LBL-11847 c. 2
TR-804

NRCC NATIONAL RESOURCE FOR COMPUTATION IN CHEMISTRY

RATMAC Primer

October 1980

LAWRENCE BERKELEY LABORATORY
UNIVERSITY OF CALIFORNIA

Prepared for the U.S. Department of Energy under Contract W-7405-ENG-48 and for the National Science Foundation under Interagency Agreement CHE-7721305

LBL-11847 c. 2

TR-804 October 1980

RATMAC Primer

by

R. J. Munn

J. M. Stewart

A. P. Norden

M. Katherine Pagoaga

Division of Agricultural and Life Sciences
University of Maryland
College Park, Maryland 20742

National Resource for Computation in Chemistry
Lawrence Berkeley Laboratory
University of California
Berkeley, California 94720

Acknowledgements

Partial support for this manual was provided by the National Science Foundation under Grant CHE-7721305.

Additional support for the project was provided by the NRCC through the Basic Energy Sciences Division of the U.S. Department of Energy under Contract W-7405-ENG-48.

The computer time for this project was supported through the facilities of the Computer Science Center of the University of Maryland.

This document has been prepared using TEXT, a machine independent document processing system, which is written in RATMAC. TEXT is available from the Computer Science Center, University of Maryland, College Park. The features of TEXT are described in CN-26 which is available from the Program Library at the Computer Science Center.

Printing History

First edition, August 1979

Second edition, October 1980

Disclaimer

The RATMAC preprocessor is based on two programs, RATFOR and MACRO, described in the book _Software Tools_ by B. W. Kernighan and P. J. Plauger (Addison-Wesley Publishing Company). The user of RATMAC is urged to purchase and read this excellent text. In addition to describing the syntax of both RATFOR and MACRO, the book contains many excellent software weapons that should be in any programmer's armory.

RATMAC is a combination of RATFOR and MACRO. Although RATMAC has been exhaustively tested as a unit, no warranty, expressed or implied, is made by the current authors as to the accuracy and functioning of RATMAC, its subprograms, related program material, or operating instructions.

No responsibility is assumed by the authors in connection with the use, attempted use, or application of these programs.

It would be appreciated if acknowledgement of the use of RATMAC be made in published work. The main reference should be to Kernighan and Plauger with a minor acknowledgement to the current authors.

TABLE OF CONTENTS

Introduction

FORTRAN is one of the oldest and most widely available high level programming languages. Its popularity as a scientific programming language has meant that its structure has been rather static with dramatic changes occurring only after considerable discussion and thought.

On the other hand, software development, in general, has proceeded rapidly in the last decade and FORTRAN has tended to look increasingly "old fashioned". In particular, it lacks many of the control structures associated with structured programming. A number of attempts have been made to augment the ANSI standard FORTRAN with appropriate control and other structures. The resulting FORTRAN dialects are often called structured or rational FORTRAN.

Structured programming is a style of programming that has become increasingly popular in the last decade. The seminal letter entitled "GO TO statement considered harmful" CACM 11,3 (March 1968) p. 147 by E. W. Dijkstra is considered by many to have been the catalyst of this popularity. By December 1974 ACM Computing Surveys printed a review issue on this subject.

In a narrow sense, structured programming is programming with a limited well-defined set of flow control constructs. Each construct has a single entrance and a single exit. This latter statement means that the GO TO statement, so beloved of FORTRAN devotees, is not among the constructs available to the programmer. This has led to the somewhat simplistic definition of structured programming as "GO TO-less programming".

In a broader sense, structured programming is a discipline. It encourages the programmer to design in a top-down sense with care being taken to modularize and isolate functions. In addition, it promotes well-defined data structures with carefully thought out interfaces between functions.

The net result in all cases is code that is easy to write initially, easy to modify subsequently, and easier for someone unfamiliar with the details of the code to understand. It is hard to give a non-trivial example that illustrates all of these characteristics.

The program below, which will be developed later in complete detail, is meant to simulate a basic calculator that has a stack and operates with reverse Polish notation. When presented with a string of operands (real numbers) and (binary) operators, the program evaluates the expression. For example, the string:

```
3.0  (enter)  2.0    +    10.    /    =
```

would produce the result 0.5, i.e. ((3+2)/10=0.5).

The "program" is:

```
get first term in expression          # initialize

WHILE (not the end of expression)     # start WHILE loop
   {
   IF (term is an operand)            # operand?
      push the operand on the stack
   ELSE IF (term is an operator)      # operator?
      {
      IF (too few values on stack)    # error
         output message     ("Poorly formed expression")
      ELSE                            # OK evaluate
         {
         pop operands from stack
         perform operation
         push result onto the stack
         }
      }
   ELSE
      output message                  # error
      ("Illegal item in expression")
   get next term in expression
   }
STOP
```

The structured programming constructs in the above program are the "WHILE" loop, the "IF...ELSE IF...ELSE" decision structures, and the statement block definition braces, {...}.

The single entrance-single exit concept of structured programming is illustrated by the "WHILE" loop. It is entered at a single point; its associated statement block is executed repeatedly so long as the entrance condition remains true. It is exited from a single point when the entrance condition becomes false.

In a similar fashion, the IF...ELSE IF...ELSE chain is entered at the top and control passes down the chain until a true condition is found. The associated statement block is then executed. If no true condition is found, then the statement block associated with the trailing ELSE is executed.

In both constructs, the program flow is "obvious" and well-defined.

The above "program" illustrates the top-down design concept. The problems of how the expression is accessed, how the stack is constructed, and what an output message does have been pushed down to a lower design level. These are details which are not important at the top level; they clutter the design. Of course, they will have to be dealt with eventually. At that point, however, further details of the program may be pushed down to an even lower level. In this way a set of "primitive" functions can often be established that can be used over and over in a variety of programming situations.

Structured FORTRAN dialects, while powerful, have a serious drawback in that they are not transportable directly from machine to machine. To overcome this difficulty, the designer of such a dialect has to supply a processor that will translate the dialect into standard ANSI FORTRAN. In the interests of inter-machine portability, such a processor is written in ANSI FORTRAN. A bootstrap process can then be used to install the processor on any machine with an ANSI FORTRAN compiler.

The use of a structured FORTRAN language obviously involves at least one additional step in the "compilation" process. The structured language has to be translated into ANSI FORTRAN before it can be compiled into machine code.

```
************                 *************
*          *                 *           *
*  RATMAC  *     RATMAC      *   ANSI    *
*  SOURCE  * -------------->*  FORTRAN  *
*   CODE   *  preprocessor   *           *
*          *                 *           *
************                 *************
```

This extra step is easily justified. The quality of the code that is written in a structured FORTRAN will, in general, be demonstrably superior to the equivalent code written in standard FORTRAN. By superior, it is meant that the code is easier to write initially, easier to debug, easier to maintain and document, and is of comparable efficiency. In addition, once a structured FORTRAN language is adopted, the user is not limited to just adding new control structures. Features such as macros and file inclusion can also be added.

One of the most successful structured FORTRAN languages is RATFOR, rational FORTRAN, produced by Brian W. Kernighan and P. J. Plauger. This success can be traced to a number of factors. First, it is a well-defined language which is easy to learn. Second, it has a bootstrap which is easy to implement on any machine with a FORTRAN compiler. Third, it has associated with it the excellent text, Software Tools, by Kernighan and Plauger, which contains an invaluable collection of well-documented programs written in RATFOR.

The language RATMAC is a direct descendant of RATFOR. RATMAC has all of the characteristics of RATFOR, but is augmented by a powerful recursive macro processor which is extremely useful in generating transportable FORTRAN programs. A macro is a collection of programming steps which are associated with a keyword. This keyword uniquely identifies the macro, and whenever it appears in a RATMAC program it is replaced by the collection of steps. This can be a powerful programming tool as will be shown below.

For example, the macro definition:

```
MACRO:(LOGTST:,ASC&XIN<=$1|!ASC&XIN>=$1)
```

can be used in the following fashion:

```
IF(LOGTST:(XTAB(7)))
```

The string LOGTST: is recognized as a macro name, and XTAB(7) is recognized as a macro argument. The net result is LOGTST: is replaced by its definition and $1 is replaced by the first (and only) argument.

The result is equivalent to coding:

```
IF(ASC&XIN<=XTAB(7)|!ASC&XIN>=XTAB(7))
```

or the more standard:

```
          IF(ASC.AND.XIN.LE.XTAB(7).OR.
          .NOT.ASC.AND.XIN.GE.XTAB(7))
```

It is convenient to consider the RATFOR features of RATMAC separately from the macro features. In fact, one set of features can be used without making use of the other set. In what follows, the enhanced control structures will be dealt with before the macro facilities are considered.

In this primer it is assumed that the reader has made use of a conventional ANSI FORTRAN compiler and is familiar with the syntax of FORTRAN.

Control Structures

Statement Blocks

One characteristic of structured languages is that they contain few statement labels and fewer GO TO´s. The absence of such features to group executable statements means that an alternative mechanism must be found. In RATMAC this is accomplished with the use of statement brackets. All statements within a pair of statement brackets are regarded as a single unit. In RATMAC, the statement brackets are left and right braces { }. Unfortunately, not all character sets contain such characters. In such an event, two digraph equivalents $(and $) are used in place of the braces. The following piece of RATMAC code contains two explicit statement groups:

```
{
X=ANS
ANS=0.0              # zero answer
}
{
X=-17.5*X; AND=-2.0
}
I=I+1                # increment counter
```

The code also contains an implied statement block I=I+1. An implied statement block is defined as a single statement not within statement brackets.

In general, angular brackets, < and > will be used to denote syntactic units within RATMAC. In this primer, the construct <statement block> will denote a general statement block.

The above RATMAC code looks much like standard FORTRAN code with minor embellishments. However, some cosmetic features should be mentioned immediately.

1. RATMAC is free form; statements may appear anywhere on an input line.

2. Any line may contain a comment. Any characters following a hash mark, #, are ignored.

3. Multiple statements may apear on a line; a semicolon is used to separate them.

Looping Constructs

RATMAC contains four looping constructs; they are the DO, FOR, WHILE, and REPEAT loops.

DO Loop

The DO loop in RATMAC is very similar to the familiar DO loop of standard FORTRAN. Its syntax is:

```
DO <loop control>
   <statement block>
```

For example, a piece of RATMAC code to sum N numbers stored in array A would be:

```
SUM=0.0                 # initialize sum
DO J=1,N
  SUM=SUM+A(J)          # accumulate sum
```

This code is very reminiscent of the corresponding FORTRAN code, only the DO statement label is missing. The range of the loop is the statement block; in this case, the single statement SUM=SUM+A(J).

Consider now the slightly more complex code for accumulating the sums necessary to do a least squares analysis on two vectors, X and Y, of length N. The RATMAC code would be:

```
SUMX=0.0; SUMY=0.0; SUMXY=0.0
SUMX2=0.0; SUMY2=0.0
DO K=1,N
   {
   SUMX=SUMX+X(K); SUMY=SUMY+Y(K)
   SUMX2=SUMX2+X(K)*X(K); SUMY2=SUMY2+Y(K)*Y(K)
   SUMXY=SUMXY+X(K)*Y(K)
   }
```

Here the DO statement block encompasses the five statements delimited by statement brackets.

The RATMAC DO has all of the properties (and deficiencies) of the FORTRAN DO. It will be as general <u>or</u> as limited as the DO allowed by a given FORTRAN compiler. For example, to sum backwards we could write the following code if a negative DO increment is allowed by the compiler:

```
SUM=0.0
DO J=N,1,-1
   SUM=SUM+A(J)
```

If a negative increment is not allowed, RATMAC will process the statement as written but a FORTRAN failure will result.

A very important syntactic restriction concerning RATMAC follows: FORTRAN does not consider the blank character significant. Thus:

```
D     O     J=1,N
```

is equivalent to

```
DOJ=1,N
```

RATMAC is not so permissive. A keyword such as DO <u>must not contain</u> imbedded blanks and it must end in a non-alphabetic character; such as a blank. Thus:

```
DO     J   =   1   ,   N
```

is allowed while

```
DOJ=1,N
```

is not allowed.

Note: The positioning of the statement brackets, {}, following a DO loop structure is <u>restricted</u>. The opening statement bracket may <u>not</u> appear on the same line as the DO command. This restriction is necessary since there is no general way of detecting the end of the <loop control> unit.

FOR Loop

The FOR loop of RATMAC is a powerful looping construct with none of the limitations associated with the familiar DO loop. The syntax of the FOR loop is:

```
FOR(<initialization>;<condition>;<reinitialization>)
    <statement block>
```

A simple example that simulates the standard DO statements first shown above, would be:

```
SUM=0.0
FOR(I=1; I<=N; I=I+1)
   SUM=SUM+A(I)
```

The loop initialization statement is I=1; the condition under which the loop is executed is I<=N; and the reinitialization that is made at the end of each loop is I=I+1.

While this example apparently simulates the equivalent DO loop, it does not in one extremely important case; namely, that when N is zero or negative. In such a case the DO loop is always executed once, whereas the FOR is not executed at all. In addition, on exiting such a "counting" FOR loop, the index I is well-defined.

If the FOR loop were limited to such "counting" loops it would not represent a significant improvement over the DO loop. The power of the FOR loop lies in the fact that the three components: <initialization>, <condition>, and <reinitialization> can be anything that the user chooses. For instance:

```
J=NSTART
FOR(A=-3.5; A<=27.2; A=A+0.1)
   {
   B(J)=A*C(J)
   J=J+1
   }
```

As another example, consider the problem of finding the position of the first and the last non-blank characters in a line of 80 A1 characters. The following RATMAC, used in conjunction with a FORTRAN 66 compiler, does the trick:

```
          INTEGER BLANK,LINE(80),BEGIN,END
          DATA BLANK/' '/   # blank A1 character
          FOR(BEGIN=1; BEGIN<=80 & LINE(BEGIN)==BLANK;
             BEGIN=BEGIN+1)
             ;              # empty statement

          FOR(END=BEGIN; END<=80 & LINE(END)!=BLANK;
             END=END+1)
             ;
          END=END-1         # point END to non-blank
```

A number of points are worthy of note. First, the FOR statement is not complete on a single line. RATMAC detects this (it is looking for a balancing right parenthesis) and takes appropriate action. Second, the condition has become a complicated logical statement; one part to stop at the end of the line and the other to stop on an appropriate character. Third, the statement within the statement block is empty (signified by the semicolon). Fourth, the logical operator has been replaced by the following more evocative representations:

&	is	.AND.
==	is	.EQ.
>=	is	.GE.
>	is	.GT.
<=	is	.LE.
<	is	.LT.
!=	is	.NE.
!	is	.NOT.

