated on (i.e. the string stored in TEMP). If a pattern match involving the literal collection is found, NFND is reset to the character position number where the match begins. For example if

```
TEMP(1) = T
    (2) = H
    (3) = E
    (4) = blank
    (5) = C
    (6) = A
    (7) = T
```

(8) = blank

. .

. .

. .

(80) .

and the literal collection pattern being used is 'DOG'CAT')), a match is found for the literal CAT at the fifth character position in TEMP and thus NFND is set to 5.

It is very important to understand that the search here is for the left most match that can be made in the input string. This is the reason the program must test every one of the literals in the collection pattern given - i.e. one of these might cause a match further to the left than a previous one. This is shown in the following example.

input string: CATDOGMOUSEHENCOWbb...

literal string: 'COW'MOUSE'CAT'))

The first element of the literal collection, namely COW, will cause a match immediately since the whole input string is searched each time around. However even though at this point the current pseudo register WORK(1,NVARB) is set to COW, the process does not stop because the rest of the collection must still be considered. A match will again be found for MOUSE but the end result is that WORK(1,NVARB) will contain CAT, it being the leftmost match.

Thus the statement

```
IF(IQ.EQ.O.OR.IQ.GE.NFND) GO TO 82
```

continues the matching process if either a match is not found or all the literals have not been considered. The only exit from this part of the program occurs when all literals have been considered.

At this point, if the value of NFND has not changed from its original setting, the program knows a match was not made and an error is signalled.

Note (16)

One must be extremely careful in programming STRAN pattern operations as the coding at this point shows. Here a test is made to see if every character in the input string has been accounted for in matching operations. This is done by considering the length of the input string (stored in LTEMP) and checking that the current pseudo register contains characters that match this string exactly to its end. For example if the operation /'ABC'/ is performed on the input string ABCbb......... where LTEMP = 80, an error would result as shown by the following code execution:

```
IST=LTEMP-LWORK(NVARB)+1                              (1)
NFND=IST-IWORK+L                                      (2)
IF(INDEX(TEMP(IST),LWORK(NVARB),
WORK(1,NVARB),LWORK(NVARB).NE.1)GO TO 602             (3)
```

In (1), IST=80-3+1=78. Thus in (3) the INDEX function will not be equal to 1 but instead will equal zero. The reason is that TEMP(78), TEMP(79) and TEMP(80) are being examined, rather than the expected TEMP(1), TEMP(2) and TEMP(3) which do in fact contain the characters A,B and C respectively.

If the same match was performed on the input string ABC where LTEMP=3 (i.e. this string was not read in but was formed by a previous composition operation) an error would not have resulted as IST in that case would equal 1.

Thus the point here is that unless the STRAN user knows exactly what string is being operated on by decomposition operators, an error (i.e. GO TO 602) could result. A successful match for the first input string (ABCbbb...) would be caused by an operator string such as /$+'ABC'+$/ where any doubts are taken care of by the two extra dollar signs. The above discussion also holds for the other opertors which transfer control to this section of coding after pattern matching has taken place.

Note (17)

If the $ operator was stored in the pseudo register used immediately before the current pseudo register all characters of the input string up to but not including the first character stored in the current pseudo register (there may be more of them) are placed in the previous pseudo register. For example an operation such as /$+'A'/ working on the input string CDA (where LTEMP=3) would first have p.r.(1) = "$" and p.r.(2) = "A". Execution of the code starting at statement 78 would then result in p.r.(1) = "CD" and p.r.(2) = "A".

Note (18)

NFND is a variable used to indicate where a match was found in the input string (i.e. what character position). If the previous pattern matching operator was not a dollar sign, this means an exact match of the current operator was made in the input string. If this is not so, an error is signalled.

Subroutine PUSH

Note (1)

The purpose of this subroutine is to store the rule names listed in a STRAN type 1 rule in the 100 element array PUSHDN. The first rule stored is placed in PUSHDN(1), the second in PUSHDN(2) and so on. The variable ITOP is used as the index to this array, and is incremented by one each time a new rule name is stored. "Popping-up" the next rule name to be used is carried out in subroutine INTERP, where ITOP is decremented by one each time a rule name is referenced.

Note (2)

Each time a comma is encountered, a test is made to see if any characters (other than the comma) have been picked up. Thus if ILST is equal to NXT one of the following conditions has occurred:

(1) a comma was found immediately preceeding the first terminating bracket as shown below.

(STK1(R1,R2,))

(2) two or more commas were found together as shown below.

(STK1(R1,R2,,R3))

(3) no characters at all were found between the second opening bracket and the first termin-

ating bracket as shown below.

(STK1())

In each of the above cases, ILST is decremented by one to allow the next character (remember the string is being searched backwards) to be examined (if one exists). For (3) an automatic return to subroutine - INTERP - is executed.

Note (3)

If only one rule name is encountered then ILST will be zero at this point. However if more than one name occurs, the very last (going backwards) is taken care of here. The reason for this is due to there being no comma at the end of the list (going backwards) but rather an opening bracket; thus a special section of coding is needed for this situation.

Subroutine EVAL

Note (1)

The purpose of subroutine EVAL is to:

(1) evaluate algebraic formulae using most of the arithmetic and built-in functions of Fortran

(2) allocate storage for subscripted and unsubscripted variables

(3) store values in and retrieve values from these variables

(4) pass the necessary information to the compass routine LOADIT for the generation of argument lists for the Fortran programs in the COMS library.

Subroutine EVAL operates on input character strings dealing with numeric data in a two pass fashion. Details of this operation are described in section 1.3.2.1 and the reader is referred here for background information. The input string is stored in array CHAR with the number of characters in the string stored in the variable N.

Explanation of the operation of the major sections of subroutine EVAL is given in the comments associated with the routine.

Subroutine NUMBER

Compass Routine LOADIT

Compass Routine PRISMS

Explanations for the major sections of coding in these routines are given in the comments of the program.

Subroutine BUGOUT

Note (1)

If the STRAN pseudo operator (DUMP) is encountered during the rule reading mode, an automatic call to subroutine BUGOUT occurs. The purpose of this subroutine is to print out the entire contents of the hash table and symbol table used by the associative memory routines.

Subroutine Getnum

Note (1)

Both this subroutine and subroutine GETCHR contain programming statements used in CDC Fortran; in particular, ENCODE and DECODE statements. These are comparable to BCD write/read statements but with no peripheral equipment invovled. Information is transferred from one area of storage to another under FORMAT specification.

Note (2)

The purpose of this subroutine is to examine storage locations (words) which contain the character codes of numbers originally read in under "A" formats (that is, left justified with blank fill), and output words containing the integer, real or octal representation of these numbers. For example the number 1, input in character code as

34555555555555555555

would be output in one of the following representations:

integer:

00000000000000000001

real:

17204000000000000000

octal:

00000000000000000034

With the above forms, a number may be used in various arithmetic calculations (character code 55=blank, 34=the number 1).

Note (3)

XMAT will contain the format specification needed to transfer the unpacked characters stored in the array BUFF to the two word array TEMP where they will be stored in packed form. Thus, consider the following example:

In function IPATRN a dollar sign operator has just been encountered followed by two digits. That is, say, $12. Subroutine GETNUM is called with the following results -

BUFF(1)=1= 34555555555555555555

BUFF(2)=2= 35555555555555555555

MCHR= number of characters = 2

KTYP= type required = say, integer

MX= 20-2

  = 18

XMAT= the string (18X,2A1)

TEMP(1)= 55555555555555555555

TEMP(2)= 55555555555555553435

XMAT= the string (I20)

B= 00000000000000000014

(recall the number stored in B is an octal number, i.e. $14_8 = 12_{10}$). Thus B is returned to function IPATRN in its proper form for arithmetic use.

Subroutine GETCHR

Note (1)

The purpose of this subroutine is to examine storage locations (words) which contain the integer, octal or real representation of a number and change the contents of these words to the character code representation of that number. In other words this subroutine provides the opposite operation to subroutine GETNUM.

Note (2)

The real, integer or octal representation of the number stored in NUMB is transferred in character code representation to the two words of array TEMP. For example if NUMB=8, i.e.

NUMB= 00000000000000000010

then the contents of TEMP(1) and TEMP(2) would be

TEMP(1)= 55555555555555555555

TEMP(2)= 55555555555555555543

(43 is the character code for the number 8).

The contents of TEMP(1) and TEMP(2) is then unpacked one character per word into the array BUFF. Thus,

BUFF(1)= 55000000000000000000

```
BUFF(2)= 5500000000000000000
  .                 .
  .                 .
  .                 .
BUFF(20)= 4300000000000000000
```

(for details of the unpacking operation see the notes on subroutine UNPACK).

Note (3)

The DO LOOP variable I counts the number of blanks encountered as each word of the array BUFF is examined. Subtracting I from 21 gives the number of digits present. This is stored in the variable NCHAR. The digits are then transferred from BUFF to the array CHAR to be passed back to the calling routine.

Subroutine WRCARD

Note (1)

If two arguments were used in the call to WRCARD, the second provides the means to determine exactly how many words of the array BUFF are to be printed. The integer divide (LENGTH+9)/10 gives the number of words. For example, in the function INDICES the following error message is set up in array MESAGE:

```
DATA MESAGE/17HERROR IN INDICES/
```

The call to WRCARD to print this is CALL WRCARD(MESAGE,17). Thus LENGTH=17 and (LENGTH+9)/10=2, so BUFF(1) and BUFF(2) are output. In this way, unnecessary use of core (usually to hold blank characters) is prevented.

Function LOCATE

Note (1)

The purpose of this function is to store and retrieve STRAN rule names using the array DICT. A maximum of 257 names may be stored since four of the 261 available elements of DICT are used to hold special information which describes the current contents of the dictionary. An attempt to store more than 257 names results in the output of an error message to this effect, followed by complete program termination (via a STOP statement).

The contents of DICT(1) is initialized in subroutine INITAL to 257. Also in this subroutine the elements DICT(2) through DICT(261) are initialized to zero.

Note (2)

The word containing the left justified character code of the rule name being stored or retrieved is left-circularly shifted by 23 bits. The new word formed is added to the original word and the lower 48 bits are masked off. The resulting value is divided by LIMIT and the remainder (which will always be between 0 and 256 inclusive) is placed in the variable LOC. This allows a hash calculation for one of the 257 available locations in the array DICT.

Note (3)

For storage of a rule name each location of DICT starting from LOC+5 is tested for availability. This is indicated by a zero value being present. When one is found the rule name is entered in that location, the dictionary contents (i.e. the first four elements of DICT) are updated, and the location where the rule name was stored is returned to the calling program.

For retrieval of a rule name the location where the variable was previously stored (i.e. LOC+5) is checked for a zero value. If a non zero value is found it is tested for equality with the word containing the rule name. A match causes the location of the rule name in DICT to be passed to the calling program. A non-matching word causes the next location (i.e. LOC=LOC+1) to be examined and this process continues until either the rule name is found or a zero word is detected. A zero word during the retrieval process indicates the name cannot be found in the dictionary and the function returns a zero value to the calling program.

Function INDEX

Note (1)

This is a general purpose function used to search a string of characters (stored consecutively in an array) for the presence of either a particular character or group of characters. The position in the string being searched of the first occurrence of the desired character(s) is returned as the value of the function. For example if the string ABCDEF is being searched for the character D, a value of 4 will be returned. Searching for the group of consecutive characters BCD will return a value of 2. If the character(s) required cannot be found at all, a value of zero is returned.

Note (2)

For efficiency sake, to save time, a limit to the number of searches that need be done is calculated. For example if the string in array A is ABCDEFGHIJ (N1=10) and that in B is EFGH (N2=4), only the first seven characters of the string in A need be examined.

Subroutine PAGE

Note (1)

Each time the program switches from the rule reading mode to the interpreting mode a new page of output headed with a title is printed. There is a problem involved here in that some of the output in either of the above two modes may take more than one page of printing (i.e. 60 lines), and thus when a new page is automatically begun by the line printer, no title will be given and the page number will not be incremented. The result is that the title may not be printed at the top of each new page and this causes the page counter present in this routine to be incorrect.

The problem could be solved by placing a line counting statement such as LCOUNT=LCOUNT+1 whereever the printing of a line of information occurs. Thus when the line counter reached sixty, the page counter could be incremented.

Subroutine PACK

Note (1)

Both subroutine PACK and UNPACK use non-ANSI Fortran coding to accomplish their tasks. The former uses the CDC ENCODE statement to pack up to a maximum of 80 characters stored one per word into 8 words having 10 characters each. The latter routine performs the reverse operation of unpacking 10 characters per word into a one character per word form by use of the CDC DECODE statement. In both cases an 80A1 format statement is used.

The above operations are needed when packed characters are to be examined singly and then repacked into their original form with any necessary changes made.

APPENDIX F

COMS FORTRAN PROGRAM LISTING

```
      PROGRAM COMS(INPUT,OUTPUT,PUNCH,TAPE5=INPUT,TAPE6=OUTPUT,TAPE7=
     1 PUNCH,DEBUG=OUTPUT)
      CALL STRAN
      CALL EXIT
      END
```

```
      SUBROUTINE STRAN
C STRAN IS THE INTERPRETER FOR THE STRAN LANGUAGE.
C     THIS SIMPLE CHARACTER MANIPULATION LANGUAGE WAS DEVISED BY
C     ROBERT C. GAMMILL AS PART OF A PHD THESIS AT MIT IN THE SPRING OF
C     1969.  THE FIRST IMPLEMENTATION WAS IN PL/I FOR THE IBM 360/65.
C     THIS SECOND IMPLEMENTATION (IN FORTRAN)  WAS CARRIED OUT AT
C     COLORADO UNIVERSITY AND NCAR DURING THE SPRING OF 1970.
C        ... THIS VERSION WORKS ON CDC 6000 SERIES COMPUTERS ...
C     THE GOALS OF THIS LANGUAGE ARE...
C        1. SIMPLICITY
C        2. MODULARITY
C        3. A MODICUM OF MACHINE INDEPENDENCE
C        4. TO SERVE AS A TEST-BED FOR SEMANTIC FACT RETRIEVAL
C     FOR EXAMPLE, AN ATTEMPT HAS BEEN MADE TO HANDLE ALL INPUT/OUTPUT
C     THROUGH TWO ROUTINES, RDCARD AND WRCARD.  THIS ALLOWS EASY
C     SWITCHING BETWEEN VARIOUS DEVICES, INCLUDING INTERACTIVE ONES.
C     THE FUNCTIONS LOCSYM,DEFSYM,STORE,FIND,ACCESS DEAL WITH FACT
C     RETRIEVAL, AS HASH CODED ASSOCIATIVE STORAGE FOR N-TUPLES OF
C     CHARACTER STRINGS.
C     YOU SHOULD NOTICE THAT THE LANGUAGE IS TOTALLY ORIENTED TOWARDS
C     CARD IMAGES (80 CHARACTER FIXED LENGTH STRINGS).
C FURTHER INFORMATION CAN BE OBTAINED FROM.......
C                 ASST. PROF. ROBERT C. GAMMILL
C                 DEPT. OF COMPUTER SCIENCE
C                 COLORADO UNIVERSITY
C                 BOULDER, COLORADO, 80302
C
C                       -OR-
C
C                 MARC S. BADER
C                 C/O APPLIED MATHEMATICS DEPARTMENT
C                 MCMASTER UNIVERSITY
C                 HAMILTON, ONTARIO
C
C
C   -STRAN- IS CALLED FROM THE MAIN PROGRAM ONLY
C
      COMMON/CNTROL/ECHO,TRACE,JUNK(11),INTAP,NOUTAP,JJJNK(4)
C
C
```

```
C     -INITAL- INITIALIZES ALL DICTIONARIES AND LIST STORAGE
C
100     CALL INITAL
C
C     ENTER RULE READING MODE.  PAGE PRODUCES A PAGE HEADING
C
101     CALL PAGE
        WRITE(NOUTAP,200)
200     FORMAT(* BEGIN READING RULES.*/)
C
C     LOAD RULES AND ACCEPT COMMANDS
C
        I=LOAD(XXX)
        GO TO (100,107,109),I
107     RETURN
C
C     ENTER RULE INTERPRETING MODE
C
109     CALL PAGE
        WRITE(NOUTAP,201)
        CALL INTERP
        GO TO 101
201     FORMAT(* BEGIN INTERPRETING RULES.*/)
        END
```

```
      FUNCTION LOAD(X)
C
C   -LOAD- IS CALLED FROM SUBROUTINE -STRAN- ONLY
C
C
C***NOTE(1)
C
      COMMON/CNTROL/ECHO,TRACE,JUNK(11),INTAP,NOUTAP,JJJNK(4)
      COMMON/STORG/DICT(261),WORK(80,9),STORE(8,257),PUSHDN(100),ITOP
     1,LWORK(9),LSTORE(257),RNAME,TNAME,FNAME,VNAME,BUFF(8),CHAR(80),
     2LCHAR,TEMP(80),LTEMP,NTUP(4),NST,EXTRA(80),NAMLIM,LTNLIM
      COMMON/RESRVD/BEGIN,ECH,ECHOFF,RETRN,DUMP,PRNTON,PRNTOFF,PNCH,
     1 NPNCH,READL,END,EQUAL,QUOTE,PLUS,PARENL,BREAK,PRNR,PARENR,COMMA,
     2 ATSIGN,PERIOD,PERCNT,AMPERS,DOLLAR,ASTRSK,SHARP,COMND(3)
      COMMON/SWTCHS/IVAL(63),LPNCH,IEVAL,IOUT,ISTOP,IREAD,ISIDE,NVARB
      INTEGER COM(10),RNAME,END,DEFINE
      LOGICAL ECHO,TRACE,LPNCH
C
C***NOTE(2)
C
      EQUIVALENCE (COM(1),BEGIN)
C
C   THIS IS THE MAIN LOOP TO COLLECT RULES, STORE THEM IN THE ARRAY
C   -STORE- (PACKED) AND THEIR LENGTHS IN -LSTORE-
C
C   READ AND UNPACK AN INPUT CARD (80 CHARACTERS) INTO ARRAY -CHAR-
C
1     CALL BCDARD(BUFF)
      CALL UNPACK(BUFF,CHAR)
C
C   DETERMINE IF THE CHARACTER STRING IN -CHAR- CONTAINS A LEFT
C   PARENTHESIS
C
      I = INDEX(CHAR,80,PARENL)
C
C   IF NOT CARD IS IGNORED
C
      IF(I.EQ.0) GO TO 1
      IBG = I + 1
C
```

```
C   CHECK FOR SECOND LEFT PARENTHESIS IN THE REMAINING CHARACTERS
C
      J = INDEX(CHAR(IBG),80-I,PARENL)
      IF(J.EQ.0) GO TO 2
C
C   OBTAIN THE LENGTH OF THE RULE NAME
C
C
***NOTE(3)
C
      NCHR=J-1
      IF(NCHR.GT.NAMLIM) NCHR=NAMLIM
C
C   PACK THE RULE NAME INTO THE VARIABLE -RNAME- AND ENTER IT IN THE
C   DICTIONARY
C
      CALL PACK(CHAR(IBG),RNAME,NCHR)
C
C   -DEFINE- IS AN ENTRY POINT IN THE FUNCTION -LOCATE-
C
      I = DEFINE(RNAME,DICT)
C
C   OBTAIN THE LENGTH OF THE REST OF THE RULE AND PACK IT INTO THE ARRAY
C   -STORE-
C
      LSTORE(I-4) = 81 - IBG - J
C
***NOTE(4)
C
      CALL PACK(CHAR(IBG+J),STORE(1,I-4),LSTORE(I-4))
      GO TO 1
C
C   CONTROL COMES HERE WHEN ONLY ONE LEFT PAREN HAS BEEN FOUND ON A CARD
C   WHILE IN RULE READING MODE.  THE CARD MUST CONTAIN A COMMAND PSEUDO
C   OPERATOR OR GO-TO (RULE NAME) SURROUNDED BY PARENTHESES
C
C   TEST FOR A RIGHT PARENTHESIS - IF NONE FOUND THE CARD IS IGNORED.
C   OBTAIN THE LENGTH OF RULE NAME AND CHECK LENGTH LIMIT
C
2     J = INDEX(CHAR(IBG),80-I,PARENR)
```

```
      IF(J.EQ.0) GO TO 1
      NCHR=J-1
      IF(NCHR.GT.NAMLIM) NCHR=NAMLIM
C
C   PACK THE RULE NAME INTO THE VARIABLE -RNAME-
C
      CALL PACK(CHAR(IBG),RNAME,NCHR)
C
C   CHECK TO SEE IF -RNAME- IS ANY PSEUDO OPERATOR (COMMAND)
C
C
***NOTE(5)
C
18    DO 24 I=1,10
      IF(COM(I).EQ.RNAME) GO TO(100,21,22,107,108,102,103,104,105,106),I
24    CONTINUE
      GO TO 199
21    ECHO=.TRUE.
      GO TO 1
22    ECHO=.FALSE.
      GO TO 1
C
C   REINITIALIZE THE WHOLE PROGRAM
C
100   LOAD=1
      RETURN
102   TRACE=.TRUE.
      GO TO 1
103   TRACE=.FALSE.
      GO TO 1
104   LPNCH=.TRUE.
      GO TO 1
105   LPNCH=.FALSE.
      GO TO 1
C
C   -RDLEX- IS AN ENTRY POINT IN THE SUBROUTINE -SETDICT-
C
106   RNAME=END
      CALL RDLEX
      IF(RNAME.NE.END) GO TO 18
```

```
      GO TO 1
C
C   STOP THE RUNNING OF THE PROGRAM
C
107   LOAD=2
      RETURN
108   CALL BUGOUT
      GO TO 1
C
C   -RNAME- WAS NOT A PSEUDO OPERATOR (COMMAND) IT MUST BE A GO-TO
C   RULE NAME
C
109   LOAD=3
      RETURN
      END
```

```
      SUBROUTINE INTERP
C
C    THIS SUBROUTINE IS CALLED FROM SUBROUTINE -STRAN- ONLY
C
C
***NOTE(1)
C
      COMMON/CNTROL/ECHO,TRACE,JUNK(11),INTAP,NOUTAP,JJUNK(4)
      COMMON/RLSRVD/BEGIN,LCH,ECHOFF,RETRN,DUMP,PRNION,PRNIOFF,PNCH,
     1 NPNCH,READL,END,EQUAL,QUOTE,PLUS,PARENL,BREAK,PRNR,PARENR,COMMA,
     2 ATSIGN,PERIOD,PERCNT,AMPERS,DOLLAR,ASTESK,SHARP,COMND(3)
      COMMON/STORG/DICT(251),WORK(80,9),STORE(8,257),PUSHDN(100),ITOP
     1,LWORK(9),LSTORE(257),RNAME,TNAME,FNAME,VNAME,BUFF(8),CHAR(80),
     2LCHAR,TEMP(80),LTEMP,NTUP(4),NST,EXTRA(80),NAMLIM,LINLIM
      INTEGER RNAME,PUSHDN,END,FNAME,TNAME
      LOGICAL TRACE
      GO TO 4
101   RETURN
C
C    OBTAIN THE NEXT RULE NAME FROM THE STACK AND DECREMENT THE NUMBER
C    REMAINING
C
3     IF(ITOP.EQ.0) GO TO 101
      RNAME=PUSHDN(ITOP)
      ITOP=ITOP-1
C
C    GET THE NEXT RULE TO BE INTERPRETED ... WHOSE NAME IS -RNAME-
C
4     IF(RNAME.EQ.END) GO TO 3
C
C    IF IT DOES NOT EXIST AN ERROR HAS OCCURRED
C
      I=LOCATE(RNAME,DICT)
      IF(I.EQ.0) GO TO 777
      IF(TRACE) WRITE(NOUTAP,202) RNAME
202   FORMAT(* INTERPRETING RULE...*A10)
C
C    PICK UP THE RULE LENGTH (I.E. THE NUMBER OF CHARACTERS) AND UNPACK
C    INTO THE ARRAY -CHAR-
C
      LCHAR = LSTORE(I-4)
      CALL UNPACK(STORE(1,I-4),CHAR,LCHAR)
C
C    SEARCH THE RULE FOR THE PAIR OF CONSECUTIVE CHARACTERS ≠/≠
C    THIS WOULD INDICATE THE RULE HAS A MIDDLE SECTION OR BODY
```

```
C
C
***NOTE(2)
C
      IBREAK = INDEX(CHAR,LCHAR,BREAK,2)
      IF(IBREAK.NE.0) GO TO 5
C
C    SEARCH THE RULE FOR THE PAIR OF CONSECUTIVE CHARACTERS ≠))≠
C    IF NEITHER OF THESE CHARACTER PAIRS ARE FOUND THEN AN ERROR HAS
C    OCCURRED
C
      I = INDEX(CHAR,LCHAR,PRNR,2)
      IF(I.EQ.0) GO TO 777
C
C    RULE IS A PUSH-DOWN RULE, CONTAINING A SEQUENCE OF RULE NAMES.
C    PLACE THE RULES ON THE PUSHDOWN STACK
C
      CALL PUSH(CHAR,I-1)
      GO TO 3
C
C    BEGIN INTERPRETING THE RULE
C
5     NBEG = IBREAK + 2
C
C    FIND THE RIGHT PAREN ENDING THE GO-TO SECTION OF THE RULE
C
      NEND = INDEX(CHAR(NBEG),LCHAR-NBEG+1,PARENR)
      IF(NEND.EQ.0) GO TO 777
      NCNT = NEND - 1
C
C    FIND THE COMMA WHICH MAY BE IN THE GO-TO SECTION
C
      NCOM = INDEX(CHAR(NBEG),NCNT,COMMA)
      IF(NCOM.NE.0) GO TO 6
C
C    NO COMMA FOUND, SO ONLY ONE GO-TO NAME.  SET -FNAME- AND -TNAME-
C    THE SAME
C
      IF(NCNT.GT.NAMLIM) NCNT = NAMLIM
C
C    PACK THE FIRST RULE NAME INTO -TNAME-
C
      CALL PACK(CHAR(NBEG),TNAME,NCNT)
      FNAME = TNAME
      GO TO 10
```

```
C
C    TWO NAMES PRESENT.  SET -TNAME- TO THE FIRST AND -FNAME- TO THE
C    SECOND
C
6      NCNT = NCOM - 1
       IF(NCNT.GT.NAMLIM) NCNT=NAMLIM
       CALL PACK(CHAR(NBEG),TNAME,NCNT)
       NBEG = NBEG + NCOM
       NCNT = NEND - NCOM - 1 $ IF(NCNT.GT.NAMLIM) NCNT=NAMLIM
C
C    PACK THE SECOND RULE NAME INTO -FNAME-
C
       CALL PACK(CHAR(NBEG),FNAME,NCNT)
C
C    BEGIN BREAKING DOWN THE BODY SECTION OF THE RULE
C
10     I = IBODY(IBREAK) $ GO TO (601,602,775,777),I
C
***NOTE(3)
C
C
C    THE FOLLOWING CODE DEFINES THE VARIOUS WAYS THAT INTERPRETATION OF A
C    RULE BODY MAY COME TO AN END
C
C    CONTROL COMES HERE IF NO ERRORS HAVE OCCURRED, AND ALL OPERATIONS ON
C    THE LEFT SIDE OF THE BODY WERE SUCCESSFULLY CARRIED OUT.
C    RNAME=END INDICATES THAT THE PRESENT RULE HAS MODIFIED ITSELF
C
601    IF(RNAME.NE.END.AND.RNAME.EQ.TNAME) GO TO 10
       RNAME = TNAME $ GO TO 4
C
C    CONTROL COMES HERE IF SOME OPERATION ON THE LEFT SIDE HAS FAILED
C
602    IF(RNAME.NE.END.AND.RNAME.EQ.FNAME) GO TO 10
       RNAME = FNAME $ GO TO 4
C
C    CONTROL COMES HERE WHEN AN ERROR OCCURS
C
775    WRITE(NOUTAP,776) VNAME $ GO TO 602
776    FORMAT(*0VARIABLE NAMED *A10* IS NOT YET STORED.*)
777    WRITE(NOUTAP,778) RNAME $ GO TO 3
778    FORMAT(*0ERROR HAS OCCURRED IN INTERPRETATION OF *A10)
       END
```

```
      FUNCTION IBODY(IBREAK)
C
C     THE FUNCTION -IBODY- IS CALLED FROM SUBROUTINE -INTERP- ONLY
C
C
***NOTE(1)
C
      COMMON/CNTROL/ECHO,TRACE,JUNK(11),INTAP,NOUTAP,JJUNK(4)
      COMMON/SWTCHS/IVAL(63),LPNCH,IEVAL,IOUT,ISTOP,IREAD,ISIDE,NVARB
      COMMON/RESRVD/BEGIN,ECH,ECHOFF,RETRN,DUMP,PRNTON,PRNTOFF,PNCH,
     1 NPNCH,READL,END,EQUAL,QUOTE,PLUS,PARENL,BREAK,PRNR,PARENR,COMMA,
     2 ATSIGN,PERIOD,PERCNT,AMPERS,DOLLAR,ASTRSK,SHARP,COMND(3)
      COMMON/STORG/DICT(261),WORK(80,9),STORE(8,257),PUSHDN(100),ITOP
     1,LWORK(9),LSTORE(257),RNAME,TNAME,FNAME,VNAME,BUFF(3),CPAR(80),
     2LCHAR,TEMP(80),LTEMP,NTUP(4),NST,EXTRA(80),NAMLIM,LINLIM
      INTEGER RNAME,VNAME,END,DEFINE,STORE,FIND,ACCESS
      INTEGER CHAR,COMND,EQUAL.ASTRSK,SHARP,AMPERS,QUOTE
      LOGICAL IOUT,IEVAL,IREAD,TRACE,LPNCH
C
C     INITIALIZE THE RULE BODY SEARCH
C     -NBEG- INDICATES THE PRESENT POSITION IN THE RULE BODY
C     -NEND- INDICATES THE POSITION OF THE END OF THE RULE BODY
C     ISIDE =0 INDICATES THE LEFT SIDE OF THE EQUALS SIGN AND ISIDE=1
C     INDICATES THE RIGHT SIDE OF THE EQUALS SIGN
C     -NVARB- KEEPS TRACK OF THE PSEUDO REGISTER NUMBER INTO WHICH WE ARE
C     PRESENTLY MOVING CHUNKS OF MATCHED STRINGS
C     -IOP- INDICATES WHAT KIND OF ASSOCIATIVE OPERATOR WE HAVE FOUND
C     IOP=0 ... NO ASSOCIATIVE OPERATOR FOUND
C     IOP=1 ... STORE OPERATOR
C     IOP=2 ... FIND OPERATOR
C     IOP=3 ... ACCESS OPERATOR
C
10    NBEG=1 $ NEND=IBREAK+1 $ ISIDE=0 $ NVARB=0 $ IOP=0
C
C     CONTROL RETURNS HERE WHEN WE ARE READY FOR THE NEXT SECTION OF THE
C     BODY
C
11    IF(NBEG.GE.NEND) GO TO 601
C
C     CHECK FOR THE EQUALS SIGN INDICATING THE BEGINNING OF THE RIGHT
C     SIDE OF THE BODY
C
      IF(CHAR(NBEG).NE.EQUAL) GO TO 110
      ISIDE = 1 $ NBEG = NBEG + 1
C
```

```
C    CHECK FOR / ENDING THE NEXT VARIABLE NAME OR ASSOCIATIVE OPERATOR
C
110     I = INDEX(CHAR(NBEG),NEND-NBEG,BREAK) $ IF(I.EQ.0) GO TO 777
        NCNT = I - 1
        NST=0 $ IOUT=.FALSE. $ IEVAL=.FALSE. $ NBG=NBEG $ IREAD=.FALSE.
C
C    CHECK FOR THE * (THE I-O OPERATOR) IN FRONT OF THE VARIABLE NAME
C
        IF(CHAR(NBG).NE.ASTRSK) GO TO 13
        IREAD=.TRUE. $ IF(ISIDE.EQ.1) IOUT=.TRUE.
12      NBG = NBG + 1 $ NCNT = NCNT - 1
C
C    CHECK FOR , (EVALUATION OPERATOR) IN FRONT OF NEXT VARIABLE NAME
C
13      IF(CHAR(NBG).NE.SHARP) GO TO 14
        IEVAL = .TRUE. $ GO TO 12
C
C    CHECK FOR + INDICATING ASSOCIATIVE OPERATOR
C
14      IF(CHAR(NBEG).EQ.AMPERS) GO TO 16
15      IF(NCNT.GT.NAMLIM) NCNT = NAMLIM
C
C    PACK THE VARIABLE NAME INTO -VNAME-
C
        CALL PACK(CHAR(NBG),VNAME,NCNT)
C
***NOTE(2)
C
        IF(ISIDE.EQ.1.OR.IREAD) GO TO 19
        LOC = LOCATE(VNAME,DICT) $ IF(LOC.EQ.0) GO TO 775 $ GO TO 20
16      DO 17 IOP=1,3
C
C    -COMND- IS AN ARRAY WHICH IS INITIALIZED IN SUBROUTINE -SETDICT-
C    COMND(1)=1HS, (2)=1HF, (3)=1HA
C
        IF(CHAR(NBEG+1).NE.COMND(IOP)) GO TO 17
C
C    OBTAIN THE PSEUDO REGISTER NUMBER WHICH FOLLOWS THE FIND OR
C    ACCESS ASSOCIATIVE MEMORY OPERATOR AND PUT IT IN -NTRACK-
C
        IF(IOP.GT.1) CALL GETNUM(NTRACK,CHAR(NBEG+2),1,1) $ GO TO 30
17      CONTINUE
C
C    INCORRECT FORM OF AN ASSOCIATIVE MEMORY CALL
C
```

```
      WRITE(NOUTAP,203) $ GO TO 602
203   FORMAT(* INCORRECT OP CODE IN ASSOC. MEMORY CALL.*)
C
C    LOCATE THE VARIABLE NAME IN THE DICTIONARY
C    IF THE VARIABLE CANNOT BE FOUND IT HAS OBVIOUSLY NOT BEEN STORED
C    YET IMPLYING AN ERROR CONDITION
C
C
C    ENTER THE VARIABLE NAME IN THE DICTIONARY
C    -DEFINE- IS AN ENTRY POINT IN THE FUNCTION -LOCATE-
C
19    LOC = DEFINE(VNAME,DICT)
      LTEMP = 0
      IF(ISIDE.EQ.1) GO TO 30
20    IF(.NOT.IREAD) GO TO 23
C
***NOTE(3)
C
C
C    READ INFORMATION (I.E. 80 COLUMNS) OFF NEXT INPUT CARD AND STORE IN
C    ARRAY -STORE-
C    THIS INFORMATION IS ASSOCIATED WITH A VARIABLE NAME THROUGH THE
C    SECOND INDEX OF THE ARRAY -STORE-
C
      CALL RDCARD(STORE(1,LOC-4))
      LSTORE(LOC-4) = 80
23    LTEMP = LSTORE(LOC-4)
C
C    UNPACK CONTENTS OF ARRAY -STORE- INTO THE ARRAY -TEMP-
C    -LTEMP- CONTAINS THE NUMBER OF CHARACTERS TO BE UNPACKED
C
      CALL UNPACK(STORE(1,LOC-4),TEMP,LTEMP)
      IF(TRACE) WRITE(NOUTAP,204) VNAME,(TEMP(J),J=1,LTEMP)
204   FORMAT(* VARIABLE *,A10,* = *,80A1)
C
C    -ISTART- IS THE CHARACTER STRING POSITION OF THE START OF THE
C    SECTION OF BODY TO BE SENT TO FUNCTION -IPATRN-
C
30    ISTART = NBEG + I
C
C    LOOK FOR THE / AT THE END OF THE PATTERN
C    IT IS FROM THIS / TO THE NEXT / THAT IS TERMED A SECTION OF THE
C    BODY. THIS SECTION WILL BE SENT TO FUNCTION -IPATRN- FOR FURTHER
C    EVALUATION
C    IF THE / IS NOT FOUND AN ERROR HAS OCCURRED
```

```
C
C
C    -L- IS THE LENGTH OF THE SECTION
C
       L = INDEX(CHAR(ISTART),NEND-ISTART,BREAK) $ IF(L.EQ.0) GO TO 777
C
***NOTE(4)
C
C
C    CHECK TO SEE IF THE / WE HAVE FOUND IS SURROUNDED BY QUOTES
C
31     IF(CHAR(ISTART+L-2).NE.QUOTE.OR.CHAR(ISTART+L).NE.QUOTE) GO TO 32
       I=INDEX(CHAR(ISTART+L),NEND-ISTART-L,BREAK) $ IF(I.EQ.0) GO TO 777
       L = L + I $ GO TO 31
C
C    SEND THE CURRENT SECTION OF BODY TO FUNCTION -IPATRN-
C
32     NBEG=ISTART+L $ I=IPATRN(IOP,ISTART,L) $ GO TO (599,602,775,777),I
C
C    PATTERN IS COMPLETED, GO TO APPROPRIATE SECTION OF CODING
C
C
***NOTE(5)
C
599    JOP = IOP + 1 $ GO TO (600,610,611,612),JOP
600    IF(ISIDE.EQ.0) GO TO 11
C
C    CALL THE EVALUATOR SENDING IT THE STRING OF CHARACTERS CONTAINED IN
C    ARRAY -TEMP-.  THE NUMBER OF CHARACTERS TO BE SENT ARE IN -LTEMP-
C
       IF(IEVAL) CALL EVAL(TEMP,LTEMP)
C
C    PACK THE CHARACTERS IN ARRAY -TEMP- (STORED ONE PER WORD) INTO
C    ARRAY -STORE- (WHERE THEY ARE STORED 10 PER WORD)
C
       CALL PACK(TEMP,STORE(1,LOC-4),LTEMP) $ LSTORE(LOC-4) = LTEMP
       IF(VNAME.EQ.RNAME) RNAME = END
C
C    TEST IF OUTPUT OF RIGHT HAND SIDE (I.E. RIGHT HAND SIDE OF EQUAL
C    SIGN) VARIABLE CONTENTS IS REQUIRED
C
       IF(TRACE) WRITE(NOUTAP,204) VNAME,(TEMP(J),J=1,LTEMP)
       IF(.NOT.IOUT) GO TO 11
C
C    IS PUNCHING OF THE SAME REQUIRED
```

```
C
      IF(LPNCH) PUNCH 206,(TEMP(J),J=1,LTEMP)
206   FORMAT(80A1)
C
C    WRITE OUT THE CONTENTS OF THE RIGHT HAND-SIDE-VARIABLE------
C
      CALL WRCARD(STORE(1,LOC-4),LTEMP) $ GO TO 11
610   IF(STORF(NST,NTUP,NTRACK)) 613,601,613
611   IF(FIND(NST,NTUP,NTRACK)) 613,602,613
612   IF(ACCESS(NST,NTUP,NTRACK)) 613,602,613
613   IOP = 0 $ GO TO 11
601   IBODY=1 $ RETURN
602   IBODY=2 $ RETURN
775   IBODY=3 $ RETURN
777   IBODY=4 $ RETURN
      END
```

```
      FUNCTION IPATRN(IOP,ISTART,L)
C
C   THE FUNCTION -IPATRN- IS CALLED FROM SUBROUTINE -INTERP- ONLY
C
C
***NOTE(1)
C
      COMMON/SWTCHS/IVAL(63),LPNCH,IEVAL,IOUT,ISTOP,IREAD,ISIDE,NVARB
      COMMON/RESRVD/BEGIN,ECH,ECHOFF,RETRN,DUMP,PRNTON,PRNTOFF,PNCH,
     1 NPNCH,READL,END,EQUAL,QUOTE,PLUS,PARENL,BREAK,PRNR,PARENR,COMMA,