Other representations of the remaining logical operators will appear shortly. It is not, however, mandatory that these symbols be used instead of the dotted logical operators of FORTRAN.

The reader should study the above example carefully. Note that the boundary cases are taken care of quite naturally; they are the all blank card, BEGIN=81 and END=80 and the card with no blank characters, BEGIN=1 and END=80.

The code, however, does have a flaw. It will fail if the text on the card contains imbedded blanks. The following code corrects the flaw, is simpler, and will execute more quickly:

```
          FOR(BEGIN=1; BEGIN<=80 & LINE(BEGIN)==BLANK;
              BEGIN=BEGIN+1)
            ;
          FOR(END=80;END>0 & LINE(END)==BLANK;
              END=END-1)
            ;
          #
```

While this code works correctly, it still contains a flaw which makes the code non-standard. Can you spot the flaw? Does BEGIN reaching 81 and END reaching 0 cause any problem? What would LINE contain if BEGIN=81 and END=0 after execution?

The problem lies in the logical condition in the FOR loop. ANSI FORTRAN does not specify the order of evaluation of such an expression. Thus, when BEGIN reaches 81 the test LINE(81)==BLANK may be performed before the test BEGIN<=80. Since LINE(81) is not defined, this may lead to program failures on some machines.

Consider now a more realistic problem. LINE(1) through LINE(80) contains a set of A1 characters and we wish to know how many "words" the card contains and where each "word" begins and ends. A "word" is a sequence of non-blank characters.

```
          INTEGER BLANK,BEGIN(40),END(40),LINE(80)
          INTEGER I,J
          DATA BLANK/' '/
          BEGIN(1)=1         # initialize first value
          FOR(I=1; BEGIN(I)<=80; I=I+1)       # count words
             {
             FOR(J=BEGIN(I); J<=80 & LINE(J)==BLANK;J=J+1)
                 ;           # null
             BEGIN(I)=J      # save start of word
             FOR( ;J<=80 & LINE(J)!=BLANK;J=J+1)
                 ;
             END(I)=J-1      # save end of word
             BEGIN(I+1)=J+1  # initialize next cycle
                             # since LINE(J) is blank
                             # we can skip it
             }               # end of I loop
          I=I-1              # get correct count
```

WHILE Loop

The WHILE loop is related to the FOR loop and each can be defined in terms of the other. The syntax of the WHILE loop is:

```
WHILE(<condition>)
  <statement block>
```

<statement block> is executed repeatedly while <condition> is true. It is entirely equivalent to the FOR statement:

```
FOR( ;<condition>; )
    <statement block>
```

Consider the following problem - a file, NFILE, contains an unknown number of card images followed by an end-of-file. We want to count and print the images. The following code using the WHILE construct and designed for FORTRAN 77 will perform the task:

```
CHARACTER LINE(80)
INTEGER KNT,NFILE,I,NOUT
LOGICAL EOF
DATA KNT/0/,EOF/.FALSE./  # initialize
DATA NFILE/10/,NOUT/6/    # sample values
WHILE(!EOF)               # do until EOF flips
   {
   READ(NFILE,1,END=99)(LINE(I),I=1,80) # read line
   1  FORMAT(80A1)
   WRITE(NOUT,2)(LINE(I),I=1,80)  # NOUT is printer
   2  FORMAT(1X,80A1)
   KNT=KNT+1              # count line
   GO TO 98               # skip over EOF code
   99  EOF=!EOF           # flip EOF
   WRITE(NOUT,3) NFILE,KNT        # write sign off
   3  FORMAT('0 FILE',I2,'CONTAINS',I5,'LINES')
   98  CONTINUE
   }                      # end WHILE loop
STOP
END
```

Some notes: The character ! is the rational representation of the logical operator .NOT.. Statement labels are allowed in RATMAC but are rarely required; FORMAT and the end-of-file statement are notable exceptions. The numbers on a FORMAT or labelled statement may appear anywhere on a line. The label must be terminated by a non-alphanumeric character such as a blank.

REPEAT Loop

The final loop construct in RATMAC is the REPEAT loop which has two variants. The syntax for each is as follows:

```
          REPEAT
            <statement block>
and
          REPEAT
            <statement block>
          UNTIL(<condition>)
```

The former is an "infinite" loop. <statement block> will be executed indefinitely (in practice such an infinite loop will be broken in some manner to be shown later). The latter loop is a conditional loop. It differs from the WHILE loop in that the exiting condition test is made at the end of the loop. The body of a REPEAT loop is always executed at least once.

The previous example can be rewritten using the REPEAT statement.

First, with the "infinite" REPEAT (we do not repeat the initialization code or format statements):

```
          REPEAT         # do forever!
             {
             READ(NFILE,1,END=99)(LINE(I),I=1,80)
             WRITE(NOUT,2)(LINE(I),I=1,80)
             KNT=KNT+1
             GO TO 98
             99 WRITE(NOUT,3) NFILE,KNT
             STOP        # stop infinite loop
             98  CONTINUE
             }           # end REPEAT loop
          END
```

Using the REPEAT...UNTIL code, the program becomes:

```
          REPEAT
             {
             READ(NFILE,1,END=99)(LINE(I),I=1,80)
             WRITE(NOUT,2)(LINE(I),I=1,80)
             KNT=KNT+1
             GO TO 98
             99  WRITE(NOUT,3) NFILE,KNT
                 EOF=!EOF   # flag end-of-file
             98  CONTINUE
             }
          UNTIL(EOF)
          STOP
          END
```

Modifying Loop Structures

One of the powerful features of top-down structured programming is that control structures have a single entrance and a single exit. This feature makes it extremely easy to follow the flow of control in a program. Occasionally, however, it is convenient to modify a loop structure either by exiting the loop other than by the usual <condition> test or by terminating a particular loop iteration prematurely.

RATMAC provides two commands to do this; they are BREAK and NEXT, respectively. These commands do violence to the basic principles of structured code and should be used sparingly if at all.

BREAK

The BREAK statement causes an immediate exit from a loop. The loop may be a FOR, DO, REPEAT, or WHILE structure. The BREAK exits from a single loop structure; to exit from a series of nested loops, a series of BREAK's, one within the scope of each loop, is required.

NEXT

The NEXT statement causes the current cycle of a loop structure to be terminated and the next one to be initiated. Simplistically, it can be regarded as an immediate transfer to the end of the loop enclosing the command.

Examples of the use of the BREAK and NEXT commands will be deferred until the decision control structures have been introduced.

Decision Control Structures

The basic structure that controls the flow of a RATMAC program is the construct:

```
IF (<condition>)
  <true block>
ELSE
  <false block>
```

<condition> is a logical condition that can assume the values to be true or false. The statement block <true block> is executed if the condition is true, otherwise the statement block <false block> is executed. Many varients of the basic control structure are available. If there is no ´false´ branch to be executed, then the ELSE may be omitted:

```
IF (<condition>)
  <true block>
```

The statement blocks may themselves contain subordinate control structures. The following construct is a common one:

```
IF (<condition1>)
   <block1>
ELSE IF (<condition2>)
   <block2>
ELSE IF (<condition3>)
   <block3>
  .
  .
  .
ELSE
    <block>
```

This basic construct can be regarded as a linear multi-way branch.

Example: Assume once again that if given a line of alphabetic character the number of letters and digits, the number of blanks, and the number of other characters is to be counted. Since it is not desired to be constrained to a particular character code, the existence of an integer function, ICHAR, will be assumed. This function returns a unique integer corresponding to an A1 character. The program to do this task in FORTRAN 66 would be:

```
#
#  define storage
#
INTEGER LINE(80)                    #
INTEGER ACHAR,ZCHAR,ZERO,NINE       #
INTEGER KALPHA,KBLANK,KOTHER        #
INTEGER BLANK
DATA KALPHA/0/,KBLANK/0/,KOTHER/0/
DATA BLANK/' '/
DATA ACHAR/'A'/,ZCHAR/'Z'/,ZERO/'0'/,NINE/'9'/
FOR (I=1; I<=80; I+I+1)
   IF (LINE(I)==BLANK)              # blank?
      KBLANK=KBLANK+1
   ELSE IF(ICHAR(LINE(I))>=ICHAR(ACHAR)& # letter or digit?
        ICHAR(LINE(I))<=ICHAR(ZCHAR)|    # | is logical .OR.
        ICHAR(LINE(I))>=ICHAR(ZERO)&
        ICHAR(LINE(I))<=ICHAR(NINE))
      KALPHA=KALPHA+1
   ELSE
      KOTHER=KOTHER+1                    #other
```

The reader may be concerned at this point about the scope of the FOR loop. Are some statement block defining brackets required? The answer is no. From the point of view of RATMAC, the whole IF...ELSE IF...ELSE structure is a single, albeit extended, statement. A second feature concerns the test for the letter/digit type. The condition is an extended one which cannot be completed on a single input line (it could be shortened and made more efficient by defining some auxillary variables). RATMAC is programmed to take care of this and will test for continuation lines. The mechanism RATMAC uses to detect such cases is a balanced parenthesis count. Note that this mechanism only applies to RATMAC statements (i.e. those that contain a RATMAC keyword). Ordinary FORTRAN statements will not be automatically continued. If an ordinary non-RATMAC statement needs to be continued, two mechanisms are provided. The first is that a line ending in a comma is automatically continued; the second mechanism will be discussed when digraphs are introduced.

Now consider the following modification of the basic problem. Suppose the count of blank characters is not to be recorded and, in addition, counting is to stop when a # character is detected in the array LINE. The salient code could be modified as follows:

```
          INTEGER SHARP
          DATA SHARP/´#´/
          FOR (I=1; I<=80; I=I+1)
            IF(LINE(I)==SHARP)          # terminate
               BREAK                    # get out of loop at
            ELSE IF (LINE(I)==BLANK)    # blank
               NEXT            # no need to look at
          #                      this character further
            ELSE IF (   )      # alphanumeric
 .
 .
 .
            ELSE                       # other
```

The BREAK statement is used to prematurely terminate the FOR loop. The NEXT command is used to skip processing of an uninteresting character.

The IF...ELSE structure can be ambiguous. Consider the indented code patterns:

```
            IF (condition1)
               IF (condition2)
                   <block2>
            ELSE
                  <block3>
```

and

```
            IF (condition1)
               IF (condition2)
                  <block2>
               ELSE
                  <block3>
```

In the first form, the indentation implies that the ELSE is associated with the first IF, and in the second one it implies that the ELSE is associatd with the second IF. A convention adopted in RATMAC is that in such an ambiguous situation, an ELSE is associated with the latest unELSE´d IF.

The various possible outcomes of such a test structure are:

condition 1	condition 2	Action
true	true	block 2
false	true	no action
true	false	block 3
false	false	no action

Thus, the second indented structure is the correct one as interpreted by the RATMAC pre-processor. Of course, the structure may be made explicit using block delimiter brackets. For example, if structure one is actually the desired structure, the following could be written:

```
IF (condition1)
   {
   IF (condition2)
      <block2>
   }
ELSE
   <block3>
```

Most writers of structured programs believe that statement brackets make the program structure more clear to the original coder and any subsequent readers. The use of statement brackets will avoid ambiguity.

Miscellaneous Features

Character Set

RATMAC is designed to function with the full ASCII character set. It will accept both upper and lower case letters.

The case of the RATFOR keywords: DO, FOR, WHILE, REPEAT, UNTIL, BREAK, NEXT, IF, ELSE, and INCLUDE is unimportant. The letters can be all upper case, all lower case, or any mixture of cases.

The case of the built-in macro names and DEFINE follow a more restrictive set of rules. The letters must be all upper case or all lower case. A case mixture is not allowed.

User defined DEFINE and macro names must be used exactly as they are defined if they are to be recognized.

On a given computer which does not accept all of the ASCII characters it may be necessary to substitute some of the unavailable characters with digraphs. The available digraphs are described below.

Program Format

The input to a RATMAC program is free form. Statements may appear anywhere on an input line and users are encouraged to make use of space and tab characters for structural indentation.

A comment may appear on any line by using the hash, #, mark. The hash mark and the remainder of the input line are ignored by RATMAC. The "C", comment, line should not be used in RATMAC code. It will be mispositioned by RATMAC (i.e. it will start in column 7 as do normal FORTRAN statements).

Blanks are significant in RATMAC. Keywords may not contain imbedded blanks - they will not be recognized and they must end with a non-alphabetic character - a blank is satisfactory for this purpose. This feature may be used to advantage if a keyword has a meaning to your compiler. For instance, the word DEFINE is used in some FORTRAN implementations. Writing the word DEFINE as DEF INE will ensure that RATMAC will not recognize the word but the local compiler will. RATMAC keywords are reserved; they should not be used as FORTRAN variables. (It should be noted that the ANSI FORTRAN standard beyond FORTRAN 77 will probably make blanks significant).

Running a RATMAC Program

The details of running a RATMAC program are very machine-dependent and operating system-dependent. Briefly, RATMAC is expecting its initial input from the system standard input unit, taken as unit 5 in the distributed version. Two output streams are produced by RATMAC. The first, appearing on unit 6, is human readable and contains echoed input, generated FORTRAN output, and any diagnostic messages. The second stream, appearing on unit 7 in the distributed RATMAC, contains the generated FORTRAN code and can be sent directly to a FORTRAN compiler for compilation. The logical unit numbers are operating system dependent and may be different on your system from the standard distribution system. It is also important to note that the statements which are passed through to the compiler must conform to either the FORTRAN 66 or FORTRAN 77 ANSI standard - whichever is being used on the local machine.