     2 ATSIGN,PERIOD,PERCNT,AMPERS,DOLLAR,ASTRSK,SHARP,COMND(3)
      COMMON/STORG/DICT(261),WORK(80,9),STORE(8,257),PUSHDN(100),TTOP
     1,LWORK(9),LSTORE(257),RNAME,TNAME,FNAME,VNAME,BUFF(8),CHAR(80),
     2LCHAR,TEMP(80),LTEMP,NTUP(4),NST,EXTRA(80),NAMLIM,LINLTM
      INTEGER DEFSYM,TEMP,WORK,DOLLAR,RNAME,END,CHAR,QUOTE
C
C   -ISTOP- IS A FLAG TELLING WHEN THE PATTERN OPERATIONS ARE THROUGH
C
      LOGICAL ISTOP
C
      IPATRN=1
C
C   INITIALIZE FOR PATTERN LOOP
C
C
C   -IEND- INDICATES THE POSITION IN THE CHARACTER STRING LOCATED IN
C   ARRAY -CHAR- OF THE ENDING SLASH FOR THE PRESENT SECTION OF
C   OPERATORS BEING PROCESSED BY -IPATRN-
C
C   -IDRIVE- INDICATES THE FIRST POSITION OF A PATTERN OPERATOR IN
C   THE CHARACTER STRING SENT TO -IPATRN-
C
C   -NBEG- INDICATES THE POSITION IN -CHAR- AFTER THE ENDING SLASH
C   FOR THE PRESENT PATTERN OPERATIONS
C
C   -NVB- INDICATES THE NUMBER OF THE CURRENT PSEUDO REGISTER IN USE
C
32    IEND=ISTART+L-1 $ IDRIVE=ISTART $ NBEG=ISTART+L
      ISTOP=.FALSE. $ IWORK=1 $ NVB=NVARB+1
C
C   THIS IS THE PATTERN LOOP.  EVERY TIME WE ARE READY FOR ANOTHER CHUNK
C   OF THE PATTERN WE RETURN HERE
C
C   LOOK FOR THE FIRST PLUS SIGN IN THE OPERATOR STRING
C
```

```
34    IPLUS = INDEX(CHAR(IDRIVE),IEND-IDRIVE,PLUS)
C
***NOTE(2)
C
      IF(IPLUS.NE.0) GO TO 35
C
C   NO + SIGN CAN BE FOUND AFTER THE NEXT CHUNK OF PATTERN. THIS
C   INDICATES THE END OF THE PATTERN
C
C
C   SET -IPLUS- TO ONE MORE THAN NUMBER OF CHARACTERS BETWEEN SLASHES
C
      IPLUS = IEND - IDRIVE + 1
C
C   SET END OF PATTERN INDICATOR
C
      ISTOP = .TRUE.
C
***NOTE(3)
C
      IF(IPLUS.EQ.1) GO TO 33
      GO TO 36
C
C   CHECK FOR QUOTE AT BEGINNING OF STRING
C
35    IF(CHAR(IDRIVE).NE.QUOTE) GO TO 36
C
C   LOOK FOR SECOND QUOTE
C
      IQUOTE = INDEX(CHAR(IDRIVE+1),IEND-IDRIVE-1,QUOTE)+2
C
C   TEST IF PLUS SIGN IS INBETWEEN THE TWO QUOTES (E.G. /≠A+B≠+$/)
C
      IF(IPLUS.GE.IQUOTE) GO TO 36
      IPLUS = IQUOTE
C
C   CHECK FOR NO FOLLOWING PLUS SIGNS AFTER THE ONE FOUND INBETWEEN
C   QUOTES (E.G. /≠A+B≠/)
C
      IF(IPLUS+IDRIVE-1.EQ.IEND) ISTOP = .TRUE.
C
C   -JSTRT- INDICATES THE POSITION OF THE FIRST PATTERN OPERATOR IN
C   THE CURRENT SECTION OF PATTERN BEING EXAMINED
C
36    JSTRT = IDRIVE
```

```
C
C    -JSTOP- INDICATES THE POSITION OF THE LAST PATTERN OPERATOR
C
        JSTOP = IPLUS + IDRIVE - 1
C
C    -IDRIVE- INDICATES THE POSITION OF THE NEXT PATTERN OPERATOR IN THE
C    CURRENT SECTION OF PATTERN
C
        IDRIVE = IDRIVE + IPLUS
C
C    -JCHAR- CONTAINS FIRST PATTERN OPERATOR OR FIRST CHARACTER OF FIRST
C    PATTERN OPERATOR
C
        JCHAR = CHAR(JSTRT)
C
***NOTE(4)
C
        ICHAR = LCS(JCHAR,6).AND.77B $ L = IVAL(ICHAR)
C
***NOTE(5)
C
        K = LLS(ISIDE,2) + IOP + 1
        GO TO (361,777,362,363,364,365,777,777),K
333     IDRIVE = IDRIVE + 1 $ GO TO 36
C
C    LEFT SIDE PATTERN MATCHING OPERATION
C
361     GO TO (81,777,67,333,66,777,75,73,777),L
C
C    LEFT SIDE, ASSOCIATIVE FIND OPERATION
C
362     GO TO (777,63,61,333,777,777,62,777,777),L
C
C    LEFT SIDE, ASSOCIATIVE ACCESS OPERATION
C
363     GO TO (777,60,777,333,777,777,777,777,777),L
C
C    RIGHT SIDE, COMPOSITION OPERATION
C
364     GO TO (777,51,777,333,777,41,37,38,777),L
C
C    RIGHT SIDE, ASSOCIATIVE STORE OPERATION
C
365     GO TO (777,50,777,333,777,777,369,777,777),L
C
```

```
C     PROCESS RIGHT SIDE COMPOSITION STRINGS
C
C
***NOTE(6)
C
C     PROCESS QUOTED STRINGS FOR ASSOCIATIVE STORE OPERATION
C
C     -NST- KEEPS TRACK OF THE PARTICULAR ELEMENT OF THE CURRENT N-TUPLE
C     BEING CONSIDERED (THAT IS, -NST- CAN EQUAL 1, 2, 3 OR 4)
C
C
C     DEFINE THE STRING AS PART OF AN N-TUPLE IN THE ASSOCIATIVE MEMORY
C
369     NST=NST+1 $ NTUP(NST)=DEFSYM(CHAR(JSTRT+1),IPLUS-3) $ GO TO 33
C
C     PROCESS QUOTED STRING COMPOSITION OPERATOR
C
C     -NCHR- CONTAINS THE LENGTH OF THE QUOTED STRING
C
C     CALCULATION OF -NDIFF- ENSURES NO MORE THAN 80 CHARACTERS ARE COMPOSED
C     INTO ONE STRING.  -LINLIM- IS INITIALIZED IN SUBROUTINE -SETDICT-
C
37      NCHR=IPLUS-3 $ NDIFF=LINLIM-LTEMP
        IF(NCHR.GT.NDIFF) NCHR = NDIFF
C
C     MOVE THE REQUIRED CHARACTERS FROM THE OPERATOR STRING TO THE STRING
C     BEING COMPOSED AND RESET -LTEMP- TO INDICATE THE CURRENT NUMBER OF
C     CHARACTERS IN THIS STRING
C
        CALL MOVE(CHAR(JSTRT+1),TEMP(LTEMP+1),NCHR)
        LTEMP=LTEMP+NCHR $ GO TO 33
C
C     PROCESS ↓ FOLLOWED BY AN INTEGER
C
C     THE INTEGER FOLLOWING THE ↓ OPERATOR IS FOUND AND PLACED IN -I-
C
38      CALL GETNUM(I,CHAR(JSTRT+1),JSTOP-JSTRT-1,1)
C
***NOTE(7)
C
        I = I - 1
C
C     IF -I- EXCEEDS -LINLIM- IT IS RESET TO -LINLIM-
C
        IF(I.GT.LINLIM) I = LINLIM
```

```
C
C     DETERMINE IF THE COMPOSITION POINTER IS TO BE POSITIONED BEFORE,
C     AFTER OR AT THE CURRENT COMPOSITION POSITION
C
      IF(LTEMP-I) 39,33,40
C
C     COMPOSITION IS TO CONTINUE AT A COLUMN BEYOND THE CURRENT COMPOSITION
C     POSITION
C     -LIM- INDICATES THE NUMBER OF COLUMNS PAST THE CURRENT POSITION
C
39    LIM = I - LTEMP
C
C     PAD THE INBETWEEN COLUMNS WITH BLANKS
C
      DO 391 J=1,LIM
391   TEMP(J+LTEMP) = 1H
C
C     RESET THE COMPOSITION POINTER TO -I-
C
40    LTEMP=I $ GO TO 33
C
C     WORK ON THE TWO EQUIVALENCE OPERATORS ΞLN,M AND ΞRN,M
C
C     LOOK FOR THE COMMA SEPARATING THE PSEUDO REGISTER NUMBER FROM THE
C     NUMBER OF CHARACTERS TO BE CONCATENATED TO THE RESULT
C
41    L=INDEX(CHAR(JSTRT+2),JSTOP-JSTRT-2,COMMA) $ IF(L.EQ.0) GO TO 777
      CALL GETNUM(I,CHAR(JSTRT+2),L-1,1)
      CALL GETNUM(M,CHAR(JSTRT+L+2),JSTOP-JSTRT-L-2,1)
C
C     -L- IS SET TO THE NUMBER OF CHARACTERS IN PSEUDO REGISTER -I-
C
      L = LWORK(I)
C
C     IF THE NUMBER OF CHARACTERS ALREADY IN THE COMPOSED STRING PLUS THE
C     NUMBER TO BE CONCATENATED IS GREATER THAN -LINLIM-, RESET -M- TO
C     MAKE THE TOTAL EQUAL TO -LINLIM-
C
      IF(M+LTEMP.GT.LINLIM) M=LINLIM-LTEMP
C
C     TEST FOR AN -R- OR AN -L- TO SEE IF THE LEFTMOST OR RIGHTMOST -M-
C     CHARACTERS ARE TO BE USED
C
      IF(CHAR(JSTRT+1).NE.1HR) GO TO 383
C
```

```
C    IF THE NUMBER OF CHARACTERS REQUESTED IS GREATER THAN THE NUMBER
C    PRESENTLY IN THE STRING IN PSEUDO REGISTER -I-, THEN BLANKS ARE
C    ADDED ON THE LEFTSIDE OF THE CURRENT COMPOSED STRING IN -TEMP-
C    TO MAKE UP THE DIFFERENCE
C
      IF(L.GE.M) GO TO 382
      LIM = M - L
      DO 381 J=1,LIM
381   TEMP(J+LTEMP) = 1H
C
C    CONCATENATE THE STRING IN PSEUDO REGISTER -I- TO THAT IN -TEMP-
C
      CALL MOVE(WORK(1,I),TEMP(LIM+LTEMP+1),L) $ GO TO 385
382   CALL MOVE(WORK(L-M+1,I),TEMP(LTEMP+1),M) $ GO TO 385
C
C    AN ELN,M OPERATOR WAS ENCOUNTERED
C
383   IF(L.GE.M) GO TO 385
      LIM = M - L
      CALL MOVE(WORK(1,I),TEMP(LTEMP+1),L)
      DO 384 J=1,LIM
C
C    BLANKS ARE ADDED TO THE RIGHT SIDE OF THE CURRENT COMPOSED STRING
C
384   TEMP(LTEMP+L+J) = 1H
      GO TO 386
385   CALL MOVE(WORK(1,I),TEMP(LTEMP+1),M)
C
C    RESET THE NUMBER OF CHARACTERS IN THE RESULTING COMPOSED STRING
C
386   LTEMP = LTEMP + M $ GO TO 33
C
***NOTE(8)
C
C
C    PROCESS A RIGHT SIDE ASSOCIATIVE STORE.  FOR EXAMPLE +S/1+3+5/
C
C
C
C    OBTAIN THE PSEUDO REGISTER NUMBER REFERENCED AND STORE IT IN -I-
C
50    CALL GETNUM(I,CHAR(JSTRT),JSTOP-JSTRT,1)
C
C    INCREMENT THE NTUPLE POSITION IN WHICH THE CONTENTS OF PSEUDO
C    REGISTER -I- WILL BE STORED, AND STORE IT IN THIS POSITION
C
```

```
      NST=NST+1 $ NTUP(NST)=DEFSYM(WORK(1,I),LWORK(I)) $ GO TO 33
C
C    PROCESS THE RIGHT SIDE COMPOSITION OPERATOR CONSISTING OF A PSEUDO
C    REGISTER NUMBER
C
51    CALL GETNUM(I,CHAR(JSTRT),JSTOP-JSTRT,1)
C
C    IF MORE CHARACTERS ARE TO BE PLACED IN THE COMPOSED STRING THAN ARE
C    ALLOWED (I.E. THE STRING WOULD EXCEED -LINLIM- CHARACTERS) THE
C    NUMBER IS RESET SO THIS MAXIMUM WILL NOT BE EXCEEDED
C
      NCHR=LWORK(I) $ NDIFF=LINLIM-LTEMP $ IF(NCHR.GT.NDIFF) NCHR=NDIFF
      CALL MOVE(WORK(1,I),TEMP(LTEMP+1),NCHR)
      LTEMP=LTEMP+NCHR $ GO TO 33
C
C    WORK ON PATTERN MATCHING FOR LEFT SIDE OF RULE BODY
C
C    BEGIN PROCESSING OPERATORS FOR FIND AND ACCESS
C
C
C    PROCESS ASSOCIATIVE ACCESS
C
60    NST = NST + 1
      CALL GETNUM(NTUP(NST),CHAR(JSTRT),JSTOP-JSTRT,1) $ GO TO 33
C
C    PROCESS ASSOCIATIVE FIND REQUEST DEALING WITH A DOLLAR SIGN
C
61    NST=NST+1 $ NTUP(NST)=0 $ GO TO 33
C
C    PROCESS ASSOCIATIVE FIND REQUEST DEALING WITH A LITERAL
C
C
C    IPLUS-3 CONTAINS THE NUMBER OF CHARACTERS IN THE LITERAL STRING
C
62    NST=NST+1 $ NTUP(NST)=LOCSYM(CHAR(JSTRT+1),IPLUS-3) $ GO TO 65
C
C    PROCESS ASSOCIATIVE FIND REQUEST DEALING WITH A PSEUDO REGISTER
C    NUMBER
C
63    CALL GETNUM(I,CHAR(JSTRT),JSTOP-JSTRT,1)
C
C    CHECK IF THE PSEUDO REGISTER CONTAINS A DOLLAR SIGN
C
      IF(WORK(1,I).EQ.DOLLAR) GO TO 61
64    NST=NST+1 $ NTUP(NST)=LOCSYM(WORK(1,I),LWORK(I))
```

```
65     IF(NTUP(NST).EQ.0) GO TO 602 $ GO TO 33
C
C   WORK ON . OPERATOR
C
C
***NOTE(9)
C
C
C   OBTAIN THE NUMBER FOLLOWING THE DOT AND STORE IT IN -I-
C
66     CALL GETNUM(I,CHAR(JSTRT+1),JSTOP-JSTRT-1,1)
       NCHR = MIN0(80-LWORK(I),JSTOP-JSTRT+1)
       J = LWORK(I) + 1
       CALL MOVE(CHAR(JSTRT-1),WORK(J,I),NCHR)
       CALL MOVE(WORK(1,I),CHAR(JSTRT-1),LWORK(I)+NCHR)
       JSTOP=JSTOP+LWORK(I) $ NEND=NEND+LWORK(I) $ RNAME=END $ GO TO 33
C
C   WORK ON DECOMPOSITION OPERATORS BEGINNING WITH $
C
C
C
C   INCREMENT THE PSEUDO REGISTER NUMBER
C
67     NVARB = NVARB + 1
C
C   TEST IF THE OPERATOR IS SIMPLY A DOLLAR SIGN BY ITSELF OR ONE OF THE
C   OTHER OPERATORS THAT BEGIN WITH A DOLLAR SIGN
C
       IF(JSTOP-JSTRT.NE.1) GO TO 69
C
***NOTE(10)
C
C
C   IF -ISTOP- IS TRUE THEN THERE ARE NO MORE OPERATORS IN THE CURRENT
C   OPERATOR STRING
C
       IF(.NOT.ISTOP) GO TO 68
C
C   RECALL THAT STRINGS BEING OPERATED ON ARE STORED IN THE ARRAY -TEMP-
C   WITH THEIR LENGTHS IN -LTEMP-
C
       LWORK(NVARB) = LTEMP - IWORK + 1
       CALL MOVE(TEMP(IWORK),WORK(1,NVARB),LWORK(NVARB))
C
***NOTE(11)
C
```

```
      GO TO 33
C
***NOTE(12)
C
68    WORK(1,NVARB)=JCHAR $ LWORK(NVARB)=1 $ GO TO 63
C
C     THE DOLLAR SIGN - LITERAL OPERATOR (E.G. $≠ABC≠) AND THE DOLLAR SIGN
C     - INTEGER OPERATOR (E.G. $5) ARE EXAMINED HERE
C
69    IF(CHAR(JSTRT+1).NE.QUOTE) GO TO 71
C
C     START WORK ON $ FOLLOWED BY QUOTED STRING
C
C
***NOTE(13)
C
C     -IST- IS A TEMPORARY POINTER TO THE CURRENT POSITION IN THE INPUT
C     STRING
C
C     INITIALIZE NUMBER OF CHARACTERS IN CURRENT PSEUDO REGISTER TO ZERO
C
C     -I- IS THE NUMBER OF CHARACTERS IN THE LITERAL FOLLOWING THE $
C     OPERATOR
C
      IST=0 $ LWORK(NVARB)=0 $ I=IPLUS-4
C
C     TEST IF PRESENT POSITION IN INPUT STRING IS GREATER THAN THE NUMBER
C     OF CHARACTERS IN INPUT STRING
C
70    IF(IWORK+IST+I-1.GT.LTEMP) GO TO 76
C
C     COMPARE -I- CHARACTERS STARTING FROM THE CURRENT INPUT STRING
C     POSITION WITH THE -I- CHARACTERS OF THE LITERAL TO BE MATCHED
C
      IF(INDEX(TEMP(IWORK+IST),I,CHAR(JSTRT+2),I).NE.1) GO TO 76
C
***NOTE(14)
C
      J=LWORK(NVARB)+1 $ CALL MOVE(CHAR(JSTRT+2),WORK(J,NVARB),I)
C
C     INCREMENT NUMBER OF CHARACTERS IN CURRENT PSEUDO REGISTER AND
C     INCREMENT THE POSITION IN THE INPUT STRING, THEN REPEAT THE WHOLE
C     PROCESS
C
      LWORK(NVARB)=LWORK(NVARB)+I $ IST=IST+I $ GO TO 70
```

```
C
C     START WORK ON THE DOLLAR SIGN FOLLOWED BY AN INTEGER
C
C     OBTAIN THE NUMBER FOLLOWING THE DOLLAR SIGN AND STORE IT IN -I-
C
71      CALL GETNUM(I,CHAR(JSTRT+1),JSTOP-JSTRT-1,1)
C
C     IF THE CURRENT POSITION (STORED IN -IWORK-) IN THE INPUT STRING PLUS
C     THE NUMBER OF REMAINING CHARACTERS TO BE MATCHED (THIS NUMBER STORED
C     IN -I-) IS GREATER THAN THE TOTAL LENGTH OF THE INPUT STRING (STORED
C     IN -LTEMP-) AN ERROR IS SIGNALLED
C
        IF(I+IWORK-1.GT.LTEMP) GO TO 602
C
C     MOVE THE -I- CHARACTERS SPECIFIED BY THE DOLLAR - INTEGER OPERATOR
C     FROM THE CURRENT INPUT STRING INTO THE CURRENT PSEUDO REGISTER.
C     ALSO, PLACE THE NUMBER OF CHARACTERS MOVED IN -LWORK(NVARB)-
C
        CALL MOVE(TEMP(IWORK),WORK(1,NVARB),I) $ LWORK(NVARB)=I $ GO TO 76
C
C     WORK ON ↓ FOLLOWED BY AN INTEGER
C
C
C
C     OBTAIN THE NUMBER AFTER THE ↓ AND PLACE IT IN -I-
C
73      NVARB=NVARB+1 $ CALL GETNUM(I,CHAR(JSTRT+1),JSTOP-JSTRT-1,1)
C
C     DETERMINE IF THE CURRENT POSITION IN THE INPUT STRING (STORED IN
C     -IWORK-) IS PASSED THE DESIRED COLUMN NUMBER (STORED IN -I-)
C
        IF(IWORK.GE.I) GO TO 74
C
C     IF THE DESIRED COLUMN NUMBER IS GREATER THAN THE TOTAL LENGTH OF THE
C     INPUT STRING AN ERROR IS SIGNALLED
C
        IF(LTEMP.LT.I-1) GO TO 602
C
C     I-IWORK GIVES THE NUMBER OF CHARACTERS TO BE MOVED FROM THE INPUT
C     STRING TO THE CURRENT PSEUDO REGISTER
C
        CALL MOVE(TEMP(IWORK),WORK(1,NVARB),I-IWORK)
C
C     THE NUMBER OF CHARACTERS MOVED IS PLACED IN -LWORK(NVARB)-
C
        LWORK(NVARB) = I - IWORK $ GO TO 76
```

```
C
C     CONTROL COMES HERE IF THE CURRENT POSITION IN THE INPUT STRING IS
C     PAST THE REQUIRED COLUMN NUMBER - IN THIS CASE THE POSITION POINTER
C     -IWORK- IS RESET AND THE NULL STRING IS INDICATED BY LWORK(NVARB)=0
C
74      IWORK = I $ LWORK(NVARB)=0 $ GO TO 76
C
C     WORK ON LITERALS
C
C
C     THE NUMBER OF CHARACTERS  INBETWEEN THE QUOTES IS GIVEN BY IPLUS-3
C
75      NVARB=NVARB+1 $ LWORK(NVARB)=IPLUS-3
C
C     MOVE THE MATCHED CHARACTERS INTO THE CURRENT PSEUDO REGISTER
C
        CALL MOVE(CHAR(JSTRT+1),WORK(1,NVARB),LWORK(NVARB)) $ GO TO 76
C
C     THE FOLLOWING CODE WILL FIND THE LEFT-MOST STRING WHICH WILL MATCH
C     A LITERAL FROM THE COLLECTION OF LITERALS STORED UNDER A VARIABLE
C     NAME. THE FORM OF THE COLLECTION OF LITERALS IS ≠LIT1≠LIT2≠LIT3≠)
C
C     INCREMENT THE PSEUDO REGISTER NUMBER
C
C     PACK AND LOOK UP THE VARIABLE NAME
C
81      NVARB=NVARB+1 $ CALL PACK(CHAR(JSTRT),I,JSTOP-JSTRT)
C
C     IF THE NAME CANNOT BE FOUND IN THE DICTIONARY AN ERROR HAS
C     OCCURRED
C
        LOC = LOCATE(I,DICT) $ IF(LOC.EQ.0) GO TO 775
C
C     OBTAIN THE NUMBER OF CHARACTERS IN THE STRING REFERENCED AND PLACE
C     IN -JCHAR-
C
        JCHAR = LSTORE(LOC-4)
C
C     UNPACK THE CONTENTS INTO THE 80 WORD ARRAY -EXTRA-
C
        CALL UNPACK(STORE(1,LOC-4),EXTRA,JCHAR)
C
C     FIND THE PAIR OF RIGHT PARENS AT THE END OF THE COLLECTION OF
C     LITERALS
C     IF TWO RIGHT PARENTHESES ARE NOT FOUND, AN ERROR HAS OCCURRED
```

```
C
C     -PRNR IS IN THE COMMON BLOCK -RESRVD- AND IS IMMEDIATELY FOLLOWED
C     BY -PARENR-. BOTH THESE VARIABLES ARE INITIALIZED IN SUBROUTINE
C     -SETDICT-
C
        JEND = INDEX(EXTRA,JCHAR,PRNR,2) $ IF(JEND.EQ.0) GO TO 777
C
***NOTE(15)
C
C
C     -IST- IS A CHARACTER POSITION POINTER IN THE LITERAL COLLECTION
C     PATTERN STRING AND ALWAYS INDICATES THE FIRST CHARACTER POSITION OF
C     THE LITERAL NOW BEING USED FOR MATCHING
C
        IST = 2 $ NFND = LTEMP - IWORK + 2
C
C     FIND THE QUOTE AT THE END OF THE NEXT LITERAL
C
C     IF NO MORE QUOTES CAN BE FOUND THE LITERALS IN THE COLLECTION
C     PATTERN HAVE ALL BEEN TESTED
C
82      NQ = INDEX(EXTRA(IST),JEND-IST,QUOTE) $ IF(NQ.EQ.0) GO TO 83
        JCHAR = NQ - 1
C
C     SEE IF THE LITERAL CAN BE FOUND IN THE STRING BEING MATCHED
C
        IQ = INDEX(TEMP(IWORK),LTEMP - IWORK + 1,EXTRA(IST),JCHAR)
C
C     -ILT- CONTAINS THE LITERAL STRING POSITION OF THE LITERAL JUST
C     PATTERNED FOR A MATCH
C
C
C     SET THE POINTER TO THE FIRST POSITION OF THE NEXT LITERAL IN THE
C     COLLECTION PATTERN
C
C
C     IF A MATCH IS NOT FOUND (IQ=0) OR THERE IS STILL A POSSIBILITY OF A
C     MATCH FURTHER TO THE LEFT THAN THE CURRENT ONE, THE SEARCH IS BEGUN
C     AGAIN AT STATEMENT 82
C
        ILT = IST $ IST = IST + NQ $ IF(IQ.EQ.0.OR.IQ.GE.NFND) GO TO 82
C
C     LITERAL FOUND TO THE LEFT OF ALL PREVIOUSLY FOUND LITERALS
C
C     STORE THE MATCHED LITERAL IN THE PRESENT PSEUDO-REGISTER-REFERENCED,
```

```
C   WITH THE NUMBER OF CHARACTERS IT HAS IN -LWORK-
C
       CALL MOVE(EXTRA(ILT),WORK(1,NVARB),JCHAR)
       LWORK(NVARB) = JCHAR $ NFND = IC $ GO TO 82
C
C   IF -NFND- HAS NOT CHANGED SINCE INITIALIZATION, THEN NONE WERE
C   FOUND
C
83     IF(NFND.EQ.LTEMP-IWORK+2) GO TO 602 $ GO TO 78
C
C   THE FOLLOWING CODE TAKES CARE OF CLEAN UP AFTER ANY OF THE
C   OPERATORS GIVEN BELOW ARE EXECUTED:
C
C   $ FOLLOWED BY A QUOTED STRING
C   $ FOLLOWED BY AN INTEGER
C   LITERAL
C   LITERAL COLLECTION PATTERN
C
C
C   DETERMINE IF THERE ARE MORE OPERATORS TO FOLLOW
C
76     IF(.NOT.ISTOP) GO TO 77
C
***NOTE(16)
C
       IST = LTEMP - LWORK(NVARB) + 1
       NFND = IST - IWORK + 1
       IF(INDEX(TEMP(IST),LWORK(NVARB),WORK(1,NVARB),LWORK(NVARB)).NE.1)
      1 GO TO 602
       GO TO 78
C
C   THE POSITION IN THE INPUT STRING WHERE THE CURRENT MATCH WAS MADE IS
C   PLACED IN -NFND-
C
77     NFND = INDEX(TEMP(IWORK),LTEMP-IWORK+1,WORK(1,NVARB),LWORK(NVARB))
C
C   THE NULL STRING IS MATCHED IF LWORK(NVARB)=0
C
       IF(LWORK(NVARB).EQ.0) NFND = 1
78     IF(NFND.EQ.0.OR.(NVB.EQ.NVARB.AND.NFND.NE.1)) GO TO 602
C
C   TEST THE NUMBER OF MATCHES MADE INDICATED BY THE NUMBER OF PSEUDO
C   REGISTERS USED
C
       IF(NVARB.LE.1) GO TO 80
```

```
C
***NOTE(17)
C
      IF(WORK(1,NVARB-1).NE.DOLLAR) GO TO 79
C
C    OBTAIN THE NUMBER OF CHARACTERS MATCHED BY THE PREVIOUS $ OPERATOR
C    AND MOVE THESE CHARACTERS INTO THE SPECIFIED PSEUDO REGISTER
C
      LWORK(NVARB-1) = NFND - 1
      CALL MOVE(TEMP(IWORK),WORK(1,NVARB-1),LWORK(NVARB-1))
      GO TO 80
C
***NOTE(18)
C
79    IF(NFND.NE.1) GO TO 602
C
C    THE PRESENT POSITION POINTER FOR THE INPUT STRING IS RESET
C    ACCORDING TO THE MATCHES JUST MADE
C
80    IWORK = IWORK + NFND + LWORK(NVARB) - 1
C
C    THIS IS THE END OF THE PATTERN LOOP
C
33    IF(.NOT.ISTOP) GO TO 34
      RETURN
602   IPATRN=2 $ RETURN
775   IPATRN=3 $ RETURN
777   IPATRN=4 $ RETURN
      END
```

```
      SUBROUTINE SETDICT(I)
C
C     THE SUBROUTINE -SETDICT- IS NEVER CALLED FROM ANOTHER ROUTINE
C     (THIS OF COURSE IS DUE TO THE FACT THAT NO EXECUTION STATEMENTS
C     OCCUR). ONLY THE ENTRY POINT -RDLEX- IS CALLED FROM THE FUNCTION
C     -LOAD-.
C
C
C     THIS IS A MACHINE AND SYSTEM DEPENDENT ROUTINE TO SET UP
C     DICTIONARIES AND INITIALIZE CONSTANTS
C
      COMMON/SWTCHS/IVAL(63),LPNCH,IEVAL,TOUT,ISTOP,IREAD,ISIDE,NVARB
      COMMON/RESRVD/BEGIN,ECH,ECHOFF,RETRN,DUMP,PRNTON,PRNTOFF,PNCH,
     1 NPNCH,READL,END,EQUAL,QUOTE,PLUS,PARENL,BREAK,PRNR,PARENR,COMMA,
     2 ATSIGN,PERIOD,PERCNT,AMPERS,DOLLAR,ASTRSK,SHARP,COMND(3)
      COMMON/STORG/DICT(261),WORK(80,9),STORE(3,257),PUSHDN(100),ITOP
     1,LWORK(9),LSTORE(257),RNAME,TNAME,FNAME,VNAME,BUFF(3),CHAR(80),
     2LCHAR,TEMP(80),LTEMP,NTUP(4),NST,EXTRA(80),NAMLIM,LINLIM
      LOGICAL LPNCH
C
C     -NAMLIM- IS THE LENGTH LIMIT (IN NUMBER OF CHARACTERS) OF RULE NAMES
C     AND VARIABLES USED IN RULES
C
C     -LINLIM- IS THE INPUT-OUTPUT CHARACTER STRING LIMIT
C
      DATA NAMLIM,LINLIM/10,80/
      DATA EQUAL,QUOTE,PLUS,PARENL,PARENR,COMMA/1H=,1H≠,1H+,1H(,1H),1H,/
      DATA ATSIGN,PERIOD,PERCNT,AMPERS,DOLLAR/1H↓,1H.,1H≡,1H+,1H$/
      DATA ASTRSK,SHARP,BREAK,PRNR/1H*,1H,,1H/,1H)/
C
C     -S-, -F- AND -A- REFER TO THE ASSOCIATIVE MEMORY COMMANDS -STORE-,
C     -FIND- AND -ACCESS- RESPECTIVELY
C
      DATA COMND/1HS,1HF,1HA/
      DATA BEGIN,END,ECH,ECHOFF/7HRESTART,3HEND,4HECHO,6HNOECHO/
      DATA PRNTON,PRNTOFF,DUMP,RETRN/5HTRACE,7HNOTRACE,4HDUMP,6HRETURN/
      DATA PNCH,NPNCH,LPNCH,READL/5HPUNCH,7HNOPUNCH,.FALSE.,6HREADLX/
      DATA IVAL/26*1,10*2,6*9,3,9,4,9,5,6,3*9,7,4*9,8,6*9/
      NAMELIST/DATA/ECHO,PARENL,PARENR,ECH,ECHOFF,BEGIN,BUFF,CHAR,RNAME,
     1 INTAP,NOUTAP,TRACE,NAMLIM
      RETURN
      ENTRY RDLEX
C
C     PERFORM A FORTRAN -NAMELIST- READ
C
```

```
READ(INTAP,DATA4) $ I=1
RETURN
END
```

```
      SUBROUTINE INITAL
C
C     SUBROUTINE -INITIAL-  IS CALLED FROM SUBROUTINE -STRAN- ONLY
C
      COMMON LIST(4000)
      COMMON/INFO/IFREE,LOC(1),IBASE
      COMMON/ASSOC/LHASH,HASH(500),NXTFRE,SYMBOL(500)
      COMMON/CNTROL/ECHO,TRACE,JUNK(11),INTAP,NOUTAP
     1,JJUNK(4)
      COMMON/STORG/DICT(261),WORK(80,9),STORE(8,257),PUSHON(100),ITOP
     1,LWORK(9),LSTORE(257)
      LOGICAL ECHO,TRACE
      INTEGER DICT,XLOCF,HASH,SYMBOL
      DATA ECHO,TRACE,INTAP,NOUTAP/.TRUE.,.FALSE.,5,6/
C
C     -ITOP- IS USED AS THE INDEX TO ARRAY -PUSHON- IN SUBROUTINE -PUSH-
C
      ITOP = 0
C
C     THE VARIABLES -IFREE-, -IBASE-, -LHASH-, NXFRE-, AND THE ARRAY
C     -LIST- ARE USED IN ASSOCIATIVE MEMORY ROUTINES
C
      IFREE = XLOCF(LIST)
      DO 1 I=1,3999
1     LIST(I) = IFREE + I
      IBASE = XLOCF(LOC) - 1
      LIST(4000)=0
      LHASH = 500
      DO 2 I=1,LHASH
2     HASH(I) = 0
      NXTFRE = XLOCF(SYMBOL)
C
C     THE ARRAY -DICT- IS USED IN FUNCTION -LOCATE- FOR RULE NAME
C     STORAGE
C
      DICT(1) = 257
      DO 3 I=2,261
3     DICT(I) = 0
      RETURN
      END
```

```
      SUBROUTINE PUSH(CHAR,IM1)
C
C   SUBROUTINE -PUSH- IS CALLED FROM SUBROUTINE -INTERP- ONLY
C
C
***NOTE(1)
C
      COMMON/STORG/DICT(261),WORK(80,9),STORE(8,257),PUSHON(100),ITOP
     1,LWORK(9),LSTORE(257)
      INTEGER COMMA,CHAR(2)
      DATA COMMA/1H,/
C
C   -IM1- POINTS TO THE LAST CHARACTER BEFORE THE FIRST OF THE TWO
C   TERMINATING BRACKETS
C
      I = IM1 + 1
      ILST = IM1
C
C   THE DO LOOP COUNTS OUT THE TOTAL NUMBER OF CHARACTERS IN THE STRING
C   STARTING WITH THE LAST CHARACTER AND WORKING BACKWARDS ONE AT A TIME
C
      DO 5 J=1,IM1
C
C   -NXT- IS USED TO POINT TO EACH CHARACTER OF THE STRING, STARTING
C   WITH THE LAST CHARACTER AND WORKING BACKWARDS ONE AT A TIME
C
      NXT = I - J
C
C   SEARCH FOR A COMMA SEPERATING TWO OR MORE RULE NAMES IN THE STRING
C   (THERE IS ALSO THE POSSIBILITY OF ONLY ONE RULE NAME IN THE STRING)
C
      IF(CHAR(NXT).NE.COMMA) GO TO 5
C
***NOTE(2)
C
      IF(ILST.EQ.NXT) GO TO 4
C
C   IF -ITOP- IS EQUAL TO 100, IT IS NOT INCREMENTED AND EACH RULE NAME
C   ENCOUNTERED FROM THEN ON IS PLACED IN PUSHON(100) THUS WIPING OUT
C   THE PREVIOUS RULE NAME STORED THERE
C
      IF(ITOP.NE.100) ITOP = ITOP + 1
C
C   -NCNT- IS SET TO THE NUMBER OF CHARACTERS IN THE RULE NAME. IF THIS
C   IS GREATER THAN 10, ONLY THE FIRST 10 ARE CONSIDERED
```

```
C
      NCNT = ILST - NXT
      IF(NCNT.GT.10) NCNT = 10
C
C   THE RULE NAME STORED UNPACKED IN -CHAR- IS PACKED INTO THE NEXT
C   LOCATION OF -PUSHDN-
C
      CALL PACK(CHAR(NXT+1),PUSHDN(ITOP),NCNT)
C
C   -ILST- IS SET TO POINT TO THE LAST CHARACTER OF THE NEXT RULE NAME
C   BEING EXAMINED
C
4     ILST = NXT - 1
5     CONTINUE
C
***NOTE(3)
C
      IF(ILST.EQ.0) RETURN
C
C   THE SAME OPERATIONS AS ABOVE ARE REPEATED FOR THE LAST RULE NAME
C   ENCOUNTERED
C
      IF(ITOP.NE.100) ITOP = ITOP + 1
      IF(ILST.GT.10) ILST = 10
      CALL PACK(CHAR,PUSHDN(ITOP),ILST)
      RETURN
      END
```

```
      INTEGER FUNCTION FIND(N,NTUPL,IFOLOW)
      COMMON/ASSOC/LHASH,HASH(500),NXTFRE,SYMBOL(500)
3     ILOC = INDEX(CHAR,N,INTEGR,8)
      IF(ILOC.EQ.0) GO TO 4
      WBIT = .FALSE.
      ICHAR = ILOC + 7
      GO TO 2
C
C   TEST FOR A SUBROUTINE CALL STATEMENT
C
4     ICALL = INDEX(CHAR,N,CALL,5)
      IF(ICALL.EQ.0) GO TO 6
C
C   IF THERE ARE NO ARGUMENTS IN THE SUBROUTINE CALL STATEMENT, THE
C   SUBROUTINE -EXECUTE- IS CALLED TO LOAD THE SUBROUTINE
C
      IF(INDEX(CHAR(ICALL+5),N-4-ICALL,PARENL).NE.0) GO TO 5
      CALL EXECUTE
      RETURN
5     ICHAR = ICALL + 4
      GO TO 7
C
C   -ICHAR- IS SET TO THE POSITION OF THE EQUALS SIGN AND -M- TO THE
C   TOTAL STRING LENGTH
C
6     ICHAR = IEQ
7     M = N
      ASSIGN = 0
C
C   THE FOLLOWING SECTION OF CODING (UP TO STATEMENT LABEL 15) EXAMINES
C   THE INPUT STRING FOR SPECIAL CHARACTERS INCLUDING BLANK, PLUS
C   MINUS, ASTERISK, SLASH, OPENING PARENTHESIS, CLOSING PARENTHESIS
C   AND COMMA. EACH OF THESE CHARACTERS HAS BEEN PREVIOUSLY STORED IN
C   THE ELEMENTS OF ARRAY -SYMB- (ONE CHARACTER PER ELEMENT) BY THE
C   USE OF A DATA STATEMENT
C
8     NBEG = 0
      FLAG = .FALSE.
      L = 0
      MM = 0
C
C   -LEVEL- IS USED TO INDICATE THE LEVEL OF NESTING OF BRACKETS. EACH
C   TIME A LEFT PARENTHESIS IS ENCOUNTERED, -LEVEL- IS INCREMENTED BY
C   1 AND EACH TIME A RIGHT PARENTHESIS IS ENCOUNTERED IT IS
C   DECREMENTED BY 1
```

```
C
      LEVEL = 0
9     L = L + 1
      IF(L.GT.100) GO TO 777
C
C   INCREMENT THE STRING POSITION POINTER
C
10    ICHAR = ICHAR + 1
C
C   TEST IF THE ENTIRE INPUT STRING HAS BEEN EXAMINED
C
      IF(ICHAR.GT.M) GO TO 40
C
C   STORE THE CURRENT CHARACTER POINTED TO BY -ICHAR- IN THE VARIABLE
C   -TEST-
C
      TEST = CHAR(ICHAR)
C
C   DETERMINE IF THE CURRENT CHARACTER IS ANY OF THOSE STORED IN ARRAY
C   -SYMB- AND IF SO, GO TO THE SECTION OF PROGRAMMING DEALING WITH THAT
C   CHARACTER
C
      DO 11 I=1,8
      IF(TEST.EQ.SYMB(I)) GO TO(15,16,20,23,26,27,28,29),I
11    CONTINUE
C
C   IF THE CURRENT CHARACTER IS NOT ONE OF THOSE STORED IN -SYMB-
C   THEN A TEST IS MADE FOR A PERIOD OR A DIGIT FROM ZERO TO NINE.
C   THIS TEST IS ONLY MADE ONCE SINCE IF A REAL OR INTEGER NUMBER IS
C   NOT FOUND A FLAG IS SET TO INDICATE THIS AND THE TEST NEED NOT BE
C   MADE AGAIN
C
      IF(FLAG) GO TO 10
      NUMBSW = .FALSE.
C
C   IF A PERIOD (I.E. DECIMAL POINT) OR A DIGIT FROM ZERO TO NINE IS
C   ENCOUNTERED, THE FLAG -NUMBSW- IS SET
C
      IF(TEST.EQ.PERIOD.OR.(TEST.GE.1H0.AND.TEST.LE.1H9)) NUMBSW=.TRUE.
C
C   -FLAG- IS SET TO .TRUE. INDICATING THE TEST FOR A NUMBER HAS BEEN
C   MADE ONCE AND NEED NOT BE MADE AGAIN
C
      FLAG = .TRUE.
C
```

```
C     -NBEG- IS SET TO THE CHARACTER POSITION WHERE THE NUMBER STARTS
C
        NBEG = ICHAR
C
C     THE NEXT CHARACTER IS NOW EXAMINED
C
        GO TO 10
C
C     IN THE REST OF THE CODING OF THIS SUBROUTINE REFERENCE WILL BE
C     MADE TO THE ARRAYS -SOURCE-, -SHIER-, -DHIER-, -FLOT-, AND
C     -OPSTCK-. THESE ARE NOT USED IN QUITE THE SAME WAY AS WAS DESCRIBED
C     IN SECTION 1.3.2.1 OF THIS REPORT BUT THERE ARE SIMILARITIES.
C     -SOURCE- IS USED TO REPRESENT THE UNARY OPERATIONS AND THE BUILT-
C     IN FUNCTIONS, AND TO STORE UP TO FIVE SUBSCRIPTED VARIABLE NAMES IN
C     THE FOLLOWING WAY:
C
C     SOURCE=1 → UNARY +
C     SOURCE=2 → UNARY -
C     SOURCE=3 → FLOAT
C     SOURCE=4 → FIX
C     SOURCE=5 → ABS
C     SOURCE=6 → SIN
C     SOURCE=7 → COS
C     SOURCE=8 → TAN
C     SOURCE=9 → ATAN
C     SOURCE=10 → EXP
C     SOURCE=11 → ALOG
C     SOURCE=12 → ALOG10
C     SOURCE=13 → SQRT
C     SOURCE=14 → FIRST SUBSCRIPTED VARIABLE NAME
C     SOURCE=15 → SECOND SUBSCRIPTED VARIABLE NAME
C     SOURCE=16 → THIRD SUBSCRIPTED VARIABLE NAME
C     SOURCE=17 → FOURTH SUBSCRIPTED VARIABLE NAME
C     SOURCE=18 → FIFTH SUBSCRIPTED VARIABLE NAME
C
C     -SHIER- IS USED TO DISTINGUISH THE FOLLOWING:
C
C     SHIER=-3 → ERROR
C     SHIER=-2 → ADDRESS
C     SHIER=-1 → REAL
C     SHIER=0 → INTEGER
C     SHIER=1 → OPENING BRACKET
C     SHIER=2 → CLOSING BRACKET
C     SHIER=3 → BINARY + AND -
C     SHIER=4 → * AND /
```

```
C     SHIER=5 → ** AND MOD
C     SHIER=6 → UNARY OPERATORS, FUNCTIONS AND SUBSCRIPTED VARIABLES
C     SHIER=7 → COMMA
C     WHEN SHIER=6, -SOURCE- DETERMINES WHICH UNARY OPERATORS, FUNCTIONS
C     OR SUBSCRIPTED VARIABLES ARE INVOLVED
C     -OPSTCK- IS USED TO STORE THE OPERATORS USED
C     -OHIER- IS USED TO STORE THE OPERATOR HIERARCHY AND HAS VALUES
C     ASSIGNED (FROM -3 TO +7) IN EXACTLY THE SAME MANNER AS IS DONE
C     WITH -SHIER-
C     -FLOT- IS EQUIVALENCED TO -SOURCE- AT THE BEGINNING OF THE PROGRAM
C     TO DEAL WITH ANY REAL QUANTITIES THAT MAY BE USED IN CALCULATIONS
C     NOTE THAT -SOURCE- IS AN INTEGER ARRAY
C
C
C     PROCESS A BLANK
C
15      IF(FLAG) CALL NUMBER(CHAR,NBEG,ICHAR)
        GO TO 10
C
C     PROCESS A PLUS
C
16      IF(.NOT.FLAG) GO TO 17
C
C     TEST FOR A PLUS EXPONENT (E.G. 10.E+5)
C
        IF(CHAR(ICHAR-1).EQ.1HE.AND.NUMBSW) GO TO 10
C
C     OBATIN THE NUMBER FOLLOWING THE PLUS SIGN AND STORE IT IN THE
C     ARRAY -SOURCE-. EACH TIME A NUMBER IS REQUIRED THE SUBROUTINE
C     -NUMBER- IS USED TO OBTAIN IT
C
        CALL NUMBER(CHAR,NBEG,ICHAR)
17      SOURCE(L) = 1
C
C     TEST FOR UNARY OPERATION
C
C
C
C     IF L=1 ONLY ONE CHARACTER HAS BEEN ENCOUNTERED SO WE ARE DEALING
C     WITH A UNARY PLUS OR MINUS
C
18      IF(L.EQ.1) GO TO 22
C
C     IF EITHER OF THE FOLLOWING CONDITIONS ARE TRUE WE ARE DEALING WITH
C     A UNARY PLUS OR MINUS
C
```

```
      IF(SHIER(L-1).GT.2.OR.SHIER(L-1).EQ.1) GO TO 22
19    SHIER(L) = 3
      GO TO 9
C
C    PROCESS MINUS
C
C
C    IF -FLAG- IS NOT TRUE WE ARE DEALING WITH A VARIABLE NAME
C
20    IF(.NOT.FLAG) GO TO 21
C
C    TEST FOR A NEGATIVE EXPONENT (E.G. 10.E-5)
C
      IF(CHAR(ICHAR-1).EQ.1HE.AND.NUMBSW) GO TO 10
      CALL NUMBER(CHAR,NBEG,ICHAR)
21    SOURCE(L) = 2
      GO TO 18
22    SHIER(L) = 6
      GO TO 9
C
C    PROCESS ASTERISK OR DOUBLE ASTERISK
C
23    IF(FLAG) CALL NUMBER(CHAR,NBEG,ICHAR)
C
C    DETERMINE WHETHER MULTIPLICATION (ONE ASTERISK) OR EXPONENTIATION
C    (TWO ASTERISKS)
C
      IF(CHAR(ICHAR+1).NE.1H*) GO TO 24
      ICHAR = ICHAR + 1
      SOURCE(L) = 5
      SHIER(L) = 5
      GO TO 9
C
C    PROCESS SINGLE ASTERISK
C
24    SOURCE(L) = 3
25    SHIER(L) = 4
      GO TO 9
C
C    PROCESS SLASH
C
26    IF(FLAG) CALL NUMBER(CHAR,NBEG,ICHAR)
      SOURCE(L) = 4
      GO TO 25
C
```

```
C     PROCESS LEFT PARENTHESIS
C
27      IF(FLAG) CALL NUMBER(CHAR,NBEG,ICHAR)
        SHIER(L) = 1
        LEVEL = LEVEL + 1
        GO TO 9
C
C     PROCESS RIGHT PARENTHESIS
C
28      IF(FLAG) CALL NUMBER(CHAR,NBEG,ICHAR)
        SHIER(L) = 2
        LEVEL = LEVEL - 1
        GO TO 9
C
C     PROCESS COMMA
C
29      IF(FLAG) CALL NUMBER(CHAR,NBEG,ICHAR)
C
C     EACH TIME A COMMA IS ENCOUNTERED IT IS REPLACED BY THE CHARACTER
C     STRING ),( AND IN THIS WAY EVALUATION OF EXPRESSIONS IS FORCED BY
C     PARENTHESIS
C
        SHIER(L) = 2
        SHIER(L+1) = 7
        SHIER(L+2) = 1
        SOURCE(L+1) = 1
        L = L + 2
        GO TO 9
C
C     DONE WITH FIRST PASS
C
C     AT THIS POINT THE SECOND PASS AS DESCRIBED IN SECTION 1.3.2.1 BEGINS
C
40      IF(FLAG) CALL NUMBER(CHAR,NBEG,ICHAR)
C
C     AN ERROR EXISTS IF ONLY ONE OPERATOR HAS BEEN PICKED UP OR IF THERE
C     IS IMPROPER NESTING OF BRACKETS
C
        IF(L.EQ.1.OR.LEVEL.NE.0) GO TO 777
C
C     -I-, -J-, AND -K- CORRESPOND TO THE SAME VARIABLES GIVEN IN FIGURE
C     1.6 OF SECTION 1.3.2.1
C
C     THE CODING FROM THIS POINT TO THE END OF THE SUBROUTINE ARRANGES
C     THE INPUT STRING IN PREFIX POLISH FORM TO ESTABLISH PRECEDENCE
```

```
C     OF OPERATORS AND EVALUATES THE RESULT
C
        I = 1
        J = 2
        K = 1
        SHIER(L) = 0
C
C     INITIALIZE FIRST ELEMENT OF OPERATOR HIERARCHY ARRAY
C
        OHIER(1) = -10
C
C     TEST FOR AN ERROR OR A NUMBER WHICH IS EITHER AN ADDRESS, A REAL
C     OR AN INTEGER
C
41      IF(SHIER(I).LT.1) GO TO 42
C
C     TEST FOR A CLOSING BRACKET
C
        IF(SHIER(I).EQ.2) GO TO 45
C
C     PLACE THE OPERATOR OR SUBSCRIPTED VARIABLE NAME IN THE OPERATOR
C     STACK AND ITS HIERARCHY IN THE ARRAY -OHIER-
C
        OPSTCK(J) = SOURCE(I)
        OHIER(J) = SHIER(I)
C
C     -JOP- KEEPS TRACK OF THE NUMBER (FROM 1-5) OF THE SUBSCRIPTED
C     VARIABLE BEING EXAMINED
C
        JOP = OPSTCK(J) - 13
C
C     TEST FOR THE OCCURRENCE OF A SUBSCRIPTED VARIABLE AND STORE ITS
C     STACK POSITION (FOR PREFIX POLISH REPRESENTATION) IN THE ARRAY
C     -ARGLOC-
C
        IF(OHIER(J).EQ.6.AND.OPSTCK(J).GT.13) ARGLOC(JOP) = K
        I = I + 1
        J = J + 1
        GO TO 41
C
C     TEST FOR A REAL NUMBER
C
42      IF(SHIER(I).EQ.-1) GO TO 43
        SOURCE(K) = SOURCE(I)
        GO TO 44
```

```
43    FLOT(K) = FLOT(I)
44    SHIER(K) = SHIER(I)
      K = K + 1
      GO TO 46
45    J = J - 1
46    I = I + 1
C
C   TEST IF THE OPERATOR AT THE TOP OF THE STACK (I.E. THE ONE
C   PRECEEDING THE CURRENT OPERATOR) HAS A HIGHER PRECEDENCE THAN
C   THE CURRENT OPERATOR OR OPERAND
C
47    IF(OHIER(J-1).GE.SHIER(I)) GO TO 51
      IF(I.LT.L) GO TO 41
      IF(SHIER(K-1).EQ.-1) GO TO 48
C
C   OBTAIN THE CHARACTER CODE REPPRESENTATION OF THE INTEGER NUMBER
C   STORED IN SOURCE(K-1) OR THE REAL NUMBER STORED IN FLOT(K-1) AND
C   STORE IT IN CHAR(NSAV+1). RECALL -NSAV- MARKS THE POSITION OF THE
C   EQUALS SIGN IN THE INPUT STRING (IF NO EQUALS SIGN EXISTS, NSAV=0)
C
      CALL GETCHR(CHAR(NSAV+1),NCHR,SOURCE(K-1),1)
      GO TO 49
48    CALL GETCHR(CHAR(NSAV+1),NCHR,FLOT(K-1),3)
C
C   THE NUMBER OF CHARACTERS IN THE INPUT STRING IS NOW INCREASED BY
C   -NCHR-
C
49    N = NSAV + NCHR
      IF(IEQ.EQ.0) RETURN
      N = N + 1
C
C   IF AN EQUALS SIGN IS PRESENT IN THE INPUT STRING, THE DOLLAR SIGN
C   CHARACTER IS ADDED TO THE END OF THE STRING. FOR EXAMPLE, I=1
C   BECOMES I=1$
C
      CHAR(N) = 1H$
C
C   -STORNL- IS AN ENTRY POINT IN THE SUBROUTINE -GETNL-. THIS
C   SUBROUTINE IS USED FOR THE STORAGE AND RETRIEVAL OF NUMERIC DATA FED
C   TO OR PRODUCED BY THE EVALUATOR
C
      CALL STORNL(CHAR,N,SOURCE(K-1),FLOT(K-1),SHIER(K-1))
      RETURN
C
C   TEST FOR A BINARY OR UNARY OPERATION
```

```
C