The following is a typical "runstream" on a UNIVAC 1100 operating system:

```
@RUN                        . sign on
@ASG,T OUTRAT,F///256       . output file
@XQT CHIMP*ABS.RATMAC       . execute RATMAC
.
.
. RATMAC code
.
.
@EOF
@ADD,E OUTRAT.              . pass generated "runstream"
                            . to system
.
. additional commands (if any)
```

RATMAC Generated FORTRAN Code

The FORTRAN code generated by RATMAC is deliberately made unattractive. All non-essential blanks are eliminated from the code. This is done to discourage the user of RATMAC from maintaining the generated FORTRAN rather than the original RATMAC code. Occasionally it becomes necessary to cross-reference the RATMAC code with the generated FORTRAN. RATMAC offers two direct methods of such cross-referencing.

Cross-referencing by Line Number

RATMAC maintains internally a line count on the generated FORTRAN code. This line count is printed on the human readable RATMAC output (assuming the S option is on; see FLAGOFF: and FLAGON: below) at the beginning of each line following the input line number and current brace count. The FORTRAN line count is reset to 1 every time an END statement is encountered. Thus, line number diagnostics from your compiler can be linked directly to the RATMAC listing (the line number may be in error by one or two since RATMAC assumes each line output is FORTRAN code when in fact some may be operating system control lines).

Cross-referencing by Comments

By judicious use of the C option of FLAGON: and FLAGOFF: (see below) the user is able to insert all non-empty RATMAC comments into the generated FORTRAN code.

FORTRAN Generated by RATMAC Constructs

The final non-direct method of cross-referencing RATMAC and its output is by direct "reading" of the FORTRAN output. The following summary shows the code that is generated by RATMAC to emulate its control structures. L, L1, L2, . . . etc. are sequential labels generated internally by RATMAC.

Note that RATMAC may, on occasion, eliminate unnecessary CONTINUE statements. This is done when RATMAC can detect the formation of "dead" or unreachable code.

DO Statement

```
     DO <limits statement>
        <statement block>
```

becomes:

```
     DO L <limits statement>
          <statement block>
 L   CONTINUE
 L1  CONTINUE
```

FOR Statement

```
     FOR (<initialization>;<condition>;<reinitialization>)
          <statement block>
```

becomes:

```
     CONTINUE
     <initialization>
 L   IF(.NOT.(<condition>)) GO TO L2
        <statement block>
 L1  GO TO L
 L2  CONTINUE
```

WHILE Statement

```
     WHILE (<condition>)
            <statement block>
```

becomes:

```
      CONTINUE
L     IF(.NOT.(<condition>)) GO TO L1
        <statement block>
      GO TO L
L1    CONTINUE
```

REPEAT Statement

```
      REPEAT
            <statement block>
```

becomes:

```
      CONTINUE
L     CONTINUE
            <statement block>
L1    GO TO L
L2    CONTINUE
```

REPEAT...UNTIL Statement

```
      REPEAT
         <statement block>
      UNTIL (<condition>)
```

becomes:

```
      CONTINUE
L     CONTINUE
           <statement block>
L1    IF(.NOT.(<condition>)) GO TO L
L2    CONTINUE
```

NEXT and BREAK Statements

The NEXT statement generates a GO TO label where label is the generated statement number immediately following the <statement block> in a DO, FOR, WHILE, or REPEAT loop.

The BREAK statement generates a GO TO label where label is the generated statement number immediately after the loop structure.

The following shows the NEXT and BREAK statements in a REPEAT...UNTIL loop:

```
          REPEAT
             {
             <statement block 1>
             NEXT
             <statement block 2>
             BREAK
             <statement block 3>
             }
          UNTIL (<condition>)
becomes:
          CONTINUE
      L   CONTINUE
             <statement block 1>
          GO TO L1
             <statement block 2>
          GO TO L2
             <statement block 3>
      L1  IF(.NOT.(<condition>)) GO TO L
      L2  CONTINUE
```

IF Statement

```
          IF(<condition>)
             <statement block>
becomes:
          IF(.NOT.(<condition>)) GO TO L
             <statement block>
      L   CONTINUE
```

IF...ELSE Statement

```
          IF(<condition>)
             <true statement block>
          ELSE
             <false statement block>
becomes:
          IF(.NOT.(<condition>)) GO TO L
             <true statement block>
          GO TO L1
     L    CONTINUE
             <false statement block>
     L1   CONTINUE
```

Reserved Characters in RATMAC

RATMAC makes extensive use of special characters. The meaning of the special characters may be overridden or avoided by using the digraph equivalents (see below).

The following is a complete list of reserved special characters:

# hash mark: used to start a RATMAC comment

; semi-colon: used to separate multiple RATMAC statements

: colon: used to terminate a macro name

> greater: used in a relational operator > or >=

< less: used in a relational operator < or <=

! exclamation point: used as logical negation in != (.NE.) or ! (.NOT.)

^ caret: same meaning as !

\ backslash: used as a logical .OR. operator

| bar: same meaning as \

& ampersand: used as a logical .AND. operator

" double quote: used to delimit a Hollerith string that is subject to DEFINE and MACRO expansion

' single quote: used to delimit a Hollerith string that is not subject to DEFINE or MACRO expansion

[ ] square brackets: used to protect a string from MACRO or DEFINE examination

{ } curly brackets: used to define a statement block

Digraphs

Character set limitations on many computers demand the introduction of digraphs. A digraph consists of an "escape" character followed by the digraph identification character. The dollar sign, $, is the RATMAC escape character. The set of digraph characters available in RATMAC can conveniently be divided into two distinct groups:

> Group 1: Digraphs for characters with a special meaning in RATMAC. Use of a digraph from this group override the usual RATMAC interpretation of the digraph symbol (i.e. > is not translated to .GT.).

Diagraph	Output Character
$>	>
$<	<
$:	:
$$	$
$(	{
$)	}
$,	,
$L	(
$N	used internally
$R	)
$S	used internally
$[	[
$]	]
$´	´

Group 2: Digraphs for statement editing

Digraph	Meaning
$P	Position next output character in position 1 of output line
$B	An essential blank in an output line
$#	Treat the next line as a continuation of the current line

Quoted Strings in RATMAC

RATMAC allows the user to use quoted string constants. These constants are often used in statements such as:

```
DATA X/´X´/
```

Because certain FORTRAN 66 dialects use different punctuation a "Q" option is provided to automatically process quoted strings. If the Q option is switched off using the FLAGOFF: macro, then quoted strings are converted automatically to the usual Hollerith form nH....

The quotes used to delimit a string constant may be either a single quote or a double quote. On output the double quoted form will be converted to the single quote form. The single and double quote form differ in another very significant way. The text string with single quotes is never scanned for macro or defines while a text string with double quotes is scanned for macro and defines.

For example:

```
macro:(MADXIM:,100)
```

```
´Maximum dimension is MAXDIM:´
```

is output intact, whereas

```
"Maximum dimension is MAXDIM:"
```

is output as

```
´Maximum dimension is 100´
```

A quoted string constant may not be continued onto a second line. If a matching quote is not found on the initial line, RATMAC supplies one automatically.

This ends the description of the elementary command syntax of RATMAC. The examples at the end of the primer contain some actual subroutines written in RATMAC. The next section of the primer is concerned with the string manipulation features of RATMAC, DEFINE, and MACRO.

File Inclusion

It is quite common in large programs for certain blocks of code to appear in a number of places. COMMON blocks are an obvious example. RATMAC provides a mechanism, the INCLUDE command, for including such blocks of code with a single command. This shortens a program and also makes its maintenance easier. The mechanism for implementing the INCLUDE command is very machine-dependent. The machine-independent version of RATMAC implements the command with the following svntax:

```
INCLUDE <logical unit number of file>
```

When an INCLUDE command is met, the RATMAC input unit is switched from its current unit to the unit specified by <logical unit number of file>. Subsequent input is taken from this file until an end-of-file is detected; input then reverts to the original input unit at the next line. A file is rewound after it has been read so that it may be INCLUDE´d many times.

INCLUDE´d files may include the INCLUDE command up to a depth of five. Recursive use of this feature is not allowed.

It is the user´s responsibility to assure that the logical units specified are available to RATMAC and that they contain the appropriate information.

Symbolic Constants

The user of RATMAC is encouraged to make extensive use of the symbolic constant feature. Two such features are available - DEFINE and MACRO:. DEFINE is simplistic and less efficient to use than MACRO:. It is included mainly to make RATMAC and RATFOR completely compatible. Users are encouraged to make use of the MACRO: facility described later. It will accomplish exactly the same function as the DEFINE and more, but, at a lower preprocessing overhead.

The syntax of the DEFINE command is:

```
DEFINE(<name>,<arbitrary string>)
```

<name> is an identifier (of up to 132 characters) containing letters and digits. <arbitrary string> is a string of arbitrary characters. For example:

```
DEFINE(EOF,-1)
DEFINE(BITSPERWORD,36)
DEFINE(VERSION,PDP)
DEFINE(MAXDIM,2000)
DEFINE(BLANK,´ ´)
```

The aim in all cases is to replace an obscure or machine dependent string by a more evocative one. In addition, the value of a string may be localized in a single place where it is easily changed. For example, if at some time in the future the end-of-file flag had to be changed to 128, then the following change in the DEFINE command would imply a global change throughout the program:

```
DEFINE(EOF,128)
```

The change would be accomplished simply by rerunning the program through the RATMAC pre-processor, followed by recompilation.

A DEFINE command may refer to another defined string. This is often a useful feature, but it has a number of pitfalls including the introduction of an unintended recursion. Consider the following two DEFINE commands followed by the invocation of "one":

```
          DEFINE(one, the number before two)
          DEFINE(two, the number after one)
```

A token is a string of letters/digits or a special character such as a blank. The token "one" is recognized as a "defined name" and its definition is substituted. The definition string is then subjected to analysis and the token "two" is found. This in turn is recognized as a "defined name" and the definition fetched and scanned. The name "one" is found and the process repeats forever.

Another difficulty arises when an attempt is made to try to redefine a name in a DEFINE command. For example, if ARGFLAG is defined as follows:

```
          DEFINE(ARGFLAG,$)
```

and subsequently an attempt is made to redefine it with the DEFINE:

```
          DEFINE(ARGFLAG,!)
```

then the syntax of the DEFINE command is violated. In the second DEFINE, ARGFLAG is recognized as a "defined token" and replaced by its definition, $. Thus, the net effect is to construct a DEFINE command:

```
          DEFINE($,!)
```

which is illegal since $ is not an alphanumeric token.

To overcome this problem, RATMAC uses the character pair [and] as string protection brackets. A string of characters placed in square brackets is not scanned for defined tokens (nor macro tokens, see below). However, the outer level of square brackets is stripped during the scan.

Consider the DEFINE command:

```
          DEFINE(title, The Wizard of Id)
```

Then the string:

```
          The [title] is title
```

would result in:

```
          The title is The Wizard of Id
```

Obviously we can use the string protection brackets when we define a name. Thus:

```
          DEFINE([ARGFLAG],!)
```

will produce a redefinition of ARGFLAG, changing it from $ to ! without producing an error message.

Macros

The MACRO: feature provided in RATMAC has all of the features of the DEFINE command plus many more. The basic syntax of the MACRO: definition command is:

```
MACRO:(<macro name>,<macro definition>)
```

where <macro name> is an alphanumeric token which is terminated by a colon and <macro definition> is an arbitrary string of characters. An underline character is considered to be an alphanumeric character if it appears in a macro name. The following are examples of macros:

```
MACRO:(EOF:,-1)
MACRO:(CHARS_PER_WORD:,4)
```

A macro is invoked by using its name at some appropriate point in a RATMAC program. The macro name is then replaced by the definition string in the FORTRAN output

So far, the MACRO command and the DEFINE command have identical properties. The MACRO command in practice is more efficient since macro names are easily recognized and non-macro names need not be looked for in RATMAC's internal macro tables. Once a DEFINE command is processed by RATMAC, the pre-processor is forced to look up <u>every</u> variable name in its internal tables. This is a time consuming process in a long program with many variables.

The threefold power of the macro facility is:

1. Arguments - a macro name may have associated arguments.

2. Built-in macros - RATMAC has several very useful built-in macros.

3. Recursion - a macro may invoke other macros including itself.

Macro Arguments

The arguments of a MACRO are represented by the combination $n where n is an integer between 1 and 9. Consider the MACRO definition:

```
MACRO:(READLINE:,READ($1,$2,END=$4)$3)
```

The invocation in a RATMAC program:

```
READLINE:(5,10,BUFFER,999)
```

will producd the FORTRAN code:

```
READ(5,10,END=999)BUFFER
```

The MACRO definition:

```
MACRO:(MAX:,MAX0($1,$2))
```

followed by:

```
MAX:(I+J,J-K+1)
```

produces:

```
MAX0(I+J,J-K+1)
```

Built-in Macros

RATMAC comes with a number of built-in macros which have proven very useful in the preparation of programs. User defined macros are such an important feature of RATMAC that another section of the primer is devoted to their preparation and use.

Defining and Deleting

MACRO:, XMACRO:, and SAVE:

MACRO: has been discussed in its simplest form previously. XMACRO: is used to delete a macro from the internal tables within the RATMAC pre-processor. The syntax of XMACRO: is:

```
          XMACRO:([<macro name>])
```

The string protection brackets are not strictly part of the syntax, but in most cases are necessary to prevent the macro <macro name> from being evaluated (i.e. being replaced by its definition).

The macro SAVE: is used to save all macros defined before and including the one named as an argument. All macros defined _after_ the macro named as an argument are deleted. The syntax of SAVE: is as follows:

```
          SAVE:([<macro name>])
```

This macro is useful in returning a set of RATMAC code to some "base" level after temporary macros have been defined. For example, consider the following:

```
MACRO:(.......)
MACRO:(LAST_PERMANENT_MACRO:,) #
  .
  .
additional code and temporary macros
  .
  .
SAVE:([LAST_PERMANENT_MACRO:]) # delete all temporary macros
```

Such constructs are useful in co-operative programming efforts since temporary macros can be made "local" to a piece of code with little effort on the user´s part.