51      IF(OHIER(J-1).GT.5) GO TO 53
C
C    PROCESS A BINARY OPERATOR. THE PARTICULAR OPERATOR TO BE PROCESSED
C    IS STORED IN -ITRAN-
C
        ITRAN = OPSTCK(J-1)
        COMMON/STORG/XXXX(251),WORK(80,9),STORE(8,257),PUSHDN(100),ITOP
      1,LWORK(9),LSTORE(257)
        COMMON/INFO/IFREE,LLL(1),IBASE
        DIMENSION NMSK(4),MASK(4)
        LOGICAL CLOSED
        INTEGER HASHC(4)
        INTEGER HASH
        INTEGER CONT
        INTEGER XLOOF
        DIMENSION IRET(9),NOPEN(9),IWHICH(9,8),NTUPL(2)
        DATA HASHC/1375243,4159,751427,4038795/
        DATA NMSK/1,3,7,15/
        DATA MASK/1,2,4,8/
        FIND = 0
        NFORM = 0
        IRET(IFOLOW) = 0
        ISTR = 17345 + N
        CLOSED = .TRUE.
        J = 0
        DO 2 I=1,N
        IF(NTUPL(I).EQ.0) GO TO 1
        NFORM = NFORM + MASK(I)
        ISTR = ISTR + NTUPL(I)*HASHC(I)
        GO TO 2
1       CLOSED = .FALSE.
        J = J + 1
        IWHICH(IFOLOW,J) = I
2       CONTINUE
        NOPEN(IFOLOW) = J
        JADRES=MOD((LCS(ISTR,23)+ISTR).AND.777777777777777B,LHASH)+1
        L = HASH(JADRES)
        IF(L.EQ.0) RETURN
        N4 = 0
        IF(CLOSED) N4 = 8
        IDSOGT = N4+N
3       IF(IDSOGT.NE.ID(L)) GO TO 6
        LOC = LNKL(L)
        I = CONT(LOC)
```

```
      IF(CLOSED) GO TO 4
      IF(ID(I).NE.NFORM) GO TO 6
      I = CONT(LNKL(I))
4     DO 5 M=1,N
      IF(NTUPL(M).EQ.0) GO TO 5
      IF(NTUPL(M).NE.NPART(I,M)) GO TO 6
5     CONTINUE
      FIND = 1
      IF(.NOT.CLOSED) IRET(IFOLOW) = LOC
      RETURN
6     LOC = LNKR(L)
      IF(LOC.EQ.0) RETURN
      L = CONT(LOC)
      GO TO 3
      ENTRY ACCESS
      L = IRET(IFOLOW)
      IF(L.NE.0) GO TO 7
      FIND = 0
      RETURN
7     L = CONT(L)
      J = CONT(LNKL(L))
      DO 8 I=1,N
C
C    MOVE OUT N SYMBOLS TO VARIABLE STORE
C
      K = NPART(J,IWHICH(IFOLOW,I))
      KCONT = CONT(K)
      LSYMB = LNKL(KCONT) - IBASE
      NCHAR = ID(KCONT)
      IF(NCHAR.LT.40000B) GO TO 8
      NCHAR = NCHAR.AND.3777B
      M = NTUPL(I)
      LWORK(M) = NCHAR
      CALL UNPACK(LLL(LSYMB),WORK(1,M),NCHAR)
8     CONTINUE
      IRET(IFOLOW) = LNKR(L)
      FIND = 1
      RETURN
      ENTRY STOPF
      IF(N.LE.4) GO TO 22
21    FIND = 0
      RETURN
22    DO 23 I=1,N
      IF(NTUPL(I).EQ.0) GO TO 21
23    CONTINUE
```

```
      NFORM = NMSK(N)
      CLOSED = .TRUE.
      IDSOGT = 8 + N
24    ISTR = 17345 + N
      DO 25 I=1,N
      KKK = NFORM.AND.MASK(I)
      IF(KKK.NE.0) ISTR = ISTR + NTUPL(I)*HASHC(I)
25    CONTINUE
      JADRES=MOD((LCS(ISTR,23)+ISTR).AND.77777777777777B,LHASH)+1
      L = XLOCF(HASH(JADRES))
      LCONT = HASH(JADRES)
      IF(LCONT.NE.0) GO TO 26
      NCELL = L
      GO TO 32
26    IF(IDSOGT.NE.ID(LCONT)) GO TO 30
      I = LNKL(LCONT)
      ICONT = CONT(I)
      JCONT = ICONT
      IF(CLOSED) GO TO 27
      IF(ID(ICONT).NE.NFORM) GO TO 30
      JCONT = CONT(LNKL(ICONT))
27    DO 28 M=1,N
      KKK = NFORM.AND.MASK(M)
      IF(KKK.EQ.0) GO TO 28
      IF(NTUPL(M).NE.NPART(JCONT,M)) GO TO 30
28    CONTINUE
      IF(CLOSED) GO TO 21
      NCELL = NUCELL(X)
      CALL STRIND(NFORM,LCOPY,LNKR(ICONT),NCELL)
      CALL STRIND(-1,-1,NCELL,I)
29    NFORM = NFORM - 1
      IDSOGT = N
      CLOSED = .FALSE.
      IF(NFORM.NE.0) GO TO 24
      FIND = 1
      RETURN
30    LNXT = LNKR(LCONT)
      IF(LNXT.EQ.0) GO TO 31
      L = LNXT
      LCONT = CONT(L)
      GO TO 26
31    NCELL = NUCELL(X)
      CALL STRIND(-1,-1,NCELL,L)
32    IF(CLOSED) GO TO 33
      NXCELL = NUCELL(X)
```

```
      CALL STRIND(IDSOGT,NXCELL,0,NCELL)
      CALL STRIND(NFORM,LCOPY,0,NXCELL)
      GO TO 29
33    LCOPY = NUCELL(X)
      CALL ST4IND(NTUPL,N,LCOPY)
      CALL STRIND(IDSOGT,LCOPY,0,NCELL)
      GO TO 29
      END
```

```
      FUNCTION LOCSYM(CHAR,NCHAR)
      COMMON/INFO/IFREE,LOC(1),IBASE
      COMMON/ASSOC/LHASH,HASH(500),NXTFRE,SYMBOL(500)
      INTEGER CHAR(2),HASH
      LOGICAL FLAG
      INTEGER CONT
      INTEGER XLOCF
      FLAG = .FALSE.
1     NAME = 0
      IDSOGT = NCHAR .OR. 4000B
      DO 2 I=1,NCHAR
2     NAME = LCS(NAME,1)+LRS(CHAR(I),54)
      ILOC=MOD((LCS(NAME,23)+NAME).AND.7777777777777B,LHASH)+1
      L = XLOCF(HASH(ILOC))
      LCONT = HASH(ILOC)
      IF(LCONT.NE.0) GO TO 5
      IF(FLAG) GO TO 3
21    LOCSYM = 0
      RETURN
3     CALL STRIND(IDSOGI,NXTFRE,0,L)
      CALL PACK(CHAR,LOC(NXTFRE-IBASE),NCHAR)
      NXTFRE = NXTFRE + (NCHAR+9)/10
4     LOCSYM = L
      RETURN
5     IF(ID(LCONT).NE.IDSOGT) GO TO 9
      ICHAR = NCHAR
      ILOC = LNKL(LCONT)
      J = 0
      K = 1
6     IF(ICHAR.LT.10) GO TO 7
      CALL PACK(CHAR(K),NAME,10)
      ICHAR = ICHAR - 10
      IF(NAME.NE.LOC(ILOC-IBASE+J)) GO TO 9
      K = K + 10
      J = J + 1
      GO TO 6
7     IF(ICHAR.EQ.0) GO TO 4
      CALL PACK(CHAR(K),NAME,ICHAR)
      IF(NAME.EQ.LOC(ILOC-IBASE+J)) GO TO 4
9     LNXT = LNKR(LCONT)
      IF(LNXT.EQ.0) GO TO 10
      L = LNXT
      LCONT = CONT(L)
      GO TO 5
10    IF(.NOT.FLAG) GO TO 21
```

```
LNXT = NUCELL(X)
CALL STRIND(-1,-1,LNXT,L)
L = LNXT
GO TO 3
ENTRY DEFSYM
FLAG = .TRUE.
GO TO 1
END
```

```
      SUBROUTINE EVAL(CHAR,N)
C
C   SUBROUTINE -EVAL- IS CALLED FROM FUNCTION -IBURY- ONLY
C
C
***NOTE(1)
C
      COMMON/KEBEG/KBEGIN,K
      COMMON/CNTROL/ECHO,TRACE,JUNK(11),INTAP,NOUTAP
     1,JJUNK(4)
      COMMON/ALGEBR/NAMES(5),SOURCE(100),SHIER(100),OHIER(100),
     1 OPSTCK(100),SAVE(80),L,NUMBSW,FLAG,TEST,MM
      DIMENSION FLOT(100)
      EQUIVALENCE(FLOT(1),SOURCE(1))
      INTEGER SOURCE,SHIER,OPSTCK,OHIER,SAVE,ARGLOC(5)
      INTEGER CHAR(100),EQUAL,ASSIGN
      INTEGER REAL(5),INTEGR(8),CALL(5)
      INTEGER PARENL,PARENR,BREAK,PLUS
      INTEGER PERIOD,QUOTE
      INTEGER TEST,SYMB(8)
      LOGICAL WRIT,FLAG,NUMBSW
      DATA REAL/1HR,1HE,1HA,1HL,1H /
      DATA INTEGR/1HI,1HN,1HT,1HE,1HG,1HE,1HR,1H /
      DATA CALL/1HC,1HA,1HL,1HL,1H /
      DATA PARENL,PARENR,EQUAL,BREAK,PLUS/1H(,1H),1H=,1H≤,1H+/
      DATA PERIOD,QUOTE/1H.,1H≠/
      DATA SYMB/1H ,1H+,1H-,1H*,1H/,1H(,1H),1H,/
C
C   THE FIRST SECTION OF CODING IN -EVAL- (UP TO STAEMENT LABEL 8)
C   DETERMINES WHETHER THE INPUT STRING IS AN INTEGER OR REAL ARRAY
C   DECLARATION, AN ASSIGNMENT STATEMENT, A SUBROUTINE CALL, OR A
C   SUBSCRIPTED OR UNSUBSCRIPTED VARIABLE NAME. DEPENDING ON WHICH
C   OF THESE IT IS, CERTAIN FLAGS ARE SET TO BE USED LATER ON IN THE
C   SUBROUTINE
C
      NSAV = 0
      ICALL = 0
      ILOC = 0
C
C   TEST FOR AN EQUALS SIGN IN THE INPUT STRING
C
      IEQ = INDEX(CHAR,N,EQUAL)
C
C   IF ONE IS NOT FOUND CONTINUE
C
```

```
      IF(IEQ.EQ.0) GO TO 1
C
C     STORE THE POSITION OF THE EQUALS SIGN FOUND IN THE STRING IN
C     -NSAV-
C
        NSAV = IEQ
C
C
C     TEST FOR AN OPENING PARENTHESIS INDICATING A SUBSCRIPTED VARIABLE
        IF(INDEX(CHAR,IEQ,PARENL).EQ.0) GO TO 6
        ASSIGN = 1
C
C     -ICHAR- IS USED AS A POINTER TO THE CURRENT CHARACTER POSITION IN
C     THE INPUT STRING
C
        ICHAR = 0
C
C     -M- POINTS TO THE LAST CHARACTER BEFORE THE EQUALS SIGN
C
        M = IEQ - 1
C
C     SINCE AN ASSIGNMENT STATEMENT WAS FOUND , NO FURTHER TESTING FOR
C     DECLARATIONS OR SUBROUTINE CALLS IS NEEDED
C
        GO TO 8
C
C     TEST FOR THE DECLARATION REAL
C
1       ILOC = INDEX(CHAR,N,REAL,5)
        IF(ILOC.EQ.0) GO TO 3
C
C     -WBIT- IS SET TO .TRUE. FOR A REAL DECLARATION AND TO .FALSE. FOR AN
C     INTEGER DECLARATION
C
        WBIT = .TRUE.
        ICHAR = ILOC + 4
2       ASSIGN = 1
        M = N
        GO TO 8
C
C     TEST FOR THE DECLARATION INTEGER
C
C
C     TEST IF THE TWO OPERANDS TO BE USED IN THE BINARY OPERATION ARE BOTH
C     INTEGERS. IF SO, GO TO THE SECTION OF CODING THAT DEALS WITH THE
```

```
C     OPERATOR ENCOUNTERED
C
        IF(SHIER(K-2).EQ.0.AND.SHIER(K-1).EQ.0) GO TO(81,82,83,84,85,86,
       187,88,89,90,91,92),ITRAN
C
C     TEST FOR A REAL NUMBER RAISED TO INTEGER POWER
C
        IF(OPSTCK(J-1).EQ.5.AND.SHIER(K-2).EQ.-1.AND.SHIER(K-1).EQ.0)
       1 GO TO 74
C
C     THE FOLLOWING LOOP DEALS WITH MIXED MODE OPERANDS. BOTH OPERANDS
C     ARE MADE REAL (IF ONE IS AN INTEGER AND THE OTHER REAL) AND
C     -ITRAN- IS SET TO DIRECT THE PROGRAM TO THE OPERATIONS PERFORMED
C     ON REALS
C
        DO 52 II=1,2
        IF(SHIER(K-II).NE.0) GO TO 52
        SHIER(K-II) = -1
        FLOT(K-II) = SOURCE(K-II)
52      CONTINUE
        ITRAN = OPSTCK(J-1) + 5
C
C     TEST IF THE TWO OPERANDS TO BE USED IN THE BINARY OPERATION ARE BOTH
C     REAL. IF SO, GO TO THE SECTION OF CODING THAT DEALS WITH THE
C     OPERATOR ENCOUNTERED
C
        IF(SHIER(K-2).EQ.-1.AND.SHIER(K-1).EQ.-1) GO TO(81,82,83,84,85,86,
       187,88,89,90,91,92),ITRAN
C
C     SINCE A TEST FOR ALL OPERATIONS HAS BEEN DONE, AN ERROR HAS
C     OCCURRED IF THE PROGRAM REACHES THIS POINT
C     SINCE ALL OPERATIONS HAVE BEEN TESTED FOR, AN ERROR AHS OCCURRED IF
C     THE PROGRAM REACHES THIS POINT
C
777     WRITE(NOUTAP,778)(CHAR(I),I=1,N)
778     FORMAT(*0ERROR IN EVALUATION OF ALGEGRAIC EXPRESSION,*80A1)
        RETURN
C
C     PROCESS A UNARY OPERATOR, A BUILT-IN FUNCTION OR A SUBSCRIPTED
C     VARIABLE
C
53      IF(OPSTCK(J-1).GT.13) GO TO 55
C
C     IF THE OPERATION BEING PERFORMED IS UNARY +, UNARY -, FLOAT, FIX,
C     OR ABS, OR THE OPERAND BEING USED IS NOT INTEGER THEN THE OPERATION
```

```
C     IS CARRIED OUT. HOWEVER IF EITHER OF THE ABOVE CASES IS NOT TRUE
C     THEN THE OPERAND IS AUTOMATICALLY CHANGED TO TYPE REAL SINCE ONE OF
C     THE OPERATIONS BEING PERFORMED ON IT (THESE INCLUDE SIN, COS, TAN,
C     ATAN, EXP, ALOG, ALOG10, OR SQRT) REQUIRES A REAL ARGUMENT
C
        IF(OPSTCK(J-1).LE.5.OR.SHIER(K-1).NE.0) GO TO 531
C
C     CHANGE THE ARGUMENT TO TYPE REAL AND CHANGE ITS CORRESPONDING
C     HIERARCHY
C
        FLOT(K-1) = SOURCE(K-1)
        SHIER(K-1) = -1
C
C     -ITRAN- DETERMINES WHICH OPERATION IS TO BE PERFORMED
C
531     ITRAN = OPSTCK(J-1)
        GO TO (61,62,63,64,65,66,67,68,69,70,71,72,73),ITRAN
C
C     THE FOLLOWING STATEMENT IS USED IN CONJUNCTION WITH THE UNARY +
C     OPERATOR (SEE STATEMENT 51)
C
54      J = J - 1
C
C     THE NEXT OPERATOR ON THE STACK IS NOW EXAMINED
C
        GO TO 47
C
C     SUBSCRIPTED VARIABLES AND SUBROUTINE CALLS ARE HANDLED HERE
C
C     -MM- IS SET TO THE SUBSCRIPTED VARIABLE BEING EXAMINED
C
55      MM = OPSTCK(J-1) - 13
        KBEGIN = ARGLOC(MM)
C
C     DETERMINE IF A SUBROUTINE CALL HAS BEEN MADE
C
        IF(I.NE.L.OR.ICALL.NE.1) GO TO 56
C
C     THE SUBROUTINE -EXECUTE- IS CALLED TO LOAD THE DESIRED SUBROUTINE
C     FROM THE COMS LIBRARY
C     THE SUBROUTINE NAME IS STORED IN -NAMES-
C     THE POSITION IN THE INPUT STRING OF THE FIRST ARGUMENT OF THE
C     CALL IS STORED IN -SOURCE(KBEGIN)-
C     THE HIERARCHY ASSIGNED TO THE FIRST ARGUMENT OF THE CALL IS STORED
C     IN -SHIER(KBEGIN)-
```

```
C     K-KBEGIN GIVES THE NUMBER OF ARGUMENTS IN THE CALL
C
        CALL EXECUTE(NAMES(MM),SOURCE(KBEGIN),FLOT(KBEGIN),SHIER(KBEGIN),
       1 K-KBEGIN)
        RETURN
C
C     PROCESS SUBSCRIPTED VARIABLES
C
56      KEND = K - 1
        K = KBEGIN + 1
C
C     UNPACK THE FIRST NAME INTO THE ARRAY -SAVE-
C
        CALL UNPACK(NAMES(MM),SAVE,10)
C
C     ANY BLANKS BETWEEN THE SUBSCRIPTED VARIABLE NAME AND THE OPENING
C     LEFT PARENTHESIS ARE OMITTED
C
        DO 57 IS=2,10
        IF(SAVE(IS).EQ.1H ) GO TO 58
57      CONTINUE
58      SAVE(IS)=PARENL
        NPOS = IS + 1
        DO 59 IS=KBEGIN,KEND
C
C     ANY INTEGER SUBSCRIPTS ARE CHANGED TO TYPE REAL
C
        IF(SHIER(IS).NE.0) SOURCE(IS)=FLOT(IS)
C
C     THE CHARACTER CODE FOR THE SUBSCRIPT VALUE IS OBTAINED AND PLACED
C     IN THE STRING STORED IN -SAVE-
C
        CALL GETCHR(SAVE(NPOS),NCHR,SOURCE(IS),1)
        NPOS = NPOS + NCHR + 1
C
C     A COMMA IS INSERTED BETWEEN EACH SUBSCRIPT AND A CLOSING RIGHT
C     PARENTHESIS PLACED AT THE END OF THE STRING
C
59      SAVE(NPOS-1) = SYMB(8)
        SAVE(NPOS-1) = PARENR
        NPOS = NPOS - 1
C
C     DETERMINE IF AN ASSIGNMENT IS TO BE MADE
C
        IF(I.NE.L.OR.ASSIGN.NE.1) GO TO 60
```

```
C
C     DETEREMINE IF AN ARRAY DECLARATION IS TO BE PROCESSED (E.G. REAL
C     XYZ(100))
C
        IF(ILOC.EQ.0) GO TO 595
C
C     AN ARRAY DECLARATION HAS BEEN ENCOUNTERED AND THUS SPACE MUST BE
C     ALLOCATED FOR IT.
C
C     -ALOCAT- IS AN ENTRY POINT IN THE SUBROUTINE -GETNL-
C
        CALL ALOCAT(SAVE,NPOS,I,J,WBIT)
C     TEST WHETHER A REAL OR INTEGER DECLARATION HAS BEEN MADE AND MOVE
C     THE CORRESPONDING CHARACTERS INTO THE ARRAY -CHAR-
C
        IF(WBIT) GO TO 591
        CALL MOVE(INTEGR,CHAR,8)
        NCHR = 9
        GO TO 592
591     CALL MOVE(REAL,CHAR,5)
        NCHR = 6
C
C     MOVE THE STRING IN -SAVE- INTO THE ARRAY -CHAR- AND RESET THE
C     NUMBER OF CHARACTERS IN THE STRING
C
592     CALL MOVE(SAVE,CHAR(NCHR),NPOS)
        N = NPOS + NCHR - 1
        RETURN
C
C     NO TYPE STATEMENT WAS MADE - WE ARE DEALING WITH THE NAME OF A
C     SUBSCRIPTED VARIABLE FOLLOWED BY AN ARGUMENT LIST
C
595     CALL MOVE(SAVE,CHAR,NPOS)
        NSAV = NPOS + 1
C
C     AN EQUALS SIGN IS PLACED IN THE STRING
C
        CHAR(NSAV) = EQUAL
C
C     THE VALUE TO BE STORED IN THIS SUBSCRIPTED VARIABLE IS NOW
C     DETERMINED
C
        GO TO 6
C
C     RETRIEVE A VALUE FROM EVALUATOR STORAGE
```

```
C
60    CALL GETNL(SAVE,NPOS,SOURCE(KBEGIN),FLOT(KBEGIN),SHIER(KBEGIN))
C
C   TEST FOR AN ERROR CONDITION
C
      IF(SHIER(KBEGIN).GE.-2) GO TO 54
      GO TO 777
C
C   UNARY OPERATIONS
C
C   UNARY PLUS
C
61    GO TO 54
C
C   UNARY MINUS FOR TYPE INTEGER AND REAL
C
62    IF(SHIER(K-1).EQ.0) GO TO 621
      FLOT(K-1) = -FLOT(K-1)
      GO TO 54
621   SOURCE(K-1) = -SOURCE(K-1)
      GO TO 54
C
C   FLOAT OPERATION
C
63    IF(SHIER(K-1).NE.0) GO TO 54
      FLOT(K-1) = SOURCE(K-1)
      SHIER(K-1) = -1
      GO TO 54
C
C   FIX OPERATION
C
64    IF(SHIER(K-1).NE.-1) GO TO 54
      SOURCE(K-1) = FLOT(K-1)
      SHIER(K-1) = 0
      GO TO 54
C
C   ABS OPERATION FOR TYPE INTEGER AND REAL
C
65    IF(SHIER(K-1).EQ.0) GO TO 651
      FLOT(K-1) = ABS(FLOT(K-1))
      GO TO 54
651   SOURCE(K-1) = IABS(SOURCE(K-1)) $ GO TO 54
C
C   BUILT-IN FUNCTIONS AND ARITHMETIC OPERATIONS ARE PROCESSED IN THE
C   FOLLOWING SECTION OF CODING
```

```
C
66    FLOT(K-1) = SIN(FLOT(K-1))          $ GO TO 54
67    FLOT(K-1) = COS(FLOT(K-1))          $ GO TO 54
68    FLOT(K-1) = TAN(FLOT(K-1))          $ GO TO 54
69    FLOT(K-1) = ATAN(FLOT(K-1))         $ GO TO 54
70    FLOT(K-1) =  EXP(FLOT(K-1))         $ GO TO 54
71    FLOT(K-1) = ALOG(FLOT(K-1))         $ GO TO 54
72    FLOT(K-1) = ALOG10(FLOT(K-1))       $ GO TO 54
73    FLOT(K-1) = SQRT(FLOT(K-1))         $ GO TO 54
74    FLOT(K-2) = FLOT(K-2)**SOURCE(K-1)                    $ GO TO 93
81    SOURCE(K-2) = SOURCE(K-2)  + SOURCE(K-1)              $ GO TO 93
82    SOURCE(K-2) = SOURCE(K-2)  - SOURCE(K-1)              $ GO TO 93
83    SOURCE(K-2) = SOURCE(K-2)  * SOURCE(K-1)              $ GO TO 93
84    SOURCE(K-2) = SOURCE(K-2)  / SOURCE(K-1)              $ GO TO 93
85    SOURCE(K-2) = SOURCE(K-2) ** SOURCE(K-1)              $ GO TO 93
86    SOURCE(K-2) = MOD(SOURCE(K-2),SOURCE(K-1))            $ GO TO 93
87    FLOT(K-2) = FLOT(K-2)  + FLOT(K-1)                    $ GO TO 93
88    FLOT(K-2) = FLOT(K-2)  - FLOT(K-1)                    $ GO TO 93
89    FLOT(K-2) = FLOT(K-2)  * FLOT(K-1)                    $ GO TO 93
90    FLOT(K-2) = FLOT(K-2)  / FLOT(K-1)                    $ GO TO 93
91    FLOT(K-2) = FLOT(K-2) ** FLOT(K-1)                    $ GO TO 93
92    FLOT(K-2) = AMOD(FLOT(K-2),FLOT(K-1))
93    K = K - 1 $ J = J - 1 $ GO TO 47
      END
```

```
      SUBROUTINE NUMBER(CHAR,NBEG,ICHAR)
C
C     SUBROUTINE -NUMBER- IS CALLED FROM SUBROUTINE -EVAL- ONLY
C
C     THE PURPOSE OF THIS SUBROUTINE IS TO:
C          (1) OBTAIN INTEGER OR REAL NUMBER REPRESENTATIONS FOR
C              CHARACTER CODED NUMERIC DATA
C          (2) DETERMINE WHICH OF THE BUILT-IN FUNCTIONS IS BEING CALLED
C              (IF ANY)
C          (3) RETRIEVE THE VALUES OF SUBSCRIPTED AND UNSUBSCRIPTED
C              VARIABLES FROM EVALUATOR STORAGE
C
      COMMON/CNTROL/ECHO,TRACE,JUNK(11),INTAP,NOUTAP
     1,JJJNK(4)
      COMMON/ALGEBR/NAMES(5),SOURCE(100),SHIER(100),OHIER(100),
     1 OPSTCK(100),SAVE(80),L,NUMBSW,FLAG,TEST,MM
      DIMENSION FLOT(100)
      EQUIVALENCE(FLOT(1),SOURCE(1))
      INTEGER SHIER,SOURCE,TEST,CHAR(2),SAVE
      LOGICAL FLAG,NUMBSW
      INTEGER FNAM
      INTEGER FNAME(12)
      DATA FNAME/3HMOD,5HFLOAT,3HFIX,3HABS,3HSIN,3HCOS,3HTAN,4HATAN,