Arithmetic

INCR:

Another pair of macros, INCR: and ARITH:, are used to provide a simple integer arithmetic facility. The macro INCR: has a single numeric argument. On invocation the macro is replaced by the argument incremented by one. For example, suppose we have the following sequence:

```
MACRO:(MAXCARD:,80)  # size of input line
INTEGER LINE(INCR:(MAXCARD:))
```

The second line of RATMAC produces:

```
INTEGER LINE(81)
```

A non-numeric argument is taken to be zero.

ARITH:

The ARITH: macro provides a more sophisticated facility; it can perform addition, subtraction, multiplication, division, and exponentiation on integer operands. The syntax of ARITH: is:

```
ARITH:(<operand>,<operator>,<operand>)
```

where <operator> is +, -, *, /, or **. As an example, suppose we have two macros:

```
          MACRO:(BITSPERWORD:,36)    # UNIVAC word
          MACRO:(BITSPERCHAR:,9)     # ASCII code
```

then

```
   MACRO:(CHARSPERWORD:,ARITH:(BITSPERWORD:,/,BITSPERCHAR:))
```

followed by the invocation:

```
          INTEGER UNPACK(CHARSPERWORD:)
```

would produce:

```
          INTEGER UNPACK(4)
```

Character String Manipulation

LENSTR:

RATMAC provides five string manipulation macros. The simplest, LENSTR:(<string>), returns the length, in characters, of the string, <string>. Thus:

```
          LENSTR:(abcdefghijklmnopqrstuvwxyz)
```

has a replacement value of 26.

SUBSTR:

The macro SUBSTR: is a substring selection macro. Its syntax is:

```
          SUBSTR:(<string>,<start>,<length>)
```

where <string> is the string to be selected and <start> is the starting character position in the string. The character positions are numbered 1,2,... from left to right. <length> specifies the number of characters to be extracted. For example:

```
          SUBSTR:(1234567890,3,5)
```

is the string 34567.

SUBSTR: is quite permissive in its arguments. If <length> is omitted (together with the preceding comma) or if it is too big, the rest of the string is used. If <start> is out of range in the string, then the null string is returned.

IFELSE:

The macro IFELSE: is a string selector macro; it has four arguments. Its syntax is:

```
          IFELSE:(<stringA>,<stringB>,<stringC>,<stringD>)
```

If <stringA> and <stringB> are identical, character for character, then <stringC> is substituted for the macro invocation. If they are not identical, then <stringD> is substituted.

In the following example, a macro, CHPW:, is set to 1 or 4 depending on the definition of a macro, PACKING:

```
   MACRO:(PACKING:,YES)                   # turn on packing
   MACRO:(CHPW:,IFELSE:(PACKING:,YES,4,1)) # select
```

CHR: and ORD:

The built-in macros, CHR: and ORD:, have the following syntax:

```
CHR:(<decimal ASCII code value>)

ORD:(<ASCII character>)
```

The two macros are used for conversion between the graphic representation of an ASCII character and its decimal numeric equivalent. For example:

```
CHR:(32) is replaced by a blank

ORD:(a) is replaced by 97
```

Preprocessor Control

FLAGON: and FLAGOFF:

Two related macros are FLAGON: and FLAGOFF:. The argument for both macros is a single <u>upper</u> <u>case</u> letter. These two macros are used to control the mode of operation of RATMAC. The letters and their meanings when used in FLAGON:(<letter>) are:

A	Do not pass generated FORTRAN directly to the compiler (this option may not be implemented on all machines).
C	Pass non-empty RATMAC comments to the output FORTRAN as standard "C" comments.
D	Print out a line-by-line description of each macro as it is evaluated.
L	List the generated FORTRAN output on the standard output unit.

M List all of the currently defined macros (with their definitions) at run termination.

Q Do not translate quoted strings ´...´ or "..." to the Hollerith form, nH....

R Pass all code through the RATFOR section of RATMAC unchanged.

S List the input lines on the standard output unit. Each line is preceded by a line number, a statement block bracket count, and a generated FORTRAN line number.

Initially, flags Q, R, and S are turned on. The R flag is useful for taking advantage of the macro feature of RATMAC without adopting the RATFOR control structures. With the R option turned off, lines of FORTRAN are passed directly to the output file after the expansion of any macros in the line. This is also useful for passing FORTRAN statements, unchanged, through the RATMAC preprocessor.

An Alternative INCLUDE Mechanism

The INCLUDE mechanism may or may not be implementable on your machine in a convenient way. This section deals with achieving the same effect using the MACRO: command. Consider the following MACRO: command:

```
          MACRO:(SYSCOM:,COMMON/SYSBLK/A(MAX),B(MAXP1) #
          COMPLEX A #
          DOUBLE PRECISION B) #
```

The COMMON block can be inserted anywhere in the FORTRAN code by invoking the name SYSCOM: in the RATMAC code.

The only limitation imposed on this use of MACRO: is the RATMAC-imposed limit on the length of the defined string (250 characters in the distributed version). However, this limitation is easily avoided by the use of string protection brackets. Consider the following three MACRO: commands:

```
        MACRO:(SYSCOMPART1:,COMMON/SYSBLK/A(MAX),B(MAXP1)) #
        MACRO:(SYSCOMPART2:,COMPLEX A #
        DOUBLE PRECISION B) #
        MACRO:(SYSCOM:,[SYSCOMPART1: SYSCOMPART2:]) #
```

Invocation of SYSCOM: produces exactly he same results as before, but the definitions are split among three MACRO:´s making available 750 (3 X 250) characters of definition. The interested reader should implement the above MACRO:´s with and without the string protection brackets in the last MACRO:. With no string protection brackets, an error message results. In this example the # which indicates the end of the line is important since all the blanks at the right hand end of the lines are stored. Leaving the # off will cause the 750 characters available to be used up very quickly.

Macros Within Macros

The Non-recursive Case

Macros may contain other macros as part of their definition. A question that immediately arises is: At what stage is a macro evaluated? Consider the following sequence of macros and an invocation:

```
MACRO:(S1:,ABCDE)
MACRO:(S2:,S1:FGHIJ)
MACRO:([S1:],VWXYZ)

S2:
```

This produces the string ABCDEFGHIJ. The macro S1: associated with the definition of S2: is evaluated immediately and the subsequent redefinition of S1: has no effect on S2:

Compare this with the following sequence:

```
MACRO:(S1:,ABCDE)
MACRO:(S2:,[S1:]FGHIJ)
MACRO:([S1:],VWXYZ)
```

followed by the invocation S2:. The result now is the string VWXYZFGHIJ. When S2: is invoked, the definition string found is S1:FGHIJ (remember a layer of string protection brackets is stripped each time a protected string is examined). The token S1: is recognized as a macro and the current definition (i.e. VWXYZ) is substituted.

String protection brackets are useful in many circumstances. They can be used to conserve space in the macro tables, stave off evaluation of a macro, and prevent evaluation of a macro whose arguments are defined by the arguments on an outer macro.

For example, in the first definition of S2: above, the definition string is 10 characters long, ABCDEFGHIJ. In the second definition of S2:, the definition string is only 8 characters long, S1:FGHIJ.

Macros Within Macros

Consider the following macros which can be used to change a dimension on an array depending on whether PACKING: is defined as YES or otherwise.

```
MACRO:(PACKING:,YES)     # turn on packing
MACRO:(CHPW:,IFELSE:(PACKING:,YES,4,1))
MACRO:(DIM:,ARITH:(ARITH:(ARITH:($1,-,1),/,CHPW:),+,1))
```

The RATMAC statement:

```
INTEGER LINE(DIM:(80))
```

might be expected to produce:

```
INTEGER LINE(20)
```

In fact, it produces the unwanted:

```
INTEGER LINE(1)
```

The reason for this lies in the macro invocation:

```
ARITH:($1,-,1)
```

The first argument is not defined (it will be when DIM: is invoked), so zero is used.

The problem is easily corrected by using string protection brackets:

```
MACRO:(DIM:,[ARITH:(ARITH:(ARITH:($1,-,1),/,CHPW:),+,1)])
```

In this case, the arithmetic will not be performed until DIM: is invoked; at which time $1 is well-defined so that INTEGER LINE(20) will be produced.

The Recursive Case

Macros may invoke themselves recursively. In this case, string protection brackets are mandatory. For example, consider the following macro to generate the factorial of a number. The macro makes use of the relationship:

```
          factorial(n) = n*factorial(n-1)
```

and

```
          factorial(0) = 1
```

Recursion is terminated when factorial(0) is reached. In a macro, the recursion is terminated by the IFELSE: macro. The following definitions perform the evaluation:

```
MACRO:(M:,[ARITH:($1,*,$2)])       # define some auxilliary
MACRO:(D:,[ARITH:($1,-,1)])        # macros
MACRO:(FACTORIAL:,[IFELSE:($1,0,1,[M:($1,FACTORIAL:(D:($1)))])])
```

The outer set of string protection brackets is required to prevent the evaluation of the IFELSE: when FACTORIAL: is defined. The inner set is required to prevent the evaluation of M:($1,FACTORIAL:(D:($1))) when the IFELSE: is being evaluated at invocation time.

Macros calling other macros tend to be complex. Errors most frequently encountered involve unbalanced brackets and parentheses. Debugging of macros may be most easily undertaken using the D option with the FLAGON: macro. This option prints out the macro evaluation stack as each closing parenthesis on a macro invocation is detected. The printout gives the macro definition, followed by the name, followed by the macro arguments; one to a line.

The form of macro arguments is quite permissive. However, a number of problems can arise if the argument contains certain special characters. An argument containing a comma, an unbalanced parenthesis, or a square bracket, interferes with the macro scanning mechanism.

A comma used as an argument character can be distinguished from an argument delimiting comma by placing the former in string protection brackets. For example, the macro call:

```
          ZAP:(A[,]B)
```

has a single argument A,B. An alternative solution would be to use the appropriate digraph:

```
          ZAP:(A$,B)
```

Note that only commas not in balanced parentheses need be treated in this fashion. For example:

```
          ZAP:(A(I,J))
```

has a single argument: A(I,J)

Unbalanced parentheses can also use the digraph mechanism. For example, to supply a left parenthesis as an argument, we could write:

```
          ZAP:($L)
```

Some Examples of Macros

This section discusses a series of macros that can be used to isolate the machine-dependence of a given program.

The macro IFDEF: is a three argument macro. The first argument is a macro name (less the terminating colon). If the macro name is currently defined, then the macro is replaced by argument two; otherwise, by argument three.

For example, the invocation:

```
IFDEF:(MACRO,YES,NO)
```

results in YES (assuming the system macro, MACRO: has not been deleted).

The macro IFDEF: works by comparing the macro name and its definition using IFELSE:. If the macro is defined, the name and definition will be different. If the macro is not defined, no substitution will occur, and the name and its ´definition´ will be the same. The macro definition is:

```
MACRO:(IFDEF:,[IFELSE:([$1:],$1:,$3,$2)])
```

The next example concerns the generation of a sequence of numbers given a "seed" number. Such a macro is useful for generating unique statement labels.

Assume that a macro, SEED:, is available which will supply a seed integer to start a sequence:

```
MACRO:(SEED:,2000)
```

The macro MAKENUM: is now defined in the following way. A new macro SEED: is defined with a value which is the current value of SEED: reduced by the value of the argument. This new macro SEED: is the invoked to produce the number. The definition is:

```
MACRO:(MAKENUM:,[MACRO:([SEED:],ARITH:(SEED:,-,$1))SEED:])
```

Some Examples of Macros

The invocation:

```
          MAKENUM:(2)
```

produces:

```
          1998
```

A subsequent invocation:

```
          MAKENUM:(5)
```

produces:

```
          1993
```

We can now use MAKENUM: and SEED: to produce a macro that will generate an error message through a macro invocation:

```
         MESSAGE:(LUN,THIS IS A MESSAGE ON LOGICAL UNIT LUN)
```

The first argument will be the logical unit, the second is the message string.

The definition of macro MESSAGE: is:

```
          MACRO:(MESSAGE:,[WRITE($1,MAKENUM:(1))
          SEED: FORMAT(1X,"$2")])
```

Assuming a value of SEED: equal to 2000 when MESSAGE: is invoked, the following code will be produced:

```
          WRITE(LUN,1999)
    1999  FORMAT(1X,'THIS IS A MESSAGE ON LOGICAL UNIT LUN')
```

Macro definitions can be cascaded easily. For example, suppose a macro FATALMESSAGE: is needed. It is to have the same characteristics as MESSAGE: except that it is terminal. A suitable definition of FATALMESSAGE: would be:

```
   MACRO:(FATALMESSAGE:,[$(MESSAGE:($1,$2);CALL EXIT$)])
```

The invocation:

```
          FATALMESSAGE:(10,FATAL ERROR - STACK OVERFLOW)
```

produces:

```
          $(
          WRITE(10,1998)
    1998  FORMAT(1X,'FATAL ERROR - STACK OVERFLOW')
          CALL EXIT
          $)
```

The reader should note carefully the inclusion of the statement block brackets in the definition string. A macro invocation may occur as part of a control structure and it is then important that all of the code lines generated by the macro be included in the scope of the control structure.

Another use of a macro is in extending the commands of the basic language. For example, a number of rational FORTRAN dialects have a feature called procedures. A procedure is a block of code that is internal to a subroutine and can be executed repeatedly from various points within the subroutine. There is no mechanism for passing arguments to a procedure. Three macros called EXECUTE:, STARTPROC:, and ENDPROC: will be developed which will simulate the built-in procedure mechanism available in other structured FORTRAN languages.

The basic mechanism to be used is the following: The EXECUTE: macro will generate an ASSIGN label to variable statement, an unconditional jump to the procedure and a labelled CONTINUE for the return from the procedure. STARTPROC: will generate a labelled CONTINUE as the entry point of the procedure. ENDPROC: will generate a GO TO based on the ASSIGN'd jump variable. Of course, it is desirable to have EXECUTE: statements which can be nested.

The preceding statements gloss over problems of how unique labels are generated and how to keep track of these labels. Before dealing with such problems, the use of procedures in generating easily understood and maintained programs will be demonstrated.