     13HEXP,3HLOG,5HLOG10,4HSQRT/
C
C     OBTAIN THE NUMBER OF CHARACTERS PASSED TO THIS SUBROUTINE
C
      NCHAR = ICHAR - NBEG
C
C     DETERMINE IF THE VALUE OF A NUMBER IS REQUIRED
C
      IF(.NOT.NUMBSW) GO TO 3
C
C     DETERMINE IF THE VALUE REQUIRED IS INTEGER OR REAL
C
      IF(INDEX(CHAR(NBEG),NCHAR,1H.).NE.0.OR.INDEX(CHAR(NBEG),NCHAR,1HE)
     1 .NE.0) GO TO 1
      CALL GETNUM(SOURCE(L),CHAR(NBEG),NCHAR,1)
      SHIER(L) = 0
      GO TO 2
1     CALL GETNUM(FLOT(L),CHAR(NBEG),NCHAR,3)
      SHIER(L) = -1
2     L = L + 1
      FLAG = .FALSE.
      RETURN
```

```
C
C     PACK THE VARIABLE OR BUILT-IN FUNCTION NAME INTO THE VARIABLE
C     -FNAM-
C
3       CALL PACK(CHAR(NBEG),FNAM,NCHAR)
C
C     DETERMINE IF A BUILT-IN FUNCTION NAME HAS BEEN CALLED
C
        DO 5 NF=1,12
        IF(FNAM.NE.FNAME(NF)) GO TO 5
C
C     IF NF=1, THE -MOD- FUNCTION IS REQUIRED
C
        IF(NF.NE.1) GO TO 4
        SHIER(L) = 5
        SOURCE(L) = 6
        GO TO 2
4       SHIER(L) = 6
        SOURCE(L) = NF + 1
        GO TO 2
5       CONTINUE
C
C     NOW TREAT SUBSCRIPTED AND UNSUBSCRIPTED VARIABLES
C
C     TREAT SUBSCRIPTED VARAIBLES
C
        IF(TEST.NE.1H() GO TO 6
        MM = MM + 1
C
C     IF MORE THAN FIVE SUBSCRIPTED VARIABLE NAMES HAVE BEEN USED IN AN
C     EXPRESSION, AN ERROR IS SIGNALLED
C
        IF(MM.GT.5) GO TO 777
8       NAMES(MM) = FNAM
        SOURCE(L) = 13 + MM
        SHIER(L) = 6
        GO TO 2
C
C     TREAT UNSUBSCRIPTED VARIABLES
C
6       CALL GETNL(CHAR(NBEG),NCHAR,SOURCE(L),FLOT(L),SHIER(L))
        IF(SHIER(L).GE.-2) GO TO 2
C
C     THE VALUE OF THE VARIABLE COULD NOT BE FOUND IN EVALUATOR
C     STORAGE AND THUS A VALUE OF ZERO IS AUTOMATICALLY ASSIGNED TO IT
```

```
C
      WRITE(NOUTAP,7) FNAM
7     FORMAT(* THE VARIABLE *A10* HAS BEEN ASSIGNED A VALUE OF ZERO.*)
      SOURCE(L) = 0
      SHIER(L) = 0
      GO TO 2
777   WRITE(NOUTAP,778)
778   FORMAT(*0ERROR, ALGEBRAIC EXPRESSION CONTAINS MORE THAN 5 SUBSCRIP
     1TED VARIABLE NAMES.*)
      MM = 5
      GO TO 8
      END
```

```
      FUNCTION INDICES(STRING,LENGTH,DOPE)
      INTEGER STRING(1),DOPE(1),MESAGE(2)
      LOGICAL ENDSW
      DATA MESAGE/17HERROR IN INDICES./
      LCOMMA=0 $ ENDSW=.FALSE. $ INDIX=0 $ MULT=1 $ IPT=1 $ LSTCOMA=1
1     IF(ENDSW) GO TO 4
      IPT=IPT+1 $ LSTCOMA=LSTCOMA+LCOMMA
      LCOMMA=INDEX(STRING(LSTCOMA),LENGTH-LSTCOMA+1,1H,)
      IF(LCOMMA.NE.0) GO TO 2
      LCOMMA=INDEX(STRING(LSTCOMA),LENGTH-LSTCOMA+1,1H))
      IF(LCOMMA.EQ.0) GO TO 8
      ENDSW = .TRUE.
2     NMULT=MULT $ MULT=DOPE(IPT) $ IF(MULT.GT.0) GO TO 3
      IF(.NOT.ENDSW) GO TO 8
      MULT = -MULT $ GO TO 31
3     IF(ENDSW) GO TO 8
31    CALL GETNUM(I,STRING(LSTCOMA),LCOMMA-1,1)
      INDIX=INDIX+NMULT*(I-1) $ GO TO 1
4     IF(INDIX.GE.MULT.OR.INDIX.LT.0) GO TO 8
      INDICES=INDIX $ RETURN
8     CALL WRCARD(MESAGE,17) $ INDICES=-DOPE $ RETURN
      END
```

```
SUBROUTINE ALLOC(IADDR,ISIZE)
INTEGER ARRAY(1000),XLOCF
DATA IFREE/1/
IADDR=XLOCF(ARRAY(IFREE))
IFREE=IFREE+ISIZE
RETURN
END
```

```
FUNCTION FLOATER(IX)
FLOATER=IX
RETURN
END
```

```
INTEGER FUNCTION FIXER(FX)
FIXER=FX
RETURN
END
```

```
      SUBROUTINE GETNL(STRING,LENGTH,FIXD,FLOT,SHIER)
      COMMON/INFO/IFREE,ILOC(1),IBASE
      INTEGER STRING(1),FIXD,FLOT,SHIER,STODICT(105),INFO(101),JUNK(100)
     1,ARASIZ,FLOATER,FIXER,FLAGS,STOSIZ,VARLIM,XLOCF,DEFINE
      DIMENSION MESAG1(4),MESAG2(4)
      LOGICAL FIRSTM,ALOCSW,STORSW,ENDSW
      DATA JFREE,JSIZE,STOSIZ,VARLIM/1,100,101,10/
      DATA MESAG1/38HERROR IN NUMERIC STORAGE OR RETRIEVAL./
      DATA MESAG2/31HNUMERIC STORAGE HAS OVERFLOWED./
      DATA FIRSTM/.TRUE./
      DATA MARK/1H$/
      STORSW=.FALSE. $ ALOCSW=.FALSE. $ ISTRT=1 $ LNGTH=LENGTH
1     IF(.NOT.FIRSTM) GO TO 2
      FIRSTM=.FALSE. $ ISIZ=STOSIZ+4 $ STODICT(1)=STOSIZ
      DO 11 I=2,ISIZ
11    STODICT(I) = 0
2     LPAR=INDEX(STRING,LNGTH,1H() $ IF(LPAR.NE.0) LNGTH=LPAR-1
      IF(STRING(ISTRT).NE.MARK) GO TO 3
      ISTRT=ISTRT+1 $ LNGTH=LNGTH-1
3     IF(LNGTH.GT.VARLIM) LNGTH=VARLIM
      CALL PACK(STRING(ISTRT),NAME,LNGTH)
4     LOC=LOCATE(NAME,STODICT) $ IF(LOC.EQ.0) GO TO 9
      IF(ALOCSW) RETURN
      FLAGS = LRS(INFO(LOC-4),30).AND.3B
      IF(.NOT.STORSW) GO TO 5
      IF(SHIER.NE.0) GO TO 41
C
C    WE ARE DEALING WITH A FIXED POINT CONSTANT
C
      ISAVE=FIXD $ IF(FLAGS.AND.1B) ISAVE=FLOATER(FIXD) $ GO TO 5
C
C    WE ARE DEALING WITH A FLOATING POINT CONSTANT
C
41    ISAVE=FIXER(FLOT) $ IF(FLAGS.AND.1B) ISAVE=FLOT
5     INDX=INFO(LOC-4).AND.777777B
      IF(FLAGS.GT.1) GO TO 6
      IF(LPAR.NE.0) GO TO 8
      IF(.NOT.STORSW) GO TO 51
      JUNK(INDX)=ISAVE $ RETURN
51    IF(STRING(ISTRT-1).NE.MARK) GO TO 52
      FIXD=XLOCF(JUNK(INDX)) $ SHIER=-2 $ RETURN
52    IF(FLAGS.NE.0) GO TO 53
      FIXD=JUNK(INDX) $ SHIER=0 $ RETURN
53    FLOT=JUNK(INDX) $ SHIER=-1 $ RETURN
6     IF(LPAR.EQ.0) GO TO 7
```

```
      INDIX=INDICES(STRING(LPAR+1),LENGTH-LPAR,JUNK(INDX))+JUNK(INDX)
      IF(INDIX.EQ.0) GO TO 8
      IF(.NOT.STORSW) GO TO 61
      ILOC(INDIX-IBASE)=ISAVE $ RETURN
61    IF(STRING(ISTRT-1).NE.MARK) GO TO 62
      SHIER=-2 $ FIXD=INDIX $ RETURN
62    IF(FLAGS.AND.1B) GO TO 63
      SHIER=0 $FIXD=ILOC(INDIX-IBASE) $ RETURN
63    SHIER=-1 $ FLOT=ILOC(INDIX-IBASE) $ RETURN
7     SHIER=-2 $ FIXD=JUNK(INDX) $ RETURN
8     CALL WRCARD(MESAG1,38)
81    IF(.NOT.ALOCSW) SHIER=-3 $ RETURN
9     IF(ALOCSW) GO TO 100
      IF(.NOT.STORSW) GO TO 8
      IF(JFREE.GT.JSIZE) GO TO 777
      IF(LPAR.NE.0) GO TO 8
      LOC=DEFINE(NAME,STODICT)
      IF(SHIER.NE.0) GO TO 91
      JUNK(JFREE)=FIXD $ INFO(LOC-4)=JFREE $ GO TO 92
91    JUNK(JFREE)=FLOT $ INFO(LOC-4)=LLS(1,30)+JFREE
92    JFREE=JFREE+1 $ RETURN
100   IF(JFREE.GT.JSIZE) GO TO 777
      LCOMMA=LPAR $ ENDSW=.FALSE. $ ARASIZ=1$ LSTCOMA=1 $ IX=JFREE
      JFREE=JFREE+1 $ LOC=DEFINE(NAME,STODICT)
101   LSTCOMA=LSTCOMA+LCOMMA
      LCOMMA=INDEX(STRING(LSTCOMA),LENGTH-LSTCOMA+1,1H,)
      IF(LCOMMA.NE.0) GO TO 102
      LCOMMA=INDEX(STRING(LSTCOMA),LENGTH-LSTCOMA+1,1H))
      IF(LCOMMA.EQ.0) GO TO 8 $ ENDSW=.TRUE.
102   CALL GETNUM(MULT,STRING(LSTCOMA),LCOMMA-1,1)
      ARASIZ=ARASIZ*MULT
      IF(ENDSW) GO TO 103
      IF(JFREE.GT.JSIZE) GO TO 777
      JUNK(JFREE)=ARASIZ $ JFREE=JFREE+1 $ GO TO 101
103   IF(JFREE.GT.JSIZE) GO TO 777
      JUNK(JFREE)=-ARASIZ $ JFREE=JFREE+1 $ CALL ALLOC(IARRAY,ARASIZ)
      JUNK(IX)=IARRAY $ IF(.NOT.SHIER) GO TO 104
      INFO(LOC-4)=LLS(3B,30)+IX $ RETURN
104   INFO(LOC-4)=LLS(2B,30)+IX $ RETURN
      ENTRY STORNL
      IEQ=INDEX(STRING,LENGTH,1H=) $ IF(IEQ.EQ.0) GO TO 8
      ISTRT=1 $ LNGTH=IEQ-1 $ STORSW=.TRUE. $ ALOCSW=.FALSE. $ GO TO 1
      ENTRY ALOCAT
      ALOCSW=.TRUE. $ ISTRT=1 $ LNGTH=LENGTH $ GO TO 1
777   CALL WRCARD(MESAG2,31) $ GO TO 81
      END
```

```
      SUBROUTINE BUGOUT
C
C   SUBROUTINE -BUGOUT- IS CALLED FROM FUNCTION -LOAD- ONLY
C
C
***NOTE(1)
C
      COMMON LIST(4000)
      COMMON/INFO/IFREE,LOC(1),IBASE
      COMMON/ASSOC/LHASH,HASH(500),NXTFRE,SYMBOL(500)
      INTEGER XLOCF,HASH,SYMBOL
900   PRINT 901,(HASH(I),I=1,LHASH)
901   FORMAT(*1 HASH TABLE CONTAINS*/(5O22))
      PRINT 902
902   FORMAT(*1LIST CONTAINS*)
      IPLACE = XLOCF(LIST) - 1
      LAST = IFREE - IPLACE
      DO 904 I=1,LAST,5
      J = IPLACE + I
      ISTOP = I + 4
      PRINT 903,J,(LIST(K),K=I,ISTOP)
903   FORMAT(1X,O6,5O22)
904   CONTINUE
      PRINT 905
905   FORMAT(*1SYMBOL DICTIONARY CONTAINS*)
      IPLACE = XLOCF(SYMBOL) - 1
      LAST = NXTFRE - IPLACE - 1
      DO 907 I=1,LAST,10
      J = IPLACE + I
      ISTOP = I + 9
      PRINT 906,J,(SYMBOL(K),K=I,ISTOP)
906   FORMAT(1X,O6,1H,,10(A10,1H,))
907   CONTINUE
      RETURN
      END
```

```
      SUBROUTINE GETNUM(B,BUFF,MCHR,KTYP)
C
C   THIS SUBROUTINE IS CALLED FROM FUNCTION -IBODY-, FUNCTION -IPATRN-,
C   SUBROUTINE -NUMBER-, FUNCTION -INDICES- AND SUBROUTINE -GETNL-
C
C
***NOTE(1)
***NOTE(2)
C
      DIMENSION FMAT(3),TEMP(2),BUFF(2)
      DATA FMAT/5H(I20),5H(O20),8H(E20.12)/
1     FORMAT(1H(,I2,2HX,,I2,3HA1))
C   THE NUMBER OF CHARACTERS TO BE ENCODED IS STORED IN -MCHR-
C   THE NUMBER OF CHARACTERS NEEDED FOR A FULL WORD (I.E. 20 OCTAL
C   CHARACTERS) IS STORED IN -MX-
C
      MX = 20 - MCHR
C
***NOTE(3)
C
      ENCODE(10,1,XMAT) MX,MCHR
      ENCODE(20,XMAT,TEMP)(BUFF(M),M=1,MCHR)
C
C   -KTYP- DETERMINES WHICH TYPE OF OUTPUT FORMAT IS TO BE USED;
C   REAL, INTEGER OR OCTAL
C
      XMAT = FMAT(KTYP)
      DECODE(20,XMAT,TEMP) B
      RETURN
      END
```

```
      SUBROUTINE GETCHR(CHAR,NCHAR,NUMB,KTYP)
C
C   THIS SUBROUTINE IS CALLED FROM THE SUBROUTINE -EVAL- ONLY
C
C
***NOTE(1)
C
      INTEGER XMAT,CHAR(NCHAR),FMAT(3),TEMP(2),BUFF(20)
      DATA FMAT/5H(I20),5H(O20),8H(E20.12)/
C
C   DETERMINE WHICH TYPE OF FORMAT IS TO BE USED FOR ENCODING. THIS IS
C   DEPENDENT ON WHETHER THE REPRESENTATION OF THE NUMBER IS INTEGER
C   (KTYP=1),OCTAL(KTYP=2) OR REAL(KTYP=3)
C
      XMAT = FMAT(KTYP)
C
***NOTE(2)
C
      ENCODE(20,XMAT,TEMP) NUMB
      CALL UNPACK(TEMP,BUFF,20)
      DO 1 I=1,20
C
***NOTE(3)
C
      IF(BUFF(I).NE.1H ) GO TO 2
1     CONTINUE
      I = 20
2     NCHAR = 21 - I
      CALL MOVE(BUFF(I),CHAR,NCHAR)
      RETURN
      END
```

```
      SUBROUTINE EXECUTE
C
C    THIS SUBROUTINE IS CALLED FROM SUBROUTINE -EVAL-  ONLY
C
C    THE PURPOSE OF -EXECUTE- IS TO CALL THE COMPASS ROUTINE LOADIT
C    (WHICH IS USED TO DYNAMICALLY LOAD FORTRAN PROGRAMS FROM THE COMS
C    LIBRARY DURING THE EXECUTION OF COMS) AND PROVIDE IT WITH THE
C    PROPER INFORMATION (STORED IN THE COMMON BLOCKS -KEBEG- AND
C    -ALGEBR-) FOR THE GENERATION OF CALLING SEQUENCES TO THE REQUIRED
C    LIBRARY PROGRAMS
C
      COMMON/KEBEG/KBEGIN,K
      COMMON/ALGEBR/NAMES(5),SOURCE(100),SHIER(100),OHIER(100),
     1 OPSTCK(100),SAVE(80),L,NUMBSW,FLAG,TEST,MM
      DIMENSION FLOT(100)
      INTEGER SHIER,SOURCE,OPSTCK,OHIER
      EQUIVALENCE(FLOT(1),SOURCE(1))
      CALL LOADIT
      RETURN
      END
```

```
      SUBROUTINE RDCARD(CODE)
C
C     THIS SUBROUTINE IS CALLED FROM FUNCTION -LOAD- AND FUNCTION -IBODY-
C
C     -RDCARD- IS USED TO READ IN 80 COLUMNS OF INFORMATION FROM AN
C     INPUT CARD. THIS INCLUDES THE READING OF BOTH STRAN RULES AND DATA
C
      COMMON/CNTROL/ECHO,TRACE,JUNK(11),INTAP,NOUTAP
     1,JJUNK(4)
      LOGICAL ECHO
      INTEGER CODE(1)
      READ(INTAP,100)(CODE(I),I=1,8)
C
C     END OF FILE TEST
C
      IF(EOF,INTAP) 2,1
C
C     IF THE (ECHO) IS TURNED ON, THE INFORMATION READ IN IS IMMEDIATELY
C     WRITTEN OUT
C
1     IF(ECHO) WRITE(NOUTAP,101)(CODE(I),I=1,8)
      RETURN
C
C     -NOUTAP- AND -INTAP- ARE INITIALIZED IN SUBROUTINE -INITIAL-
C
2     WRITE(NOUTAP,102) INTAP
      STOP
100   FORMAT(8A10)
101   FORMAT(9H INPUT...,8A10)
102   FORMAT(*1END OF FILE READ ON INPUT TAPE*I2)
      END
```

```
      SUBROUTINE WRCARD(BUFF,LENGTH)
C
C   THIS SUBROUTINE IS CALLED FROM FUNCTION -IBODY-,FUNCTION -INDICES-
C   AND SUBROUTINE -GETNL-
C
C   THE MAIN PURPOSE OF -WRCARD- IS TO PROVIDE A ROUTINE FOR OUTPUTTING
C   ERROR MESSAGES WITHOUT HAVING TO USE -WRITE- AND -FORMAT- STATEMENTS
C   EACH TIME AN ERROR IN A PARTICULAR ROUTINE OCCURS
C
      COMMON/CNTROL/ECHO,TRACE,JUNK(11),INTAP,NOUTAP
     1,JJJNK(4)
      INTEGER BUFF(8)
C
C   STORE THE NUMBER OF ARGUMENTS IN THE CALL TO -WRCARD- (I.E. ONE
C   OR TWO) IN -NARGS-
C
      NARGS=NUMP(NARGS)
      LNGTH = 8
C
C***NOTE(1)
C
      IF(NARGS.GT.1) LNGTH = (LENGTH+9)/10
      WRITE(NOUTAP,1)(BUFF(I),I=1,LNGTH)
1     FORMAT(1X,8A10)
      RETURN
      END
```

```
      FUNCTION LOCATE(NAME,DICT)
C
C   THE FUNCTION -LOCATE- IS CALLED FROM SUBROUTINE -INTERP-, FUNCTION
C   -IBODY-, FUNCTION -IPATRN-, AND SUBROUTINE -GETNL-
C
C
***NOTE(1)
C
C   -DICT- IS AN ARRAY TO BE USED AS A DICTIONARY.  -NAME- IS THE SYMBOL
C   TO BE LOOKED UP
C   CONTENTS OF DICT(1) = LENGTH OF -DICT- MINUS 4
C   CONTENTS OF DICT(2) = TOTAL NUMBER OF ENTRIES PRESENTLY IN -DICT-
C   CONTENTS OF DICT(3) = SUM OF THE DEPTHS AT WHICH ENTRIES HAVE
C                            BEEN STORED
C   CONTENTS OF DICT(4) = MAXIMUM DEPTH AT WHICH ANY ENTRY HAS BEEN
C                            STORED
C   INITIAL CONTENTS OF DICT(5) THROUGH DICT(DICT(1)+4) = ZERO
C
      INTEGER DICT(2)
      LOGICAL FLAG
      FLAG = .FALSE.
C
C   SET -LIMIT- TO THE MAXIMUM DICTIONARY LENGTH
C
1     LIMIT = DICT(1)
C
***NOTE(2)
C
      LOC=MOD((LCS(NAME,23)+NAME).AND.7777777777777777B,LIMIT)
C
***NOTE(3)
C
      DO 5 I=1,LIMIT
C
C   SINCE THE MINIMUM VALUE OF -LOC- IS ZERO, THE FIRST AVAILABLE
C   LOCATION FOR EITHER STORAGE OR RETRIEVAL IS LOC+5
C
      IF(DICT(LOC+5).NE.0) GO TO 4
C
C   A .TRUE. FLAG INDICATES STORAGE AND A .FALSE. FLAG INDICATES
C   RETRIEVAL
C
      IF(FLAG) GO TO 2
      LOCATE = 0
      RETURN
```

```
C
C     STORE THE RULE NAME
C
2       DICT(LOC+5) = NAME
C
C     INCREMENT THE TOTAL NUMBER OF ENTRIES
C
        DICT(2) = DICT(2) + 1
        DICT(3) = DICT(3) + I
C
C     IF CURRENT DEPTH IS GREATER THAN ANY PREVIOUS DEPTH, THE VALUE OF
C     -DICT(4)- IS RESET
C
        IF(I.GT.DICT(4)) DICT(4) = I
3       LOCATE = LOC + 5
        RETURN
4       IF(DICT(LOC+5).EQ.NAME) GO TO 3
        LOC = LOC + 1
C
C     IF -LOC- EQUALS 257, EITHER THE WHOLE DICTIONARY HAS BEEN SEARCHED
C     FOR THE RETRIEVAL OF A RULE NAME, OR THE WHOLE DICTIONARY IS FULL
C     OF RULE NAMES AND NO MORE CAN BE STORED
C
        IF(LOC.EQ.LIMIT) LOC = 0
5       CONTINUE
        PRINT 100
100     FORMAT(*1DICTIONARY FULL, EXECUTION TERMINATED.*)
        STOP
C
C     FOR STORAGE OF RULE NAMES A CALL TO THE ENTRY POINT -DEFINE- IS MADE
C     AND THE FLAG IS SET TO .TRUE.
C
        ENTRY DEFINE
        FLAG = .TRUE.
        GO TO 1
        END
```

```
      FUNCTION INDEX(A,N1,B,NM2)
C
C
C   THE FUNCTION -INDEX- IS CALLED FROM FUNCTION -LOAD-, SUBROUTINE
C   -INTERP-, FUNCTION -IPATRN-, SUBROUTINE -EVAL-, SUBROUTINE
C   -NUMBER-, FUNCTION -INDICES- AND SUBROUTINE -GETNL-
C
C
***NOTE(1)
C
C   INDEX IS A FUNCTION WHICH FINDS THE FIRST POSITION IN THE CHARACTER
C   STRING A WHERE THE CHARACTER STRING B CAN BE FOUND
C   N1 IS THE LENGTH OF CHARACTER STRING A
C   N2 IS THE LENGTH OF CHARACTER STRING B
C   IF NO VALUE IS GIVEN FOR N2, IT IS ASSUMED TO BE 1
C   IF THE STRING IN -B- DOES NOT OCCUR IN THE STRING IN -A-, INDEX=0
C
      INTEGER A(2),B(2)
      N2 = 1
C
C   IF THE NUMBER OF ARGUMENTS IN THE CALL TO -INDEX- IS GREATER THAN
C   3, -N2- IS RESET TO THE VALUE OF THE FOURTH ARGUMENT
C
      IF(NUMP(X).GT.3) N2 = NM2
C
C   TEST IF THE LENGTH OF THE CHARACTER STRING BEING SEARCHED FOR IS
C   GREATER THAN THE LENGTH OF THE CHARACTER STRING BEING SEARCHED
C
      IF(N2.GT.N1) GO TO 11
C
***NOTE(2)
C
      LIMIT = N1 - N2 + 1
      DO 10 I=1,LIMIT
C
C   COMPARE EACH ELEMENT OF -A- WITH THE FIRST ELEMENT OF -B-
C
1     IF(A(I).NE.B(1)) GO TO 10
C
C   IF ONLY ONE CHARACTER IS INVOLVED THE SEARCH IS TERMINATED
C
      IF(N2.EQ.1) GO TO 6
C
C   -IM1- IS USED FOR PROGRAMMING THE FOLLOWING DO LOOP SEARCH
C
```

```
      IM1 = I - 1
      DO 5 J=2,N2
C
C   ONCE THE FIRST CHARACTERS HAVE BEEN MATCHED, SUCCEEDING CHARACTERS
C   IN -A- AND -B- ARE COMPARED SINCE EACH CHARACTER IN -B- MUST MATCH
C   EXACTLY THOSE IN -A-
C
      IF(A(IM1 + J).NE.B(J)) GO TO 10
5     CONTINUE
C
C   A SUCCESSFUL SEARCH
C
6     INDEX = I
      RETURN
10    CONTINUE
C
C   AN UNSUCCESSFUL SEARCH
C
11    INDEX = 0
      RETURN
      END
```

```
      SUBROUTINE PAGE
C
C    SUBROUTINE -PAGE- IS CALLED FROM SUBROUTINE -STRAN- ONLY
C
C
***NOTE(1)
C
      COMMON/CNTROL/ECHO,TRACE,JUNK(11),INTAP,NOUTAP
     1,JJJNK(4)
      LOGICAL FIRSTM
      DATA FIRSTM,ACPEL/ .TRUE., 0.0/
C
C    TEST IF FOR THIS PARTICULAR RUN, SUBROUTINE -PAGE- HAS BEEN CALLED
C    BEFORE
C
      IF(.NOT.FIRSTM) GO TO 1
      CALL DATE(DAY)
      IPAGE = 0
      FIRSTM = .FALSE.
C
C    -SECOND- IS A SYSTEM ROUTINE RETURNING AS ITS VALUE, IN -S-, THE
C    AMOUNT OF TIME (IN SECONDS) WHICH HAS PASSED SINCE EXECUTION OF THE
C    PROGRAM BEGAN
C
1     CALL SECOND(S)
C
C    CALCULATE THE DIFFERENCE IN TIME BETWEEN THIS CALL TO THE SUBROUTINE
C    -SECOND- AND THE LAST CALL
C
      CPEL=S-ACPEL
C
C    SET -ACPEL- TO THE CURRENT VALUE OF -S- TO BE USED NEXT TIME AROUND
C
      ACPEL=S
      IPAGE = IPAGE + 1
C
C    WRITE OUT THE TITLE WITH THE ELAPSED CP TIME AND THE PAGE NUMBER
C
      WRITE(NOUTAP,100) DAY,CPEL,IPAGE
100   FORMAT(*1STRAN INTERPRETER ON*,A10,14X,*ELAPSED CP TIME=*,F6.3,