The problem chosen is the simulation of a calculator that operates using reverse Polish notation. It is assumed that an array of integer codes which define the expression to be evaluated is available. Eight integer codes are required. The following table gives the meaning of each code:

MACRO:	CODE	MEANING
OP:	0	operand is in corresponding position of array REANUM
SUB:	1	binary subtraction
ADD:	2	binary addition
MUL:	3	binary multiplication
DIV:	4	binary division
EXP:	5	exponentiation
ASG:	6	assignment - display result
EOL:	7	end of expression

The basic structure of the program is very easy to write. It is:

```
WHILE(NOT END OF EXPRESSION)
   $(
   IF(OPERAND)
      PUSH ON STACK
   ELSE IF (OPERATOR)
      $(
      POP OPERANDS FROM STACK
      DO OPERATION
      PUSH RESULT ON STACK
      $)
   ELSE
      ERROR
   GET NEXT PROGRAM UNIT
   $)
```

It is trivial to translate the program into RATMAC assuming that the EXECUTE: macro is available. The translated program is:

```
WHILE(PRGSTR(PTR)!=EOL:)
   $(
   IF(PRGSTR(PTR)==OP:)       # operand
      $(
      EXECUTE:(PUSHONSTACK)
      $)
   ELSE IF(PRGSTR(PTR)>=SUB:&PRGSTR(PTR)<=EXP:)
      $(
      EXECUTE:(POPOFFSTACK)
      OP2=OP
      EXECUTE:(POPOFFSTACK)
      OP1=OP
      EXECUTE:(DOOPER)
      EXECUTE:(PUSHONSTACK)
      $)
   ELSE IF(PRGSTR(PTR)==ASG:)
      $(
      WRITE(STDOUT:,1)STACK(NSTK)
      1  FORMAT(' CURRENT VALUE',G12.6)
      $)
   ELSE
      $(
      MESSAGE:(STDOUT:,ILLEGAL ITEM IN STRING)
      $)
   PTR=PTR+1
   $)
```

The overall structure of the program is relatively clear since it follows the general outline closely. It is obvious that the program is in the array PRGSTR and the current "operation" is pointed to by an index PTR. The following assumption in the operator test has been made: a single range test will identify an operator. This obviously limits the integer values that can be assigned to represent operators.

With the basic "top" level of the program developed, the development of the three procedures required to implement the nitty-gritty operations can now proceed. Some conventions must be adopted. STACK is assumed to be a real array and NSTK is assumed to be an integer variable. STACK will be used to hold intermediate results and NSTK will record the position of the last entry in the stack. Initially NSTK is zero. The three procedures can be written as follows:

First, "PUSHONSTACK"

```
STARTPROC:(PUSHONSTACK)
   NSTK=NSTK+1           # increment stack pointer
   STACK(NSTK)=REANUM(PTR)  # get operand
ENDPROC:(PUSHONSTACK)
```

Second, "POPOFFSTACK"

```
          STARTPROC:(POPOFFSTACK)
             IF(NSTK>0)          # enough operands?
                $(
                OP=STACK(NSTK)
                NSTK=NSTK-1
                $)
             ELSE                # no - error off with message
                $(
                FATALMESSAGE:(STDOUT:,ILL-FORMED EXPRESSION)
                $)
          ENDPROC:(POPOFFSTACK)
```

Third, "DOOPER"

```
          STARTPROC:(DOOPER)
             IF(PRGSTR(PTR)==SUB:)
                REANUM(PTR)=OP1-OP2
             ELSE IF(PRGSTR(PTR)==ADD:)
                REANUM(PTR)=OP1+OP2
             ELSE IF(PRGSTR(PTR)==MUL:)
                REANUM(PTR)=OP1*OP2
             ELSE IF(PRGSTR(PTR)==DIV:)
                $(
                IF(OP2==0.0)
                   $(
                   FATALMESSAGE:STDOUT:,DIVISION BY ZERO)
                   $)
                ELSE
                   REANUM(PTR)=OP1/OP2
                $)
             ELSE IF(PRGSTR(PTR)==EXP:)
                $(
                IF(OP1<=0.0)
                   $(
                   FATALMESSAGE:(STDOUT:,EXPONENTIATION ERR)
                   $)
                ELSE
                   REANUM(PTR)=OP1**OP2
                $)
             ELSE           # this cannot happen
                   $(
                   FATALMESSAGE:(STDOUT:,SOMETHING WRONG)
                   $)
          ENDPROC:(DOOPER)
```

This completes the program. The procedure mechanism has allowed a fairly complex piece of code to be broken up into four blocks, each one assigned a specific task. Of course, this could be done by using appropriately defined subroutines. But, the procedures mechanism avoids the overhead of a subroutine call and localizes all of the code in one routine.

A final point remains; the placement of the procedure code. Any convenient unreachable part of the code is acceptable. The most obvious place in a routine is between the RETURN or STOP statement and the END statement. The procedure must be defined <u>after</u> the first EXECUTE: invocation. The order in which the procedures are placed is also important. The ENDPROC: macro deletes macros associated with the procedure so it cannot be referenced by a subsequent EXECUTE:. Thus if a procedure executes another procedure, the "calling" procedure must be defined before the "called" procedure.

Having written the program, the only remaining unwritten pieces are the macros required to implement the mechanism. The definition of EXECUTE: is:

```
MACRO:(EXECUTE:,[IFDEF:($1,,[[MACRO:([$1:],MAKENUM:(1))]])$#
ASSIGN MAKENUM:(1: TO J $1:; GOTO $1:; SEED: CONTINUE])
```

Although this macro looks formidable, it is relatively simple in operation. The first line checks to see whether a macro corresponding to the procedure name already exists. If the macro exists, nothing happens. If if does not, a label is generated using MAKENUM: and is saved by defining a macro based on the procedure name.

The second line produces a second statement label and generates an ASSIGN'd GO TO, followed by an unconditional transfer, followed by a labelled CONTINUE. Assuming a SEED: value of 2000, the invocation:

```
          EXECUTE:(PROC)
```

would generate:

```
          ASSIGN 1998 TO J1999
          GOTO 1999
     1998 CONTINUE
```

and would define a macro PROC: with a definition 1999. Note that it has been assumed that the user is not currently using a variable J1999 in the program. By defining the prefix letter through a macro, it would be possible to eliminate such a possible conflict.

Some Examples of Macros

The STARTPROC: and ENDPROC: macros are simple to implement. They are:

```
MACRO:(STARTPROC:,$1: CONTINUE)
MACRO:(ENDPROC:,GOTO J $1: [XMACRO:([$1:])])
```

The invocation:

```
STARTPROC:(PROC)
```

generates:

```
1999 CONTINUE
```

while the invocation:

```
ENDPROC:(PROC)
```

produces:

```
GOTO J1999
```

and removes the macro PROC: from the macro tables.

Another use of macros is in the debugging phase of program development. It is quite common practice to insert diagnostic print statements in a program for debugging purposes. When the debugging phase is over, the print statements are removed. Such removal can be time consuming and quite often the print statements need to be reinserted at some later time.

The next example shows how a macro, DEBUG:, can be used to insert diagnostic printouts and subsequently remove them.

Macro DEBUG: may have a maximum of nine arguments (the maximum allowed by the macro processor). The first argument will be an identification string, the second argument will be a format statement, the remaining seven arguments will be program variables to be printed.

For example:

```
          DEBUG:(LOOP ONE,2F12.2,AVAR,SUM)
```

will generate:

```
          WRITE(STDOUT:,1999) AVAR,SUM
    1999  FORMAT(1X,´LOOP ONE´,2F12.2)
```

where STDOUT: is the standard output unit number. The macro DEBUG: is quite straightforward. The only problem is deciding how many arguments are present and distinguishing the first variable name (which will be inserted without a leading comma) from subsequent variable names (which will be inserted with a leading comma). The macro MAKENUM: is used to generate a unique statement label for the format.

The definition of DEBUG: is:

```
   MACRO:(DEBUG:,[MAKENUM:(1) FORMAT(1X,"$1",$2)
   WRITE(STDOUT:,SEED:)ARG:($3)CARG:($4)CARG:($5)CARG:($6)$#
   CARG:($7)CARG:($8)CARG:($9)])
```

The two auxilliary macros ARG: and CARG: are very similar. They test the argument passed to them for the null value. If the value is null, then nothing is generated; if it is not, then an argument (possibly preceded by a comma) is generated.

The definitions of ARG: and CARG: are:

```
          MACRO:(ARG:,[IFELSE:($1,,,$1 )])
          MACRO:(CARG:,[IFELSE:($1,,,$,$1 )])
```

Note that the digraph $, has been used to stop IFELSE: from becoming confused in counting arguments. A blank has also been generated after each variable name. This is to stop the variable name and the macro name from being concatenated resulting in non-recognition of the macro name.

After the debugging phase is over, the debug code can be quite simply eliminated by replacing DEBUG: by a new DEBUG: macro with the following definition:

```
          MACRO:(DEBUG:,) # null DEBUG:
```

The debug statements are never physically removed from the program and can be reactivated by returning to the full debug definition, re-preprocessing the needed subroutines, and recompiling the program.

Two Complete Examples

This primer is concluded with two complete examples. The first is an interpolation sub-program, INTERP. INTERP can be used to interpolate in a sorted table of real numbers. Two methods, a binary search and a sequential search, are provided to locate suitable table values for the interpolation routine. A listing of the RATMAC code, the generated code and a copy of the test output is shown. The example shows how "job-control" lines may be interspersed in the code. In addition, a simple use of "internal procedures" is shown.

The second example is a simple line-oriented file editor with the ability to insert, replace, and delete lines from a file. The line editor is included to illustrate the use of "top-down" design, and "internal procedures" to aid in the design procedure.

Example I: A RATMAC Program for Interpolation in a Table

```
MACRO:(CTRLSTMT:,[$P@FOR,IS $B $1/$2])   # UNIVAC JCL LINE
MACRO:(SEED:,2000)
MACRO:(MAKENUM:,[MACRO:([SEED:],ARITH:(SEED:,-,$1))SEED:])
MACRO:(IFDEF:,[IFELSE:([$1:],$1:,$3,$2)])
MACRO:(EXECUTE:,[IFDEF:($1,,[[MACRO:([$1:],MAKENUM:(1))]])$#
ASSIGN MAKENUM:(1) TO J $1:; GOTO $1:; SEED: CONTINUE])
MACRO:(STARTPROC:,$1: CONTINUE)
MACRO:(ENDPROC:,GO TO J $1: [XMACRO:([$1:])])
MACRO:(DEBUG:,[MAKENUM:(1) FORMAT(1X,"$1",$2)
WRITE(STDOUT:,SEED:)ARG:($3)CARG:($4)CARG:($5)CARG:($6)$#
CARG:($7)CARG:($8)CARG:($9)])
MACRO:(STDOUT:,6)      # PRINTER UNIT
MACRO:(ARG:,[IFELSE:($(1,,,$1 )])
MACRO:(CARG:,[IFELSE:($1,,,$,$1 )])


CTRLSTMT:(INTERP)
SUBROUTINE INTERP(XIN,YOUT,XTAB,YTAB,MAXTAB,NSRCH,BSRCH,NORDER)
REAL XTAB(MAXTAB),YTAB(MAXTAB)
LOGICAL BSRCH
#
# THIS ROUTINE PERFORMS A NORDER (ASSUMED EVEN) POINT
# INTERPOLATION IN TABLES XTAB/YTAB OF LENGTH MAXTAB
#
# XIN SPECIFIES THE POINT TO BE INTERPOLATED; YOUT IS THE
```

Example I

```
# INTERPOLATED VALUE
# ** NOTE ** XIN MUST LIE WITHIN THE TABLE RANGE
# THE ROUTINE ASSUMES THAT XTAB IS SORTED IN EITHER
# ASCENDING OR DESCENDING ORDER
#
# IF BSRCH IS .TRUE. A BINARY SEARCH IS USED TO LOCATE
# APPROPRIATE VALUES FROM THE TABLES
#
# IF BSRCH IS .FALSE. A LINEAR SEARCH STARTING AT INDEX
# NSRCH IS INITIATED TO LOCATE THE VALUES.  NSRCH IS UPDATED
# DURING THE SEARCH PROCESS
#
# BSRCH = .TRUE. IS APPROPRIATE FOR "RANDOM" INTERPOLATION
# BSRCH = .FALSE. IS APPROPRIATE FOR "PROGRESSIVE"
# INTERPOLATION
#
# LOCAL VARIABLES
#
LOGICAL ASC  # FLAGS SORT MODE OF XTAB
#
# DEFINE A USEFUL LOGICAL CONSTRUCTION
#
MACRO:(LOGTST:,(ASC&XIN<=$1\ !ASC&XIN>=$1))
#
# SET ASC
#
IF(XTAB(1) < XTAB(MAXTAB))
  ASC = .TRUE.  # ASCENDING
ELSE
  ASC = .FALSE.  # DESCENDING
NHALF=NORDER/2
NUP=MAXTAB-NHALF
#
# CLASSIFY INTERPOLATION
#
IF(MAXTAB<=NORDER)  # TWO FEW VALUES - DO BEST WE CAN
  $(
  MP=1; MU=MAXTAB
  $)
ELSE IF(LOGTST:(XTAB(NHALF))) # TOO CLOSE TO LOW END
  $(
  # OF TABLE - ADJUST
  # INTERPOLATION
  MP=1; MU=NORDER
  $)
ELSE IF(!LOGTST:(XTAB(NUP)))  # TOO CLOSE TO TOP END
  # OF TABLE - ADJUST
  # INTERPOLATION
  $(
  MP=MAXTAB+1-NORDER; MU=MAXTAB
  $)
ELSE IF(BSRCH)  # BINARY SEARCH
  $(
  MP=1
  MU=MAXTAB
  REPEAT
```

```
    $(
    K=(MP+MU)/2
    IF(LOGTST:(XTAB(K)))
      MU=K-1
    ELSE
      MP=K+1
    DEBUG:(BINARY,2E12.6$,I4,XIN,XTAB(K),K)
    $)
  UNTIL(MP>MU)
  MP=MU-NHALF+1
  MU=MP+NORDER-1
  $)
ELSE
  $(
  IF(NSRCH<1 \ NSRCH>MAXTAB)   # NSRCH OUT OF RANGE
    NSRCH=(1+MAXTAB/2)   # ADJUST IT
  WHILE(!LOGTST:(XTAB(NSRCH)))
    NSRCH=NSRCH+1   # SEARCH FORWARD
  WHILE(LOGTST:(XTAB(NSRCH-1)))
    NSRCH=NSRCH-1   # SEARCH BACKWARDS
  MP=NSRCH-NHALF
  MU=MP+NORDER-1
  DEBUG:(SEQUENTIAL,2E12.6[,],I4,XIN,XTAB(NSRCH),NSRCH)
  $)
NSRCH=MP+NHALF-1   # UPDATE NSRCH EVEN
# BSRCH = .TRUE.
#
# DO INTERPOLATION
#
YOUT=GRANGE(XIN,MP,MU,XTAB,YTAB)
RETURN
END


CTRLSTMT:(GRANGE)
FUNCTION GRANGE(XIN,MIN,MAX,XTAB,YTAB)
REAL XTAB(1),YTAB(1)
#
# DOES LAGRANGE INTERPOLATION
#
GRANGE=0.0
FOR(I=MIN; I<=MAX; I=I+1)
  $(
  PROD=YTAB(I)
  STO=XTAB(I)
  FOR(J=MIN; J<=MAX; J=J+1)
    IF(I!=J)
      PROD=PROD*(XIN-XTAB(J))/(STO-XTAB(J))
    GRANGE=GRANGE+PROD
  $)
RETURN
END
```

Example I

```
CTRLSTMT:(MAIN)
LOGICAL BS
DIMENSION X(201),Y(201),U(201),W(201)
N=201
NS=0
#
# GENERATE TWO SETS OF TABLES
#
FOR(I=-100; I<=100; I=I+1)
  $(
  X(I+101)=I
  U(I+101)=-I
  Y(I+101)=I**4-I**3
  W(I+101)=-Y(I+101)
  $)
#
# GENERATE TABLE USE BINARY SEARCH
#
M=4  # ORDER OF INTERPOLATION
BS=.TRUE.
EXECUTE:(PRINTABLE)
BS=.FALSE.
EXECUTE:(PRINTABLE)
STOP
STARTPROC:(PRINTABLE)
WRITE(STDOUT:,1) BS
1  FORMAT('1 TEST OF INTERP BINARY SEARCH>',L2)
FOR(I=1; I<=10; I=I+1)
  $(
  XX=I*I
  XX=XX+0.5
  UU=-XX
  YCAL=XX**4-XX**3
  WCAL=-YCAL
  CALL INTERP(XX,YOUT,X,Y,N,NS,BS,M)
  WRITE(STDOUT:,2) I,NS,XX,YOUT,YCAL
  2  FORMAT(1X,2(2X,I3),3(2X,E20.7))
  CALL INTERP(UU,WOUT,U,W,N,NS,BS,M)
  WRITE(STDOUT:,2)I,NS,UU,WOUT,WCAL
  WRITE(STDOUT:,3)
  3  FORMAT(1X)
$)
ENDPROC:(PRINTABLE)
END
```

FORTRAN Code

```
@FOR,IS INTERP/
      SUBROUTINEINTERP(XIN,YOUT,XTAB,YTAB,MAXTAB,NSRCH,BSRCH,NORDER)
      REALXTAB(MAXTAB),YTAB(MAXTAB)
      LOGICALBSRCH
      LOGICALASC
      IF(.NOT.(XTAB(1).LT.XTAB(MAXTAB)))GOTO23000
      ASC=.TRUE.
      GOTO23001
23000 CONTINUE
      ASC=.FALSE.
23001 CONTINUE
      NHALF=NORDER/2
      NUP=MAXTAB-NHALF
      IF(.NOT.(MAXTAB.LE.NORDER))GOTO23002
      MP=1
      MU=MAXTAB
      GOTO23003
23002 CONTINUE
      IF(.NOT.((ASC.AND.XIN.LE.XTAB(NHALF).OR..NOT.ASC.AND.XIN.GE.XT
     *HALF))))GOTO23004
      MP=1
      MU=NORDER
      GOTO23005
23004 CONTINUE
      IF(.NOT.(.NOT.(ASC.AND.XIN.LE.XTAB(NUP).OR..NOT.ASC.AND.XIN.GE
     *B(NUP))))GOTO23006
      MP=MAXTAB+1-NORDER
      MU=MAXTAB
      GOTO23007
23006 CONTINUE
      IF(.NOT.(BSRCH))GOTO23008
      MP=1
      MU=MAXTAB
23010 CONTINUE
      K=(MP+MU)/2
      IF(.NOT.((ASC.AND.XIN.LE.XTAB(K).OR..NOT.ASC.AND.XIN.GE.XTAB(K
     *GOTO23013
      MU=K-1
      GOTO23014
23013 CONTINUE
      MP=K+1
23014 CONTINUE
1999  FORMAT(1X,'BINARY',2E12.6,I4)
      WRITE(6,1999)XIN,XTAB(K),K
23011 IF(.NOT.(MP.GT.MU))GOTO23010
23012 CONTINUE
      MP=MU-NHALF+1
      MU=MP+NORDER-1
      GOTO23009
23008 CONTINUE
      IF(.NOT.(NSRCH.LT.1.OR.NSRCH.GT.MAXTAB))GOTO23015
      NSRCH=(1+MAXTAB/2)
```

```
 23015 CONTINUE
 23017 IF(.NOT.(.NOT.(ASC.AND.XIN.LE.XTAB(NSRCH).OR..NOT.ASC.AND.XIN.
      *TAB(NSRCH))))GOTO23018
       NSRCH=NSRCH+1
       GOTO23017
 23018 CONTINUE
 23019 IF(.NOT.((ASC.AND.XIN.LE.XTAB(NSRCH-1).OR..NOT.ASC.AND.XIN.GE.
      *(NSRCH-1))))GOTO23020
       NSRCH=NSRCH-1
       GOTO23019
 23020 CONTINUE
       MP=NSRCH-NHALF
       MU=MP+NORDER-1
 1998  FORMAT(1X,´SEQUENTIAL´,2E12.6,I4)
       WRITE(6,1998)XIN,XTAB(NSRCH),NSRCH
 23009 CONTINUE
 23007 CONTINUE
 23005 CONTINUE
 23003 CONTINUE
       NSRCH=MP+NHALF-1
       YOUT=GRANGE(XIN,MP,MU,XTAB,YTAB)
       RETURN
       END


 @FOR,IS GRANGE/
       FUNCTIONGRANGE(XIN,MIN,MAX,XTAB,YTAB)
       REALXTAB(1),YTAB(1)
       GRANGE=0.0
       I=MIN
 23021 IF(.NOT.(I.LE.MAX))GOTO23023
       PROD=YTAB(I)
       STO=XTAB(I)
       J=MIN
 23024 IF(.NOT.(J.LE.MAX))GOTO23026
       IF(.NOT.(I.NE.J))GOTO23027
       PROD=PROD*(XIN-XTAB(J))/(STO-XTAB(J))
 23027 CONTINUE
 23025 J=J+1
       GOTO23024
 23026 CONTINUE
       GRANGE=GRANGE+PROD
 23022 I=I+1
       GOTO23021
 23023 CONTINUE
       RETURN
       END
```

```
@FOR,IS MAIN/
      LOGICALBS
      DIMENSIONX(201),Y(201),U(201),W(201)
      N=201
      NS=0
      I=-100
23029 IF(.NOT.(I.LE.100))GOTO23031
      X(I+101)=I
      U(I+101)=-I
      Y(I+101)=I**4-I**3
      W(I+101)=-Y(I+101)
23030 I=I+1
      GOTO23029
23031 CONTINUE
      M=4
      BS=.TRUE.
      ASSIGN1996TOJ1997
      GOTO1997
1996  CONTINUE
      BS=.FALSE.
      ASSIGN1995TOJ1997
      GOTO1997
1995  CONTINUE
      STOP
1997  CONTINUE
      WRITE(6,1)BS
1     FORMAT('1 TEST OF INTERP BINARY SEARCH>',L2)
      I=1
23032 IF(.NOT.(I.LE.10))GOTO23034
      XX=I*I
      XX=XX+0.5
      UU=-XX
      YCAL=XX**4-XX**3
      WCAL=-YCAL
      CALLINTERP(XX,YOUT,X,Y,N,NS,BS,M)
      WRITE(6,2)I,NS,XX,YOUT,YCAL
2     FORMAT(1X,2(2X,I3),3(2X,E20.7))
      CALLINTERP(UU,WOUT,U,W,N,NS,BS,M)
      WRITE(6,2)I,NS,UU,WOUT,WCAL
      WRITE(6,3)
3     FORMAT(1X)
23033 I=I+1
      GOTO23032
23034 CONTINUE
      GOTOJ1997
      END
```

<u>Results of the Calculation</u>

```
@XQT
TEST OF INTERP BINARY SEARCH> T
BINARY  .150000+01  .000000    101
BINARY  .150000+01  .500000+02 151
BINARY  .150000+01  .250000+02 126
BINARY  .150000+01  .120000+02 113
BINARY  .150000+01  .600000+01 107
BINARY  .150000+01  .300000+01 104
BINARY  .150000+01  .100000+01 102
BINARY  .150000+01  .200000+01 103
    1  102          .1500000+01             .1125000+01           .1687500+01
BINARY -.150000+01  .000000    101
BINARY -.150000+01 -.500000+02 151
BINARY -.150000+01 -.250000+02 126
BINARY -.150000+01 -.120000+02 113
BINARY -.150000+01 -.600000+01 107
BINARY -.150000+01 -.300000+01 104
BINARY -.150000+01 -.100000+01 102
BINARY -.150000+01 -.200000+01 103
    1  102         -.1500000+01            -.1125000+01          -.1687500+01

BINARY  .450000+01  .000000    101
BINARY  .450000+01  .500000+02 151
BINARY  .450000+01  .250000+02 126
BINARY  .450000+01  .120000+02 113
BINARY  .450000+01  .600000+01 107
BINARY  .450000+01  .300000+01 104
BINARY  .450000+01  .400000+01 105
BINARY  .450000+01  .500000+01 106
    2  105          .4500000+01             .3183750+03           .3189375+03
BINARY -.450000+01  .000000    101
BINARY -.450000+01 -.500000+02 151
BINARY -.450000+01 -.250000+02 126
BINARY -.450000+01 -.120000+02 113
BINARY -.450000+01 -.600000+01 107
BINARY -.450000+01 -.300000+01 104
BINARY -.450000+01 -.400000+01 105
BINARY -.450000+01 -.500000+01 106
     2  105           -.4500000+01          -.3183750+03          -.3189375+03

BINARY  .950000+01  .000000    101
BINARY  .950000+01  .500000+02 151
BINARY  .950000+01  .250000+02 126
BINARY  .950000+01  .120000+02 113
BINARY  .950000+01  .600000+01 107
BINARY  .950000+01  .900000+01 110
BINARY  .950000+01  .100000+02 111
     3  110            .9500000+01           .7287125+04           .7287687+04
BINARY -.950000+01  .000000    101
BINARY -.950000+01 -.500000+02 151
BINARY -.950000+01 -.250000+02 126
BINARY -.950000+01 -.120000+02 113
BINARY -.950000+01 -.600000+01 107
BINARY -.950000+01 -.900000+01 110
BINARY -.950000+01 -.100000+02 111
     3  110           -.9500000+01          -.7287125+04          -.7287687+04

BINARY  .165000+02  .000000    101
BINARY  .165000+02  .500000+02 151
BINARY  .165000+02  .250000+02 126
BINARY  .165000+02  .120000+02 113
BINARY  .165000+02  .180000+02 119
BINARY  .165000+02  .150000+02 116
BINARY  .165000+02  .160000+02 117
BINARY  .165000+02  .170000+02 118
     4  117            .1650000+02           .6962737+05           .6962794+05
BINARY -.165000+02  .000000    101
BINARY -.165000+02 -.500000+02 151
BINARY -.165000+02 -.250000+02 126
BINARY -.165000+02 -.120000+02 113
BINARY -.165000+02 -.180000+02 119
BINARY -.165000+02 -.150000+02 116
BINARY -.165000+02 -.160000+02 117
BINARY -.165000+02 -.170000+02 118
     4  117          -.1650000+02          -.6962737+05          -.6962794+05

BINARY  .255000+02  .000000    101
BINARY  .255000+02  .500000+02 151
BINARY  .255000+02  .250000+02 126
BINARY  .255000+02  .370000+02 138
BINARY  .255000+02  .310000+02 132
BINARY  .255000+02  .280000+02 129
BINARY  .255000+02  .260000+02 127
     5  126           .2550000+02           .4062431+06           .4062437+06
BINARY -.255000+02  .000000    101
BINARY -.255000+02 -.500000+02 151
BINARY -.255000+02 -.250000+02 126
BINARY -.255000+02 -.370000+02 138
BINARY -.255000+02 -.310000+02 132
BINARY -.255000+02 -.280000+02 129
BINARY -.255000+02 -.260000+02 127
     5  126          -.2550000+02          -.4062431+06          -.4062437+06

BINARY  .365000+02  .000000    101
BINARY  .365000+02  .500000+02 151
BINARY  .365000+02  .250000+02 126
BINARY  .365000+02  .370000+02 138
BINARY  .365000+02  .310000+02 132
BINARY  .365000+02  .340000+02 135
BINARY  .365000+02  .350000+02 136
BINARY  .365000+02  .360000+02 137
     6  137           .3650000+02           .1726262+07           .1726263+07
BINARY -.365000+02  .000000    101
BINARY -.365000+02 -.500000+02 151
BINARY -.365000+02 -.250000+02 126
BINARY -.365000+02 -.370000+02 138
BINARY -.365000+02 -.310000+02 132
BINARY -.365000+02 -.340000+02 135
BINARY -.365000+02 -.350000+02 136
BINARY -.365000+02 -.360000+02 137
     6  137          -.3650000+02              -.1726262+07           -.1726263+07

BINARY  .495000+02  .000000    101
BINARY  .495000+02  .500000+02 151
BINARY  .495000+02  .250000+02 126
BINARY  .495000+02  .370000+02 138
BINARY  .495000+02  .430000+02 144
BINARY  .495000+02  .460000+02 147
BINARY  .495000+02  .480000+02 149
BINARY  .495000+02  .490000+02 150
     7  150           .4950000+02               .5882437+07            .5882438+07
BINARY -.495000+02  .000000    101
BINARY -.495000+02 -.500000+02 151
BINARY -.495000+02 -.250000+02 126
BINARY -.495000+02 -.370000+02 138
BINARY -.495000+02 -.430000+02 144
BINARY -.495000+02 -.460000+02 147
BINARY -.495000+02 -.480000+02 149
BINARY -.495000+02 -.490000+02 150
     7  150          -.4950000+02              -.5882437+07           -.5882438+07

BINARY  .645000+02  .000000    101
BINARY  .645000+02  .500000+02 151
BINARY  .645000+02  .750000+02 176
BINARY  .645000+02  .620000+02 163
BINARY  .645000+02  .680000+02 169
BINARY  .645000+02  .650000+02 166
BINARY  .645000+02  .630000+02 164
BINARY  .645000+02  .640000+02 165
     8  165           .6450000+02               .1703934+08            .1703934+08
BINARY -.645000+02  .000000    101
BINARY -.645000+02 -.500000+02 151
BINARY -.645000+02 -.750000+02 176
BINARY -.645000+02 -.620000+02 163
BINARY -.645000+02 -.680000+02 169
BINARY -.645000+02 -.650000+02 166
BINARY -.645000+02 -.630000+02 164
BINARY -.645000+02 -.640000+02 165
    8  165          -.6450000+02              -.1703934+08           -.1703934+08

BINARY  .815000+02  .000000    101
BINARY  .815000+02  .500000+02 151
BINARY  .815000+02  .750000+02 176
BINARY  .815000+02  .880000+02 189
BINARY  .815000+02  .810000+02 182
BINARY  .815000+02  .840000+02 185
BINARY  .815000+02  .820000+02 183
    9  182           .8150000+02               .4357814+08            .4357814+08
BINARY -.815000+02  .000000    101
BINARY -.815000+02 -.500000+02 151
BINARY -.815000+02 -.750000+02 176
BINARY -.815000+02 -.880000+02 189
BINARY -.815000+02 -.810000+02 182
BINARY -.815000+02 -.840000+02 185
BINARY -.815000+02 -.820000+02 183
    9  182          -.8150000+02              -.4357814+08           -.4357814+08

   10  199           .1005000+03               .1010000+09            .1010000+09
   10  199          -.1005000+03              -.1010000+09           -.1010000+09

 TEST OF INTERP BINARY SEARCH> F
SEQUENTIAL  .150000+01  .200000+01 103
    1  102           .1500000+01               .1125000+01            .1687500+01
SEQUENTIAL -.150000+01 -.200000+01 103
    1  102          -.1500000+01              -.1125000+01           -.1687500+01

SEQUENTIAL  .450000+01  .500000+01 106
    2  105           .4500000+01               .3183750+03            .3189375+03
SEQUENTIAL -.450000+01 -.500000+01 106
    2  105          -.4500000+01              -.3183750+03           -.3189375+03
```

```
SEQUENTIAL  .950000+01  .100000+02 111
    3  110          .9500000+01          .7287125+04          .7287687+04
SEQUENTIAL -.950000+01 -.100000+02 111
    3  110         -.9500000+01         -.7287125+04         -.7287687+04

SEQUENTIAL  .165000+02  .170000+02 118
    4  117          .1650000+02          .6962737+05          .6962794+05
SEQUENTIAL -.165000+02 -.170000+02 118
    4  117         -.1650000+02         -.6962737+05         -.6962794+05

SEQUENTIAL  .255000+02  .260000+02 127
    5  126          .2550000+02          .4062431+06          .4062437+06
SEQUENTIAL -.255000+02 -.260000+02 127
    5  126         -.2550000+02         -.4062431+06         -.4062437+06

SEQUENTIAL  .365000+02  .370000+02 138
    6  137          .3650000+02          .1726262+07          .1726263+07
SEQUENTIAL -.365000+02 -.370000+02 138
    6  137         -.3650000+02         -.1726262+07         -.1726263+07

SEQUENTIAL  .495000+02  .500000+02 151
    7  150          .4950000+02          .5882437+07          .5882438+07
SEQUENTIAL -.495000+02 -.500000+02 151
    7  150         -.4950000+02         -.5882437+07         -.5882438+07

SEQUENTIAL  .645000+02  .650000+02 166
    8  165          .6450000+02          .1703934+08          .1703934+08
SEQUENTIAL -.645000+02 -.650000+02 166
    8  165         -.6450000+02         -.1703934+08         -.1703934+08

SEQUENTIAL  .815000+02  .820000+02 183
    9  182          .8150000+02          .4357814+08          .4357814+08
SEQUENTIAL -.815000+02 -.820000+02 183
    9  182         -.8150000+02         -.4357814+08         -.4357814+08

   10  199          .1005000+03          .1010000+09          .1010000+09
   10  199         -.1005000+03         -.1010000+09         -.1010000+09
NORMAL EXIT.     EXECUTION TIME:   330 MILLISECONDS.
```