     1 15X,*PAGE*,I3/1H0)
      RETURN
      END
```

```
      SUBROUTINE PACK(A,B,NUMB)
C
C     THE SUBROUTINE -PACK- IS CALLED FROM FUNCTION -LOAD-, SUBROUTINE
C     -INTERP-, FUNCTION -IBODY-, FUNCTION -IPATRN-, SUBROUTINE
C     -PUSH-, FUNCTION -LOCSYM-, SUBROUTINE -NUMBER- AND SUBROUTINE
C     -GETNL-
C
C
***NOTE(1)
C
      DIMENSION A(2),B(2)
      N = 80
C
C     DETERMINE THE NUMBER OF ARGUMENTS THIS ROUTINE WAS CALLED WITH
C
      IF(NUMP(X).GT.2) N = NUMB
C
C     THE FOLLOWING TEST IS NECESSARY SINCE THE ENCODING OF ZERO OR LESS
C     CHARACTERS CAUSES AN EXECUTION ERROR
C
      IF(N.LE.0)RETURN
      ENCODE(N,1,B)(A(I),I=1,N)
1     FORMAT(80A1)
      RETURN
      END
```

```
      SUBROUTINE UNPACK(A,B,NUMB)
C
C   SUBROUTINE -UNPACK- IS CALLED FROM FUNCTION -LOAD-, SUBROUTINE
C   -INTERP-, FUNCTION -IBODY-, FUNCTION -IPATRN-, FUNCTION -FIND-,
C   SUBROUTINE -EVAL- AND SUBROUTINE -GETCHR-
C
C***SEE NOTE(1) OF SUBROUTINE -PACK-
C
      DIMENSION A(2),B(2)
      N = 80
C
C   DETERMINE THE NUMBER OF ARGUMENTS IN THE CALL TO THIS ROUTINE
C
      IF(NUMP(X).GT.2) N = NUMB
C
C   THE FOLLOWING TEST IS NECESSARY SINCE THE DECODING OF ZERO OR LESS
C   CHARACTERS CAUSES AN EXECUTION ERROR
C
      IF(N.LE.0)RETURN
      DECODE(N,1,A)(B(I),I=1,N)
1     FORMAT(80A1)
      RETURN
      END
```

```
      SUBROUTINE MOVE(A,B,N)
C
C     SUBROUTINE -MOVE- IS CALLED FROM FUNCTION -IPATRN-, SUBROUTINE
C     -EVAL- AND SUBROUTINE -GETCHR-
C
      DIMENSION A(2),B(2)
C
C     MOVE THE ELEMENTS OF ARRAY -A- INTO ARRAY -B-
C
      DO 1 I=1,N
1     B(I) = A(I)
      RETURN
      END
```

```
INTEGER FUNCTION CONT(N)
COMMON/INFO/IFREE,LOC(1),IBASE
CONT = LOC(N-IBASE)
RETURN
END
```

```
INTEGER FUNCTION LNKR(N)
LNKR = N.AND.7777773
RETURN
END
```

```
INTEGER FUNCTION LNKL(N)
LNKL = LRS(N,30).AND.777777B
RETURN
END
```

```
INTEGER FUNCTION ID(N)
ID = LRS(N,48)
RETURN
END
```

```
SUBROUTINE STRIND(I,J,K,N)
COMMON/INFO/IFREE,LOC(1),IBASE
INTEGER CONT
II = I
IF(II.EQ.-1) II = ID(CONT(N))
JJ = J
IF(JJ.EQ.-1) JJ = LNKL(CONT(N))
KK = K
IF(KK.EQ.-1) KK = LNKR(CONT(N))
JJ = JJ.AND.777777B
LOC(N-IBASE) = LLS(II,48).OR.LLS(JJ,30).OR.(KK.AND.777777B)
RETURN
END
```

```
      SUBROUTINE ST4IND(NTUPL,N,NDEST)
      COMMON/INFO/IFREE,LOC(1),IBASE
      DIMENSION NTUPL(2)
      IPAT = 0
      DO 1 I=1,N
      NEXT = NTUPL(I).AND.77777B
1     IPAT = IPAT .OR. LLS(NEXT,15*(4-I))
      LOC(NDEST-IBASE) = IPAT
      RETURN
      END
```

```
INTEGER FUNCTION NPART(N,I)
NPART = LRS(N,15*(4-I)).AND.77777B
RETURN
END
```

```
      INTEGER FUNCTION NUCELL(X)
      COMMON/INFO/IFREE,LOC(1),IBASE
      IF(IFREE.EQ.0) GO TO 10
      NUCELL = IFREE
      IFREE = LOC(IFREE-IBASE)
      RETURN
10    PRINT 11
11    FORMAT(* FREE LIST EXHAUSTED.*)
      STOP
      END
```

```
SUBROUTINE RCELL(N)
COMMON/INFO/IFREE,LOC(1),IBASE
LOC(N-IBASE) = IFREE
IFREE = N
RETURN
END
```

```
          IDENT   PRIMS
          LIST    F
*PRIMS IS A SET OF BASIC SUBROUTINES USED BY THE PRECEEDING FORTRAN
*ROUTINES
*   THESE ROUTINES CAN BE USED WITH EITHER THE RUN OR FTN COMPILERS.
*   A SWITCH SELECTS THE PROPER SECTION OF CODING FOR THE PARTICULAR
*   COMPILER BEING USED
*
         ENTRY      NUMP,LCS,LLS,LRS
*
*   NUMP IS A ROUTINE THAT CAN BE USED BY ANY FORTRAN SUBROUTINE OR
*   FUNCTION TO DETERMINE THE NUMBER OF ARGUMENTS IN THE CALL TO THAT
*   SUBROUTINE OR FUNCTION
*    THE NUMBER OF ARGUMENTS CAN BE RETRIEVED IN TWO WAYS.
*                   (1)   NARGS = NUMP(DUMMY)
*                   (2)   CALL NUMP(NARGS)
*   IN BOTH CASES THE NUMBER OF ARGUMENTS WILL BE CONTAINED IN NARGS.
L00002    VFD       42/0HNUMP,18/1
NUMP      DATA    0
          IFEQ    *F,2,37        SELECT FTN OR RUN COMPILER
*   FTN COMPILER:
          SA2     NUMP           GET ADDRESS OF CALL TO NUMP
          MX0     6              CREATE MASK IN X0
          BX2     -X0*X2         MASK LOWER 42 BITS
          AX2     30             SHIFT TO ADDRESS POSITION
          SA2     X2-1           GET CONTENTS OF TRACEBACK WORD
          MX3     42             CREATE MASK IN X3
          BX2     -X3*X2         MASK LOWER 6 BITS
          SA2     X2+2           ADDRESS OF ENTRY POINT WORD OF SUBROUTINE
*                                CONTAINING CALL TO NUMP
          BX2     -X0*X2         MASK LOWER 42 BITS
          AX2     30             SHIFT TO ADDRESS POSITION
          SA2     X2-2           GET ADDRESS OF ARGUMENT ADDRESS TABLE
*                                USED BY THE ROUTINE THAT CALLED THE
*                                ROUTINE CONTAINING THE CALL TO NUMP
          SB1     -15            SET B1 TO THE CONSTANT -15
          MX3     18             CREATE MASK IN X3
          LX3     3              PLACE MASK IN LOWER 3 BITS OF X3
          SX4     51100B         CREATE SA1 B0+K INSTRUCTION IN X4
          LX4     3              LEFT SHIFT X4 3 BITS
          MX5     12             CREATE MASK IN X5
          LX5     15             PLACE MASK IN LOWER END OF X5
LOOP1     SB1     B1+15          LOOP TO DETERMINE WHICH PART OF THE WORD
*                                CONTAINS THE SA1 INSTRUCTION
          LX3     15             RESPECTIVE MASKS IN X3, X4 AND X5 ARE
```

```
*                                      SHIFTED IN PREPARATION FOR EXTRACTION OF
*                                      THE ADDRESS OF THE ARGUMENT ADDRESS TABLE
         LX4      15
         LX5      15
         BX6      X5*X2                MASK OUT BITS 3 TO 14 OF X5
         IX6      X6-X4                SUBTRACT TO SEE IF NEEDED INSTRUCTION
*                                      PRESENT
         ZR       X6,FIND              IF ZERO INSTRUCTION HAS BEEN FOUND
         EQ       LOOP1                SEARCH FOR INSTRUCTION CONTINUES
FIND     BX2      X2*X3                MASK OUT ADDRESS OF ARGUMENT TABLE
         AX2      B1                   RIGHT SHIFT CONTENTS OF B1 INTO X2
         SB1      B0                   SET B1 TO ZERO
         SA2      X2-1                 PREPARE ADDRESS IN A2 FOR THE FOLLOWING
*                                      LOOP
BACK     SA2      A2+1                 OBTAIN NEXT ADDRESS IN ARGUMENT TABLE
         ZR       X2,ZWORD             ARGUMENT TABLE SEARCHED FOR ZERO WORD
*                                      WHICH INDICATES ITS TERMINATION
         SB1      B1+1                 INCREMENT COUNTER
         EQ       BACK                 REPEAT SEARCH
ZWORD    SX6      B1                   NUMBER OF ARGUMENTS PLACED IN X6
         SA6      X1                   NUMBER OF ARGUMENTS PLACED IN X1
         SKIP     10                   SKIP NEXT 10 INSTRUCTIONS
*  RUN COMPILER:
        SA1         NUMP               GET ADDRESS OF CALL TO NUMP
        AX1         30                 SHIFT TO ADDRESS POSITION
        SA1         X1-1               GET CONTENTS OF CALL TO NUMP
        SA1         X1+1               GET ADDRESS OF CALL TO CALLING SUBROUTINE
        AX1         30                 SHIFT TO ADDRESS POSITION
        SA1         X1-1               GET CONTENTS OF CALL TO CALLING SUBROUTINE
        AX1         18                 SHIFT TO GET ARGUMENT COUNT
        MX6         54                 CREATE MASK FOR ARG COUNT
        BX6         -X6*X1             USE AND OPERATION TO GET THE ARGUMENT COUNT
        SA6         B1                 STORE COUNT IN NUMPS ARGUMENT
         EQ       NUMP
* LCS DOES ORDINARY LEFT CIRCULAR SHIFT.
LCS     DATA        0
         IFEQ     *F,2,6
*  FTN COMPILER:
         SA3      X1
         SA1      A1+1
         SA2      X1
         SB2      X2
         LX6      B2,X3
         SKIP     4
*  RUN COMPILER:
```

```
         SA2         B2
         SA1         B1
         SB2         X2
         LX6         B2,X1
         EQ          LCS
* LLS DOES LEFT END-OFF SHIFT
LLS      DATA        0
          IFEQ     *F,2,11
*  FTN COMPILER:
          SA3      X1
          SA1      A1+1
          SA2      X1
          SB2      X2
          LX6      B2,X3
          SA4      59
          SB2      A4-B2
          MX0      1
          AX0      B2,X0
          BX6      X0*X6
          SKIP     9
*  RUN COMPILER:
         SA2         B2
         SA1         B1
         SB2         X2
         LX6         B2,X1
         SA0         59
         SB2         A0-B2
         MX0         1
         AX0         B2,X0
         BX6         X0*X6
         EQ          LLS
* LRS DOES RIGHT END-OFF SHIFT.
LRS      DATA        0
          IFEQ     *F,2,10
*  FTN COMPILER:
          SA3      X1
          SA1      A1+1
          SA2      X1
          SB2      X2
          AX6      B2,X3
          SB2      B2-1
          MX0      1
          AX0      B2,X0
          BX6      -X0*X6
          SKIP     8
```

```
*   RUN COMPILER:
          SA2       B2
          SA1       B1
          SB2       X2
          AX6       B2,X1
          SB2       B2-1
          MX0       1
          AX0       B2,X0
          BX6       -X0*X6
          EQ        LRS
           END
```

```
          IDENT LOADIT
*
*
*   -LOADIT- IS A COMPASS SUBROUTINE CALLED FROM SUBROUTINE -EXECUTE-
*   THAT GENERATES A CALL TO A USER DEFINED SUBROUTINE WHICH IS LOCATED
*   ON A FILE SEPERATE FROM THE COMS PROGRAM
*   LINKAGE OF ARGUMENTS IS ALSO PERFORMED - ALL ARGUMENTS ARE PASSED BY
*   ADDRESS (I.E. THE ADDRESS OF A TEMPORARY LOCATION CONTAINING THE
*   PRESENT VALUE OF THE ARGUMENT)
*   NOTE  (1) THE FILE ON WHICH THE USER DEFINED SUBROUTINE IS TO BE
                LOCATED IS CALLED -COMSLIB-
*         (2) A USER REQUEST TO THE LOADER IS PERFORMED BY -LOADIT-
*               THROUGH -LDREQ- MACROS UNDER SCOPE 3.4 REVISION C. ANY
*   CHANGE IN THE EXPANSION OF THESE MACROS MAY CAUSE -LOADIT- TO FAIL
*
*
          LIST   S
          ENTRY  LOADIT
          VFD    42/0HLOADIT,18/0
LOADIT    DATA   0
*
*   /ALGEBR/ CONTAINS THE ARRAYS -NAMES(5)-, -SOURCE(100)- AND
*   -PHIER(100)-
*
          USE    /ALGEBR/
BL1       BSS    205
          USE    *
*
*   /KEBEG/ CONTAINS THE VARIABLES -KBEGIN- AND -K-
*
          USE    /KEBEG/
BL2       BSS    2
          USE    *
*
*   THE MACRO  WRITE  IS PROVIDED FOR A CONVENIENCE TO THE USER FOR ERROR
*   TRACEBACK IF DESIRED
*
WRITE     MACRO  J,R.
          EXT    RIGHT
          SB1    J
+         RJ     RIGHT
-         VFD    12/1701B,18/R0-1
          ENDM
*   THE MACRO  FTAB  EXAMINES EACH OF THE FIRST SI  ARGUMENTS IN THE CALL A
*   DETERMINES WHETHER EACH IS BEING PASSED BY DIRECT OR INDIRECT
```

```
*  ADDRESS. IF BY INDIRECT ADDRESS, THE DIRECT ADDRESS IS OBTAINED AND
*  TRANSFERRED TO THE PROPER LOCATION EXPECTED BY THE SUBROUTINE. IF BY
*  DIRECT ADDRESS, THE ADDRESS IS PASSED AS IS TO THE PROPER LOCATION
*  EXPECTED BY THE SUBROUTINE
*
SETAB     MACRO  J
          IRP    J
          SX0    2
*
*  PICK UP THE VALUE IN -SHIER(J)- AND PUT IT IN X4
*
          SA4    SHIER+J
*
*  ADD 2 TO THIS VALUE
*
          IX0    X0+X4
*
*  DETERMINE WHETHER THE VALUE IN -SHIER(J)- IS -2 OR NOT. IF IT IS -2
*  THEN THE ARGUMENT IS BEING PASSED BY INDIRECT ADDRESS. IF NOT, IT IS
*  BEING PASSED BY DIRECT ADDRESS
*
          NZ     X0,SB→J
*
*  SET X2 TO THE CONTENTS OF -SOURCE(J)- WHICH IS THE ADDRESS OF THE
*  TEMPORARY CONTAINING THE PRESENT VALUE OF THE ARGUMENT BEING
*  CONSIDERED
*
          SA2    SOURCE+J
*
*  SET THE CORRESPONDING B REGISTER TO THIS ADDRESS
*
          SB→J   X2
          EQ     LABL→J
*
*  SET THE CORRESPONDING B REGISTER TO THE CONTENTS OF -SOURCE(J)-
*
SB→J      SB→J   SOURCE+J
*
*  X3 CONTAINS THE NUMBER OF ARGUMENTS BEYOND THE SIXTH IN THE USER
*  DEFINED SUBROUTINE
*
LABL→J    SX3    X1-J
*
*  TEST IF ALL OF THE ARGUMENTS IN THE CALL HAVE BEEN CONSIDERED
*
```

```
          ZR     X3,FIN
          IRP
          ENDM
*
*  THE ADDRESS  BL1  REFERS TO -NAMES(1)-, BL1+5 REFERS TO -SOURCE(1)-
*  AND BL1+105 REFFRS TO -SHIER(1)-
*
SOURCE    SET    BL1+4
SHIER     SET    BL1+104
*
*  X1 CONTAINS THE NAME OF THE USER DEFINED SUBROUTINE BEING CALLFD
*  (LEFT JUSTIFIED WITH BLANK FILL)
*
          SA1    BL1
          BX6    X1
*
*  A LOCATION -SNAME- IS CREATED TO REFERENCE THE USER DEFINED
*  SUBROUTINE NAME
*
          SA6    =SSNAME
*
*  THE FOLLOWING SECTION OF PROGRAMMING UP TO LABEL  DONE  REPLACES THE
*  BLANK FILL IN X1 BY ZERO FILL
*
          SB3    24
          SB1    55B
*
*  PLACE NAME WITH BLANK FILL INTO X2
*
          SA2    SNAME
*
*  CREATE A MASK IN X0 TO PULL OUT CHARACTERS
*
LOOP1     MX0    6
*
*  LEFT CIRCULAR SHIFT MASK BY B3 BITS AND PLACE IT IN X0
*
          LX0    B3,X0
*
*  MASK OUT THE FIRST CHARACTER INTO X3
*
          BX3    X2*X0
          SB4    B3-6
*
*  RIGHT END OFF SHIFT CHARACTER IN X3 BY B4 BITS LEAVING IT THE LOWER 6
```

```
*  BITS OF X3
*
         AX3     B4
         SB2     X3
*
*  TEST IF CHARACTER IS A BLANK
*
         NE      B1,B2,DONE
*
*  IF A BLANK THEN REPLACE IT WITH 6 BITS OF ZEROS AND REPEAT THE
*  PROCEDURE UNTIL A BLANK IS NOT FOUND
*
         SB3     B3+6
         BX2     -X0*X2
         EQ      LOOP1
*
*  X6 WILL CONTAIN THE USER DEFINED SUBROUTINE NAME WITH ZERO FILL
*  (EXCLUDING THE LOWER 18 BITS OF THE WORD SINCE THESE ARE NOT NEEDED
*  ANYWAYS)
*
DONE     BX6     X2
*
*  TRANSFER THIS NAME TO THE NECESSARY WORDS USED BY THE  LDREQ  MACROS
*
         SA6     ENTR+1
         SA6     ENTRA+2
*
*  USER REQUEST TO THE LOADER IS INITIATED
*
         LOADER PADDR
*
*  X1=CONTENTS OF VARIABLE -K-
*  X2=CONTENTS OF VARIABLE -KBEGIN-
*
         SA1     BL2+1
         SA2     BL2
*
*  K-KBEGIN  GIVES THE NUMBER OF SUBROUTINE ARGUMENTS WITH WHICH THE
*  USER CALL WAS ISSUED
*
         IX1     X1-X2
         SETAB   (1,2,3,4,5,6)
*
*  X2 IS SET THE ENTRY POINT ADDRESS OF THE NOW LOADED USER DEFINED
*  SUBROUTINE
```

```
*
          SA2     ENTR+1
*
*  X4 IS SET TO THE ADDRESS OF THE VFD ERROR TRACEBACK WORD OF THE ABOVE
*  SUBROUTINE
*
          SX4     X2-1
*
*  X2 IS SET TO THE ADDRESS WHERE THE SEVENTH ARGUMENT WILL BE PLACED
*
          IX2     X4-X3
*
*  THE SECTION OF CODING UP TO -FIN- PLACES THE ADDRESS OF THE SEVENTH
*  ARGUMENT, EIGHTH ARGUMENT AND SO ON INTO CONSECUTIVE LOCATIONS
*  BEFORE THE ENTRY POINT OF THE SUBROUTINE BEING CALLED (ACCORDING
*  TO RUN-COMPASS FORTRAN LINKAGE)
*
          SX3     6
LOOP2     SX3     X3+1
*
*  THE FOLLOWING SIX STATEMENTS ARE ALMOST IDENTICAL TO THE MACRO
*  -SETAB- CODING AND PERFORM THE SAME JOB
*
          SX0     2
          SA5     X3+SHIER
          IX0     X0*X5
          NZ      X0,SBX
          SA1     X3+SOURCE
          BX6     X1
*
*  THE CONTENTS OF X6 ARE PLACED AT THE PROPER ADDRESS THAT THE CALLED
*  SUBROUTINE WILL REFERENCE
*
          SA6     X2
          EQ      SKIP
SBX       SX6     X3+SOURCE
          SA6     X2
*
*  THE NEXT CONSECUTIVE ADDRESS IS OBTAINED
*
SKIP      SX2     X2+1
          IX5     X2-X4
*
*  TEST IF ALL THE ARGUMENTS OF THE CALL HAVE BEEN TAKEN CARE OF
*
```

```
         NZ      X5,LOOP2
*
*  X1 CONTAINS THE CODING FOR A RETURN JUMP INSTRUCTION IN THE UPPER 30
*  BITS OF THE WORD
*
FIN      SA1     RJ
*
*  X2 CONTAINS THE NAME OF THE USER DEFINED SUBROUTINE TO BE CALLED
*
         SA2     ENTR+1
         SX6     X2
*
*  LEFT CIRCULAR SHIFT X6 BY 30 BITS
*
         LX6     30
*
*  X6 CONTAINS A RETURN JUMP INSTRUCTION TO THE NAME OF THE USER DEFINED
*  SUBROUTINE
*
         BX6     X6+X1
*
*  PLACE THE ABOVE FORMED INSTRUCTION INTO LOCATION -CALL-
*  TIMING MAKES IT NECESSARY FOR TWO INSTRUCTIONS TO BE THE SAME
*
         SA6     CALL
         SA6     CALL
*
*  AT THIS POINT A RETURN JUMP IS CARRIED OUT AND THE USER DEFINED
*  SUBROUTINE WILL BE EXECUTED. UPON RETURN TO -LOADIT- THE PROGRAM
*  IMMEDIATELY RETURNS TO SUBROUTINE -EXECUTE
*
CALL     DATA    0
         EQ      LOADIT
RJ       RJ      **
*
*  THE FOLLOWING ARE LOAD MACROS USED IN USER INITIATED LOADING DURING
*  EXECUTION
*
PADDR    LDREQ   BEGIN
*
*  MAP,X   GIVES A COMPLETE MAP OF ALL THE ENTRY POINTS
*
         LDREQ   MAP,X
*
*  SLOAD   LOADS THE ROUTINE WHOSE NAME IS STORED IN -SNAME- FROM THE
```

```
*  FILE -COMSLIB-
*
ENTRA      LDREQ  SLOAD,COMSLIB,(SNAME)
*
*  SATISFY  IS USED TO SATIFY UNSATIFIED EXTERNALS DURING THE USER LOAD
*
           LDREQ  SATISFY,(RUN2P3)
*
*  ENTRY  RETURNS THE ENTRY POINT OF THE USER LOADED ROUTINE
*
ENTR       LDREQ  ENTRY,(SNAME)
           LDREQ  END
           END
```