Example II: A RATMAC Program to Implement a Simple File Editor

The file editor is designed to copy an input file to an output file. During the copy process, the file is modified by commands entered via a standard input unit, STDIN:. A command has the general format:

```
*X,<initial line>,<final line>
```

The asterisk must be the first character on the input line. X is a single letter from the set I(insertion), R(replacement), D(deletion), and E(end of commands). <initial line> and <final line> are integer line numbers used to identify the lines of the input file to be modified. A typical modification stream might be:

```
*I,17                insert after line 17
this text
and this text
and this text
**                   this line terminates the insert text
*R,54,57             replaces lines 54 through 57
with this text
**
*D,59               delete a single line
*D,65,102           delete a group of lines
```

Using procedures and top down design, the program has four "levels". The top level is the procedure MAIN which in nine lines defines the control flow of the editing process. It pushes down to a lower level decisions on how instructions are read and decoded, or how the files are read or written. The second level contains the procedures INTERPRET_INSTRUCTION, READ_INSTRUCTION, and COPY_REST, which also put off to a lower level decisions on how files are read and written. The third level contains some utility procedures, while the fourth level, CTOI, END_OF_FILE, WRITE_LINE, READ_LINE, PUT_LINE, and GET_LINE contain the "primitives" that interface with the I/O system of the computer.

Each procedure is short and has a single function which is easy to understand.

RATMAC Code for the Line Editor

```
#
#    A SIMPLE LINE ORIENTED EDITOR
#
#    COMMANDS
#
#    *I,M      INSERT FOLLOWING TEXT AFTER MTH LINE
#    *D,M,N    DELETE LINES M TO N
#    *R,M,N    REPLACE LINES M TO N BY FOLLOWING TEXT
#    **        TERMINATE INSERTION OR REPLACEMENT TEXT
#    *E        END EDITING PROCESS
#
#    *I,2
#      INSERTED LINE
#    **
#    *R,7,10
#      NEW REPLACEMENT LINE
#    **
#    *D,20,30
#    *E
#
MACRO:(CTRLSTMT:,$P@[FOR,IS]$B$1/$2)


CTRLSTMT:(EDITOR,FOR)
MACRO:(STDIN:,5)  #  STANDARD INPUT
MACRO:(STDOUT:,6) #  STANDARD OUTPUT
#
# IF AN ARGUMENT IS PRESENT
# GENERATE, ARGUMENT
#
MACRO:(CARG:,[IFELSE:($1,,,$,$1 )])
#
# IF AN ARGUMENT IS PRESENT
# GENERATE ARGUMENT
#
MACRO:(ARG:,[IFELSE:($1,,,$1 )])
#
#    THE MACRO DEBUG: IS A GENERAL PURPOSE DEBUGGING MACRO
#
#    ARGUMENT 1 IS AN IDENTIFICATION MESSAGE
#    ARGUMENT 2 IS A FORMAT
#
#    ARGUMENTS 3 THRU 9 ARE VARIABLES TO BE DUMPED
#
MACRO:(DEBUG:,[MAKENUM:(1) FORMAT(1X,"$1",$2)
WRITE(STDOUT:,SEED:)ARG:($3)CARG:($4)CARG:($5)CARG:($6)$#
CARG:($7)CARG:($8)CARG:($9)])
#
# PROCEDURE EXECUTION MACRO
#
MACRO:(EXECUTE:,[IFDEF:($1,,[[MACRO:([$1:],MAKENUM:(1))]])$#
ASSIGN MAKENUM:(1) TO J $1:; GOTO $1:; SEED: CONTINUE])
```

Example II

```
#
# DEFINE START OF
# PROCEDURE
#
MACRO:(STARTPROC:,$1: CONTINUE)
#
# DEFINE THE END OF A
# PROCEDURE
#
MACRO:(ENDPROC:,GOTO J $1: [XMACRO:([$1:])])
#
# EXAMINES IF
# A MACRO IS
# DEFINED
#

MACRO:(IFDEF:,[IFELSE:([$1:],$1:,$3,$2)])
#
#
#
MACRO:(SEED:,2000)
MACRO:(MAKENUM:,[MACRO:([SEED:],ARITH:(SEED:,-,$1))SEED:])
MACRO:(MESSAGE:,[WRITE($1,MAKENUM:(1))
SEED: FORMAT(1X,"$2")])
MACRO:(CHARACTER:,INTEGER) # MAP CHARACTER ONTO INTEGER
MACRO:(GETVALCHAR:,FLD(0,6,$1))   #  UNIVAC SPECIFIC
MACRO:(HUGE:,100000) # A HUGE NUMBER OF LINES TO END FILE
MACRO:(IN:,9) # EDITOR INPUT FILE
MACRO:(OUT:,10) # EDITOR OUTPUT FILE
MACRO:(MAXCARD:,80) # CHARACTERS ON A CARD
#
CHARACTER: INSTR(MAXCARD:), NCL(MAXCARD:)
#
LOGICAL EOFSTI
LOGICAL NOEND
LOGICAL EOFIN
DATA LNO/0/ # INITIAL LINE NUMBER
DATA EOFSTI,NOEND,EOFIN/.FALSE.,.TRUE.,.FALSE./
#
# PROGRAM DRIVER
#
EXECUTE:(MAIN)
#
# - MAIN EDITOR PROCEDURE -
#
STARTPROC:(MAIN)
EXECUTE:(GET_LINE)
EXECUTE:(READ_INSTRUCTION)
REPEAT
  $(
  EXECUTE:(INTERPRET_INSTRUCTION)
EXECUTE:(WRITE_LINE)
EXECUTE:(READ_INSTRUCTION)
$)
STOP
ENDPROC:(MAIN)
```

```
#
#
# - PROCEDURE TO EXECUTE AN INSTRUCTION -
#
STARTPROC:(INTERPRET_INSTRUCTION)
EXECUTE:(COPY)  # GET TO FIRST LINE
IF(KEY=='I')
  $(
  EXECUTE:(PUT_LINE)
  EXECUTE:(INSERT)
  $)
ELSE IF (KEY=='R')
  $(
  EXECUTE:(INSERT)
  EXECUTE:(SKIP)
  $)
ELSE IF (KEY=='D')
  $(
  EXECUTE:(SKIP)
  $)
ELSE IF (KEY=='E')
  $(
  EXECUTE:(COPY_REST)
  $)
  ELSE
  $(
  MESSAGE:(STDOUT:,ERROR IN INSTRUCTION)
  EXECUTE:(COPY_REST)
  $)
ENDPROC:(INTERPRET_INSTRUCTION)
#
#
# - PROCEDURE TO READ AN INSTRUCTION AND DECODE IT -
#
STARTPROC:(READ_INSTRUCTION)
#
# READS A LINE FROM STARDARD INPUT AND INTERPRETS INSTRUCTION
#
READ(STDIN:,1,END=2) (INSTR(I),I=1,80)
1 FORMAT(80A1)
GO TO 3
2 INSTR(1)='*'
INSTR(2)='E'
INSTR(3)=' '
3 CONTINUE
KEY=INSTR(2)
IPT=3
IF(INSTR(IPT)==',')
  $(
  IPT=IPT+1
  EXECUTE:(CTOI)
  M=NUMBER
  $)
ELSE
  M=LNO
IF(INSTR(IPT)==',')
```

```
   $(
   IPT=IPT+1
   EXECUTE:(CTOI)
   N=NUMBER
   $)
 ELSE
   N=M
 DEBUG:(INSTRUCTION,1(A1,3I4),INSTR(2),LNO,M,N)
 ENDPROC:(READ_INSTRUCTION)
 #
 #
 # - PROCEDURE TO INSERT A BLOCK OF TEXT IN FILE FROM STDIN:
 #
 STARTPROC:(INSERT)
 #
 # READS LINES FROM STDIN: AND PUTS THEM IN A FILE
 #
 # TERMINATES WHEN ´NOEND´ BECOMES FALSE
 #
 DEBUG:( START INSERT ,3(2X,I4),LNO,M,N)
 NOEND=.TRUE.
 EXECUTE:(READ_LINE)
 WHILE (NOEND)
   $(
   EXECUTE:(PUT_LINE)
   EXECUTE:(READ_LINE)
   $)
 EXECUTE:(GET_LINE)
 DEBUG:( END INSERT ,3(2X,I4),LNO,M,N)
 ENDPROC:(INSERT)
 #
 #
 # - PROCEDURE TO COPY REMAINDER OF FILE TO OUTPUT FILE -
 #
 STARTPROC:(COPY_REST)
 #
 # COPIES REST OF INPUT FILE TO OUTPUT FILE
 #
 DEBUG:( COPY_REST ,3(2X,I4),LNO,M,N)
 M=HUGE:  # SET LAST LINE TO A HUGE NUMBER
 EXECUTE:(COPY)
 EXECUTE:(END_OF_FILE)
 DEBUG:( END COPY_REST ,3(2X,I4),LNO,M,N)
 STOP
 ENDPROC:(COPY_REST)
 #
 #
 # - PROCEDURE TO COPY A BLOCK OF LINES FROM FILE TO OUTPUT F
 #
 STARTPROC:(COPY)
 #
 # COPIES FROM INPUT FILE TO OUTPUT FILE
 #
 DEBUG:(START COPY,3I4,LNO,M,N)
 WHILE(LNO<M)
   $(
  EXECUTE:(PUT_LINE)
  EXECUTE:(GET_LINE)
  $)
DEBUG:(END COPY,3I4,LNO,M,N)
ENDPROC:(COPY)
#
#
# - PROCEDURE TO SKIP A BLOCK OF LINES ON THE INPUT FILE -
#
STARTPROC:(SKIP)
#
# SKIPS LINES ON INPUT FILE
#
DEBUG:(START SKIP,3I4,LNO,M,N)
WHILE(LNO<=N)
  $(
  EXECUTE:(GET_LINE)
  $)
DEBUG:(END SKIP,3I4,LNO,M,N)
ENDPROC:(SKIP)
#
#
# - PROCEDURE TO TRANSLATE A CHARACTER STRING TO AN INTEGER
#
STARTPROC:(CTOI)
WHILE(INSTR(IPT)==´ ´)
  IPT=IPT+1
NUMBER=0
DEBUG:(CTOI START,1(A1,I4),INSTR(IPT),IPT)
REPEAT $(
  K=FLD(0,6,INSTR(IPT))-FLD(0,6,´0´)
  IF(K<0 \ K>9)
    BREAK
  IPT=IPT+1
  NUMBER=10*NUMBER+K
  $)
DEBUG:(END CTOI,2I4,NUMBER,IPT)
ENDPROC:(CTOI)
#
#
# - PROCEDURE TO READ A REPLACEMENT LINE FROM STDIN: -
#
STARTPROC:(READ_LINE)
#
# READS A LINE FROM STDIN
#
IF(!EOFSTI)
  $(
  READ(STDIN:,1,END=12) (NCL(I),I=1,80)
  GO TO 13
  12 EOFSTI=.TRUE.
  13 CONTINUE
  $)
IF(NCL(1)==´*´ & NCL(2)==´*´)
  NOEND=.FALSE.
DEBUG:( READ_LINE ,80A1,(NCL(I),I=1,80))
```

```
ENDPROC:(READ_LINE)
#
#
# - PROCEDURE TO WRITE THE CURRENT FILE LINE TO STDOUT: -
#
STARTPROC:(WRITE_LINE)
DEBUG:( WRITE_LINE ,3(2X,I4),LNO,M,N)
WRITE(STDOUT:,14)  LNO,(NCL(I),I=1,80)
14 FORMAT(' CURRENT LINE ',I8,1X,80A1)
ENDPROC:(WRITE_LINE)
#
#
# - PROCEDURE TO WRITE AN EOF ON THE OUTPUT FILE -
#
STARTPROC:(END_OF_FILE)
ENDFILE OUT:
REWIND OUT:
DEBUG:(END OF FILE,I4,LNO)
ENDPROC:(END_OF_FILE)
#
#
# - PROCEDURE TO GET A LINE FROM THE INPUT FILE -
#
STARTPROC:(GET_LINE)
#
# GETS A LINE FROM FILE IN: - STORED IN NCL
#
LNO=LNO+1
IF(!EOFIN)
  $(
  READ(IN:,1,END=10) (NCL(I),I=1,80)
  GO TO 11
  10 EOFIN=.TRUE.
  LNO=HUGE:+1
  11 CONTINUE
  $)
DEBUG:( FILE IN ,L2$,I4$,80A1,EOFIN,LNO,(NCL(I),I=1,80))
ENDPROC:(GET_LINE)
#
#
# - PROCEDURE TO PUT A LINE INTO THE OUTPUT FILE -
#
STARTPROC:(PUT_LINE)
#
# PUTS NCL INTO FILE OUT:
#
DEBUG:( FILE OUT ,80A1,(NCL(I),I=1,80))
WRITE(OUT:,1)(NCL(I),I=1,80)
ENDPROC:(PUT_LINE)
END
```