HADB070 //// END OF LIST ////

APPENDIX G

CONTROL CARDS FOR RUNNING A COMS JOB

DECK(1)

```
JOB CARD
ATTACH(COMSLIB,COMSLIB,ID=COMSPROG,PW=A)
RUN(S)
COMPASS(S=LDRTEXT)
REWIND,OUTPUT.
LGO.
END OF RECORD
[ COMS FORTRAN PROGRAM
END OF RECORD
[ LOADIT COMPASS PROGRAM
END OF RECORD
[ STRAN PROGRAM STATEMENTS
END OF FILE
```

Note (1) With the RUN-FORTRAN compiler a COMPASS program which requires the loading of a system library (in this case the LDRTEXT library which contains loader macros), must be assembled with a separate control card.

Note (2) The above method of running a COMS job is simple but is dependent on the reading of almost 3000 cards. The experienced CDC 6400 user should set up a permanent file to hold the object code in the following way:

DECK(2)

```
JOB CARD
REQUEST,LGO,*PF.
RUN(S)
CATALOG(LGO,COMS,ID=COMSPROG,RP=999,XR=B)
END OF RECORD
 COMS FORTRAN PROGRAM
END OF FILE
```

With the above deck, running a job entails:

DECK(3)

```
JOB CARD
ATTACH(COMSLIB,COMSLIB,ID=COMSPROG,PW=A)
ATTACH(Y,COMS,ID=COMSPROG,PW=B)
COMPASS(S=LDRTEXT)
LOAD(X)
LGO.
END OF RECORD
 LOADIT COMPASS PROGRAM
END OF RECORD
 STRAN PROGRAM STATEMENTS
END OF FILE
```

Note (3) If any namelist reads are done from COMS library programs, the namelist data is placed amongst the STRAN program statements.

Note (4) Decks (1) and (2) assume the program library has been previously read in and cataloged according to the deck set-ups shown in Figure 1.6.

Note (5) If and when COMS is adapted to run under the FTN-FORTRAN compiler, the system library LDRTEXT can be loaded without a separate COMPASS control card as shown below:

DECK(4)

```
JOB CARD
ATTACH(COMSLIB,COMSLIB,ID=COMSPROG,PW=A)
FTN(S=LDRTEXT)
REWIND,OUTPUT.
LGO.
END OF RECORD
 [ COMS FORTRAN PROGRAM

   LOADIT COMPASS PROGRAM ]
END OF RECORD
 [ STRAN PROGRAM STATEMENTS ]
END OF FILE
```

APPENDIX H

A Sample Coroutine Program

The program shown below is based on the flowcharts given in Figures 2.2 and 2.3. It is written in Fortran with an accompanying assembly language routine COR used to handle the bilateral coroutine linkage. The data used is shown following the program. As can be seen, implicit switching is accomplished by the position of calls made from one coroutine to another. The reader is not to be mislead by the use of "subroutine" header cards, as these exist only for the benefit of the Fortran compiler and do not imply subroutine type linkage at all. The program is almost self explanatory through its comments but particular sections of COR needing more detailed discussion have note references to the material below.

Details of the Compass Subroutine COR

Note (1)

The compass subroutine COR can presently handle linkage for two Fortran coroutines. This number is easily extended by making changes to two cards of the program. The first is the ENTRY card which specifies the number of entry points for coroutines in COR. To increase this number to say eight from the present two, one would replace the blank between COR2 and COR3 in the current ENTRY card with a comma, and replace the comma between COR8 and COR9 with a blank. The second card to change is the call to the macro BUILD which presently looks like

```
BUILD CORTN,(1,2),3,4,5,6,7,8,9,10
```

For eight coroutines this would be changed to

```
BUILD CORTN,(1,2,3,4,5,6,7,8),9,10
```

The only limit on the number of coroutines one could use is the available core of the machine.

Note (2)

The macro BUILD is used when a call to another coroutine is made from inside the coroutine currently being executed. Its job is to obtain the name of the calling coroutine and the address in this coroutine following the call. This address is then stored in the block SAVE where it will be picked up by any succeeding calls made from other coroutines to the current coroutine.

Note (3)

The coding in the next few statements has the following effect. SA2 X1-1 will place the address of the actual call to this coroutine in A2, and the word containing the call in X2. It is important to note that a CALL statement (i.e. the 60 bit word) generated by the Fortran-RUN compiler contains the following code:

1) a return jump (01) to the called routine in the upper 30 bits of the word

2) a jump instruction (which is never executed) to the header address of the routine the call was made from (i.e. the address of the VFD word containing the routine's name) in the lower 30 bits of the word.

Thus the instruction SA2 X2 will place the lower 18 bits (because it is a "set" instruction) of the CALL word in A2 (these bits contain the address of the VFD word) and the contents of the VFD word in X2. In particular, if say coroutine 2 was called from coroutine 1, the octal contents of X2 will be

```
X2 = 03172224163400000000
      C O R T N 1
```

At this point a mask is formed in the lower six bits of X0 and the contents of register X2 is right shifted 24 bits resulting in

```
X2 = 00000000031722241634
              C O R T N 1
```

The last six bits containing the coroutine number are masked in X0 for the purpose of setting up a word in the block SAVE which will hold the return address to the calling coroutine if a call is made to it by any other coroutine.

The point here is that the program, because it is leaving one coroutine to enter another, must remember where to re-enter this coroutine and have this information available in a common location for all other coroutines to reference. Using the particular example above, the re-entry address, currently in X1, is placed into X6 by SX6 X1 and thence placed in the first word of the block SAVE by SA6 REF-28+X0 (where X0 = $34_8 = 28_{10}$).

Data cards used for the coroutine program:

```
DECLRTN...**DIMENSION A(5),B(100)**
DECLRTN...**INTEGER A,B****COMMON A**
ASSIGN...**IZ=5096****Y=3.2****Q=1.763****B(1)=(Y+Q)**
CALLSUB...**CALL XYZ(A,B,Y)**
CALC...**SUM=B(5)+SQRT(Y*IZ)-A(1)**
INPUT...**READ(5,2)C,D,QTIME**
FORMAT...**2FORMAT(3F20.6)**
DATA...**5.6,18.7,20.92****ENDDATA**
CALLFUNC...**CALL F(C)**
END...**ENDOFPROGRAM**
```

```
                PROGRAM CORTN(INPUT,OUTPUT,TAPE5=INPUT,TAPE6=OUTPUT)
          C
          C     THIS PROGRAM SHOWS A POSSIBLE IMPLEMENTATION OF COROUTINE LINKAGE
          C     USING AN ASSEMBLY PROGRAM CALLED -COR-
          C
000003          COMMON CARD(80),I
          C
000003          WRITE(6,10)
000007    10    FORMAT(1H1)
          C     -I- IS USED AS AN INPUT POINTER TO EACH CHARACTER READ FROM A CARD
          C
000007          I=81
          C
          C     -INIT- IS AN ENTRY POINT IN THE ROUTINE -COR- USED FOR SETTING UP
          C     THE INITIAL RETURN ADDRESS OF EACH COROUTINE
          C
000010          CALL INIT
          C
          C     BEGIN COROUTINE EXECUTION
          C
000011          CALL CORTN1
000012          STOP
000014          END

UNUSED COMPILER SPACE
010500
```

```
                SUBROUTINE CORTN1
          C
          C    -CORNT1- REFERS TO COROUTINE -SQUISH- WHERE EACH PAIR OF ADJACENT
          C    ASTERISKS FOUND IN THE INPUT STRING ARE REPLACED BY AN UPWARD ARROW
          C
          C    ALL CHARACTERS ENCOUNTERED ARE SENT TO THE WRITING COROUTINE
          C    (-CORTN2-) BUT ONLY THOSE BETWEEN TWO CONSECUTIVE UPWARD ARROWS
          C    ARE OUTPUT
          C
          C
000002          COMMON CARD(80),I,C1,C2
000002          INTEGER CARD,C1,C2,STAR,EQUIV,ARROW
000002          DATA STAR,EQUIV,ARROW/1H*,1H≡,1H↑/
          C
          C    CALL FOR THE NEXT CHARACTER OF THE INPUT STRING STORED IN -CARD-
          C
000002    10    CALL RDCARD
000003          C1=CARD(I)
          C
          C    AN EQUIVALENCE CHARACTER ≡ SIGNALS THE END OF THE INPUT STRING
          C
000005          IF(C1.EQ.EQUIV) GO TO 50
000007          I=I+1
          C
          C    TEST FOR AN ASTERISK
          C
000010          IF(C1.EQ.STAR) GO TO 20
          C
          C    -COR2- IS AN ENTRY POINT IN THE ASSEMBLY LANGUAGE ROUTINE -COR-
          C    (AS IS -COR1-).  BECAUSE OF THE LINKAGE INVOLVED, CALLS TO OTHER
          C    COROUTINES MUST BE MADE VIA -COR-.  THUS -COR2- REALLY MEANS -CORTN2-
          C
000012    40    CALL COR2
000013          GO TO 10
000014    20    CALL RDCARD
000015          C2=CARD(I)
000017          I=I+1
```

```
              C
              C    IF ONE ASTERISK IS FOUND, A SECOND IS TESTED FOR IN -C2-
              C
              C    IF THE CHARACTER IN -C2- IS NOT AN ASTERISK. IT IS TRANSFERRED
              C    TO -C1-
              C
000021               IF(C2.EQ.STAR) GO TO 30
000023               CALL COR2
000024               C1=C2
000026               CALL COR2
000027               GO TO 10
              C
              C    TWO ASTERISKS HAVE BEEN FOUND SO -C2- IS SET TO AN UPWARD ARROW
              C
000030        30     C1=ARROW
000031               CARD(I-1)=ARROW
000033               GO TO 40
              C
              C    SINCE WE ARE DEALING WITH COROUTINES HERE, A RETURN WILL ACT JUST
              C    LIKE A STOP - IN FACT IF A RETURN IS HIT IN EITHER COROUTINE AN
              C    AUTOMATIC STOP TAKES PLACE.  THIS IS TAKEN CARE OF BY -COR-
              C
000034        50     RETURN
000035               END

UNUSED COMPILER SPACE
010400
```

```
                SUBROUTINE CORIN2
000002          COMMON CARD(80),I,C1,C2
000002          INTEGER CARD,C1,C2,ARROW,EQUIV,BAR
000002          INTEGER OUT(120)
000002          DATA ARROW,EQUIV,BAR/1H↑,1H≡,1H⁻/
          C
          C    THE RETURN STATEMENT (I.E. A STOP COMMAND) WASN≠T NEEDED IN THIS
          C    COROUTINE BUT THE FOLLOWING DUMMY STATEMENT HAD TO BE PUT IN TO
          C    SATISFY THE RUN COMPILER.  -I- WILL NEVER BE ZERO
          C
000002          IF(I.EQ.0) GO TO 70
          C
          C    -J- IS A POSITION POINTER FOR THE OUTPUT LINE
          C
000003    60    J=1
          C
          C    THE OUTPUT LINE -OUT- IS INITIALIZED TO BLANKS
          C
000004          DO 10 L=1,120
000006    10    OUT(L)=1H
000012    30    CALL COR1
000013          IF(CARD(I-2).EQ.ARROW) GO TO 20
          C
          C    TEST FOR THE FIRST OF TWO ARROW CHARACTERS.  IF ONE EXISTS, ALL CALLS
          C    TO -CORIN1- ARE MADE FROM WITHIN A LOOP WHICH OUTPUTS THE CHARACTERS
          C    RETURNED
          C
000016          IF(C1.EQ.ARROW) GO TO 20
000017          GO TO 30
000020    20    CALL COR1
          C
          C    TEST FOR THE SECOND ARROW CHARACTER
          C
000021          IF(C1.EQ.ARROW) GO TO 40
          C
          C    PLACE -C1- IN THE OUTPUT LINE
          C
000023          OUT(J)=C1
```

```
000025             J=J+1
000026             GO TO 20
              C
              C    A SECOND ARROW WAS ENCOUNTERED SO AN EQUIVALENCE CHARACTER IS PLACED
              C    IN THE OUTPUT LINE TO SIGNAL ITS TERMINATION
              C
000027        40   OUT(J)=EQUIV
              C
              C    PRINT THE OUTPUT LINE
              C
000031             WRITE(6,50) OUT
000037        50   FORMAT(1X,120A1)
000037             CARD(I-1)=BAR
              C
              C    START THE PROCESS OVER
              C
000041             GO TO 60
000042        70   RETURN
000043             END
```

```
              SUBROUTINE RDCARD
         C
         C    -RDCARD- IS USED AS A SUBROUTINE IN THIS PROGRAM TO READ THE INPUT
         C    STRINGS
         C
000002        COMMON CARD(80),I
000002        INTEGER CARD
         C
         C    IF THE INPUT POINTER EQUALS 81 A NEW CARD IS READ INTO THE ARRAY
         C    -CARD-
         C
000002        IF(I.EQ.81) GO TO 10
000004        RETURN
000005   10   READ(5,40) (CARD(J),J=1,80)
000017   40   FORMAT(80A1)
000017        WRITE(6,50) (CARD(K),K=1,80)
000031   50   FORMAT(1H0,*INPUT CARD=    *,80A1)
000031        I=1
000032        RETURN
000033        END

UNUSED COMPILER SPACE
010500
```

```
          IDENT COR
****NOTE(1)
          LIST D,A,M,E
*
*    EACH OF THE FOLLOWING ENTRY POINTS CORRESPOND TO THE COROUTINE(I.E.
*   .CORTN) BEING CALLED IN THE FORTRAN PROGRAM
*
          ENTRY COR1,COR2 COR3,COR4,COR5,COR6,COR7,COR8,COR9,COR10
          ENTRY    INIT
+         VFD 42/0HCOR,18/0
*
*    THE BLOCK -SAVE- IS USED TO STORE THE RE-ENTRY ADDRESS IN THE
*    CURRENT COROUTINE BEING EXECUTED
*
          USE SAVE
REF       BSS        0
          USE        *
*
*    THE BLOCK -LATER- IS USED BY BOTH THE ROUTINE -INIT- AND THE MACRO
*    -BUILD-
*
          USE LATER
+         VFD        42/0HINIT,18/0
*
*    -INIT- IS RESPONSIBLE FOR OBTAINING THE ADDRESS IN THE MAIN
*    PROGRAM WHERE ANY COROUTINE EXECUTING A RETURN STATEMENT WILL
*    TRANSFER CONTROL
*
INIT      DATA       0
*
*    X1 WILL CONTAIN A JUMP (04) TO THE INSTRUCTION IN THE MAIN PROGRAM
*    FOLLOWING THE CALL TO -INIT-
*
          SA1        INIT
*
*    RIGHT END OFF SHIFT X1 30 BITS
*
```

```
          AX1       30
*
*   SET X6 TO THE CONSTANT 1
*
          SX6 1
*
*   THE NUMBER IN X6 RESULTING FROM THE ADDITION WILL BE THE ADDRESS
*   OF THE WORD FOLLOWING THE CALL TO THE FIRST COROUTINE IN THE MAIN
*   PROGRAM (I.E. IN THIS CASE,THE ADDRESS OF THE -STOP- INSTRUCTION)
*
          IX6 X6+X1
*
*   LEFT CIRCULAR SHIFT X6 30 BITS
*
          LX6       30
          USE *
****NOTE(2)
BUILD     MACRO SUB,P
          IRP       P
          USE SAVE
*
*   INITIALLY,FOR THE FIRST CALL TO ANY COROUTINE,THE ADDRESS OF THE
*   FIRST WORD TO BE EXECUTED (I.E. THE WORD IMMEDIATELY FOLLOWING
*   THE ENTRY/EXIT LINE) IS PLACED IN -STR-. THIS ADDRESS IS CHANGED BY
*   THE MACRO -BUILD- FOR ALL SUCCEEDING CALLS TO THE COROUTINE
*
STR→P     VFD 60/=X→SUB→P+1
          USE *
COR→P     DATA 0
*
*   X1 WILL CONTAIN A JUMP (04) TO THE INSTRUCTION IN THE CALLING
*   COROUTINE FOLLOWING THE CALL STATEMENT
*
          SA1 COR→P
*
*   RIGHT END OFF SHIFT X1 30 BITS
*
          AX1 30
```

```
****NOTE(3)
          SA2 X1-1
          SA2 X2
          MX0 6
          LX0 6
          AX2 24
          BX0 X0*X2
          SX6        X1
          SA6 REF-28+X0
*
*   THE WORD CONTAINING THE RE-ENTRY ADDRESS IN THE CALLED COROUTINE
*   IS PLACED IN X1. THE CONTENTS OF THIS WORD WAS PLACED THERE BY THE
*   PRECEEDING TEN INSTRUCTIONS AS DESCRIBED IN NOTE(2). THAT IS,STR1,
*   STR2,STR3,.... REFER TO THE SAME WORDS OF THE BLOCK -SAVE- AS
*   REF-28+X0 WHERE X0=1,2,3,....
*
          SA1 STR→P
*
*   THE CONTENTS OF X1 IS PLACED IN B1 AND A JUMP MADE TO THIS ADDRESS
*   (THE MOST RECENT ENTRY POINT OF THE CALLED SUBROUTINE)
*
          SB1 X1
          JP B1
*
*   THE ADDRESS CALCULATED BY THE ROUTINE -INIT- (I.E. THE CONTENTS OF
*   X6) IS STORED IN THE ENTRY/EXIT WORD OF EVERY COROUTINE IN USE.
*   THIS IS NECESSARY SINCE THE PROGRAM DOES NOT KNOW WHICH COROUTINE
*   WILL EXECUTE A RETURN STATEMENT FIRST-THUS ALL OF THEM NEED BE
*   PREPARED
*
          USE LATER
          SA6        =X→SUB→P
          USE *
          IRP
          ENDM
          BUILD CORTN,(1,2) ,3,4,5,6,7,8,9,10)
          IRP        1,2                                          BUILD    .1
          USE SAVE                                                BUILD    .1
STR→1     VFD 60/=X→CORTN→1+1                                     BUILD    .1
```

```
STR1      VFD 60/=XCORTN1+1                         BUILD   .1
           USE *                                    BUILD   .1
COR→1      DATA 0                                   BUILD   .1
COR1      DATA 0                                    BUILD   .1
           SA1 COR→1                                BUILD   .1
           SA1 COR1                                 BUILD   .1
           AX1 30                                   BUILD   .1
           SA2 X1-1                                 BUILD   .1
           SA2 X2                                   BUILD   .1
           MX0 6                                    BUILD   .1
           LX0 6                                    BUILD   .1
           AX2 24                                   BUILD   .1
           BX0 X0*X2                                BUILD   .1
           SX6       X1                             BUILD   .1
           SA6 REF-28+X0                            BUILD   .1
           SA1 STR→1                                BUILD   .1
           SA1 STR1                                 BUILD   .1
           SB1 X1                                   BUILD   .1
           JP B1                                    BUILD   .1
           USE LATER                                BUILD   .1
           SA6       =X→CORTN→1                     BUILD   .1
           SA6       =XCORTN1                       BUILD   .1
*                                                   BUILD   .1
           USE *                                    BUILD   .1
           USE SAVE                                 BUILD   .1
STR→2      VFD 60/=X→CORTN→2+1                      BUILD   .1
STR2      VFD 60/=XCORTN2+1                         BUILD   .1
           USE *                                    BUILD   .1
COR→2      DATA 0                                   BUILD   .1
COR2      DATA 0                                    BUILD   .1
           SA1 COR→2                                BUILD   .1
           SA1 COR2                                 BUILD   .1
           AX1 30                                   BUILD   .1
           SA2 X1-1                                 BUILD   .1
           SA2 X2                                   BUILD   .1
           MX0 6                                    BUILD   .1
           LX0 6                                    BUILD   .1
           AX2 24                                   BUILD   .1
           BX0 X0*X2                                BUILD   .1
```

```
          SX6        X1                                                    BUILD     .1
          SA6 BFF-28+X0                                                    BUILD     .1
          SA1 STR+2                                                        BUILD     .1
          SA1 STR2                                                         BUILD     .1
          SB1 X1                                                           BUILD     .1
          JP B1                                                            BUILD     .1
          USE LATER                                                        BUILD     .1
          SA6        =X+CORTN+2                                            BUILD     .1
          SA6        =XCORTN2                                              BUILD     .1
 ,                                                                         BUILD     .1
          USE *                                                            BUILD     .1
          IRP                                                              BUILD     .1
          ENDM                                                             BUILD     .1
          USE LATER
          EQ INIT
          USE *
 *
 *
 *
 DEFAULT SYMBOLS DEFINED BY COMPASS.

 CORTN1
 CORTN2

          END

 STORAGE USED             171 STATEMENTS        8 SYMBOLS
 MODEL 73 ASSEMBLY      0.744 SECONDS          21 REFERENCES
```

```
INPUT CARD=    DECLRIN...**DIMENSION A(5),B(100)**
DIMENSION A(5),B(100)≡

INPUT CARD=    DECLRIN...**INTEGER A,B****COMMON A**
INTEGER A,B≡
COMMON A≡

INPUT CARD=    ASSIGN...**IZ=5096****Y=3.2****Q=1.763****B(1)=(Y+Q)**
IZ=5096≡
Y=3.2≡
Q=1.763≡
B(1)=(Y+Q)≡

INPUT CARD=    CALLSUB...**CALL XYZ(A,B,Y)**
CALL XYZ(A,B,Y)≡

INPUT CARD=    CALC...**SUM=B(5)+SQRT(Y*IZ)-A(1)**
SUM=B(5)+SQRT(Y*IZ)-A(1)≡

INPUT CARD=    INPUT...**READ(5,2)C,D,QTIME**
READ(5,2)C,D,QTIME≡

INPUT CARD=    FORMAT...**2FORMAT(3F20.6)**
2FORMAT(3F20.6)≡

INPUT CARD=    DATA...**5.6,18.7,20.92****ENDDATA**
5.6,18.7,20.92≡
ENDDATA≡

INPUT CARD=    CALL FUNC...**CALL F(C)**
CALL F(C)≡

INPUT CARD=    END...**END OF PROGRAM**≡
END OF PROGRAM≡

                    HADB04X  //// END OF LIST ////
```