FORTRAN Code

```
@FOR,IS EDITOR/FOR
      INTEGERINSTR(80),NCL(80)
      LOGICALEOFSTI
      LOGICALNOEND
      LOGICALEOFIN
      DATALNO/0/
      DATAEOFSTI,NOEND,EOFIN/.FALSE.,.TRUE.,.FALSE./
      ASSIGN1998TOJ1999
      GOTO1999
1998  CONTINUE
1999  CONTINUE
      ASSIGN1996TOJ1997
      GOTO1997
1996  CONTINUE
      ASSIGN1994TOJ1995
      GOTO1995
1994  CONTINUE
23000 CONTINUE
      ASSIGN1992TOJ1993
      GOTO1993
1992  CONTINUE
      ASSIGN1990TOJ1991
      GOTO1991
1990  CONTINUE
      ASSIGN1989TOJ1995
      GOTO1995
1989  CONTINUE
23001 GOTO23000
23002 CONTINUE
*DIAGNOSTIC*  CONTROL CAN NEVER REACH THE NEXT STATEMENT
      STOP
*DIAGNOSTIC*  CONTROL CAN NEVER REACH THE NEXT STATEMENT
      GOTOJ1999
1993  CONTINUE
      ASSIGN1987TOJ1988
      GOTO1988
1987  CONTINUE
      IF(.NOT.(KEY.EQ.´I´))GOTO23003
      ASSIGN1985TOJ1986
      GOTO1986
1985  CONTINUE
      ASSIGN1983TOJ1984
      GOTO1984
1983  CONTINUE
      GOTO23004
23003 CONTINUE
      IF(.NOT.(KEY.EQ.´R´))GOTO23005
      ASSIGN1982TOJ1984
      GOTO1984
1982  CONTINUE
      ASSIGN1980TOJ1981
      GOTO1981
1980  CONTINUE
```

```
      GOTO23006
23005 CONTINUE
      IF(.NOT.(KEY.EQ.'D'))GOTO23007
      ASSIGN1979TOJ1981
      GOTO1981
1979  CONTINUE
      GOTO23008
23007 CONTINUE
      IF(.NOT.(KEY.EQ.'E'))GOTO23009
      ASSIGN1977TOJ1978
      GOTO1978
1977  CONTINUE
      GOTO23010
23009 CONTINUE
      WRITE(6,1976)
1976  FORMAT(1X,'ERROR IN INSTRUCTION')
      ASSIGN1975TOJ1978
      GOTO1978
1975  CONTINUE
23010 CONTINUE
23008 CONTINUE
23006 CONTINUE
23004 CONTINUE
      GOTOJ1993
1995  CONTINUE
      READ(5,1,END=2)(INSTR(I),I=1,80)
1     FORMAT(80A1)
      GOTO3
2     INSTR(1)='*'
      INSTR(2)='E'
      INSTR(3)=' '
3     CONTINUE
      KEY=INSTR(2)
      IPT=3
      IF(.NOT.(INSTR(IPT).EQ.','))GOTO23011
      IPT=IPT+1
      ASSIGN1973TOJ1974
      GOTO1974
1973  CONTINUE
      M=NUMBER
      GOTO23012
23011 CONTINUE
      M=LNO
23012 CONTINUE
      IF(.NOT.(INSTR(IPT).EQ.','))GOTO23013
      IPT=IPT+1
      ASSIGN1972TOJ1974
      GOTO1974
1972  CONTINUE
      N=NUMBER
      GOTO23014
23013 CONTINUE
      N=M
23014 CONTINUE
1971  FORMAT(1X,'INSTRUCTION',1(A1,3I4))
      WRITE(6,1971)INSTR(2),LNO,M,N
      GOTOJ1995
1984  CONTINUE
1970  FORMAT(1X,' START INSERT ',3(2X,I4))
      WRITE(6,1970)LNO,M,N
      NOEND=.TRUE.
      ASSIGN1968TOJ1969
      GOTO1969
1968  CONTINUE
23015 IF(.NOT.(NOEND))GOTO23016
      ASSIGN1967TOJ1986
      GOTO1986
1967  CONTINUE
      ASSIGN1966TOJ1969
      GOTO1969
1966  CONTINUE
      GOTO23015
23016 CONTINUE
      ASSIGN1965TOJ1997
      GOTO1997
1965  CONTINUE
1964  FORMAT(1X,' END INSERT ',3(2X,I4))
      WRITE(6,1964)LNO,M,N
      GOTOJ1984
1978  CONTINUE
1963  FORMAT(1X,' COPY_REST ',3(2X,I4))
      WRITE(6,1963)LNO,M,N
      M=100000
      ASSIGN1962TOJ1988
      GOTO1988
1962  CONTINUE
      ASSIGN1960TOJ1961
      GOTO1961
1960  CONTINUE
1959  FORMAT(1X,' END COPY_REST ',3(2X,I4))
      WRITE(6,1959)LNO,M,N
      STOP
*DIAGNOSTIC*  CONTROL CAN NEVER REACH THE NEXT STATEMENT
      GOTOJ1978
1988  CONTINUE
1958  FORMAT(1X,'START COPY',3I4)
      WRITE(6,1958)LNO,M,N
23017 IF(.NOT.(LNO.LT.M))GOTO23018
      ASSIGN1957TOJ1986
      GOTO1986
1957  CONTINUE
      ASSIGN1956TOJ1997
      GOTO1997
1956  CONTINUE
      GOTO23017
23018 CONTINUE
1955  FORMAT(1X,'END COPY',3I4)
      WRITE(6,1955)LNO,M,N
      GOTOJ1988
1981  CONTINUE
1954  FORMAT(1X,'START SKIP',3I4)
      WRITE(6,1954)LNO,M,N
```

```
 23019 IF(.NOT.(LNO.LE.N))GOTO23020
       ASSIGN1953TOJ1997
       GOTO1997
 1953  CONTINUE
       GOTO23019
 23020 CONTINUE
 1952  FORMAT(1X,'END SKIP',3I4)
       WRITE(6,1952)LNO,M,N
       GOTOJ1981
 1974  CONTINUE
 23021 IF(.NOT.(INSTR(IPT).EQ.' '))GOTO23022
       IPT=IPT+1
       GOTO23021
 23022 CONTINUE
       NUMBER=0
 1951  FORMAT(1X,'CTOI START',1(A1,I4))
       WRITE(6,1951)INSTR(IPT),IPT
 23023 CONTINUE
       K=FLD(0,6,INSTR(IPT))-FLD(0,6,'0')
       IF(.NOT.(K.LT.0.OR.K.GT.9))GOTO23026
       GOTO23025
 23026 CONTINUE
       IPT=IPT+1
       NUMBER=10*NUMBER+K
 23024 GOTO23023
 23025 CONTINUE
 1950  FORMAT(1X,'END CTOI',2I4)
       WRITE(6,1950)NUMBER,IPT
       GOTOJ1974
 1969  CONTINUE
       IF(.NOT.(.NOT.EOFSTI))GOTO23028
       READ(5,1,END=12)(NCL(I),I=1,80)
       GOTO13
 12    EOFSTI=.TRUE.
 13    CONTINUE
 23028 CONTINUE
       IF(.NOT.(NCL(1).EQ.'*'.AND.NCL(2).EQ.'*'))GOTO23030
       NOEND=.FALSE.
 23030 CONTINUE
 1949  FORMAT(1X,' READ_LINE ',80A1)
       WRITE(6,1949)(NCL(I),I=1,80)
       GOTOJ1969
 1991  CONTINUE
 1948  FORMAT(1X,' WRITE_LINE ',3(2X,I4))
       WRITE(6,1948)LNO,M,N
       WRITE(6,14)LNO,(NCL(I),I=1,80)
 14    FORMAT(' CURRENT LINE ',I8,1X,80A1)
       GOTOJ1991
 1961  CONTINUE
       ENDFILE10
       REWIND10
 1947  FORMAT(1X,'END OF FILE',I4)
       WRITE(6,1947)LNO
       GOTOJ1961
 1997  CONTINUE
       LNO=LNO+1
```

```
      IF(.NOT.(.NOT.EOFIN))GOTO23032
      READ(9,1,END=10)(NCL(I),I=1,80)
      GOTO11
10    EOFIN=.TRUE.
      LNO=100000+1
11    CONTINUE
23032 CONTINUE
1946  FORMAT(1X,' FILE IN ',L2,I4,80A1)
      WRITE(6,1946)EOFIN,LNO,(NCL(I),I=1,80)
      GOTOJ1997
1986  CONTINUE
1945  FORMAT(1X,' FILE OUT ',80A1)
      WRITE(6,1945)(NCL(I),I=1,80)
      WRITE(10,1)(NCL(I),I=1,80)
      GOTOJ1986
      END
      END OF COMPILATION:          3 DIAGNOSTICS.
```

Input Text

```
THE FOLLOWING TEXT WILL ILLUSTRATE THE COMMANDS
OF THE LINE  EDITOR.  THE END OF THIS LINE SHOULD
HAVE 2 NUMBERED LINES INSERTED AFTER IT.
LINES A AND B SHOULD BE DELETED.
A
B
C
D
LINES X AND Y SHOULD BE REPLACED BY R1 AND R2.
X
Y
Z
```

Edit Text

```
*I,3
1 FIRST INSERTED LINE
2 SECOND INSERTED LINE
**
*D,5,6
*R,10,11
R1
R2
**
*E
```

Edited Text

```
THE FOLLOWING TEXT WILL ILLUSTRATE THE COMMANDS
OF THE LINE  EDITOR.  THE END OF THIS LINE SHOULD
HAVE 2 NUMBERED LINES INSERTED AFTER IT.
1 FIRST INSERTED LINE
2 SECOND INSERTED LINE
LINES A AND B SHOULD BE DELETED.
C
D
LINES X AND Y SHOULD BE REPLACED BY R1 AND R2.
R1
R2
Z
```

RATMAC Error Messages

The general format of an error message is:

```
***** ERROR AT LINE XXX
<ERROR MESSAGE>
<ADDITIONAL DIAGNOSTIC INFORMATION>
```

A trace-back through INCLUDE´d files is also given if appropriate. XXX is a RATMAC line number.

The following table lists the error messages:

MESSAGE	EXPLANATION	FATAL
missing left parenthesis	self-explanatory	no
missing parenthesis in CONDITION	"	no
illegal break	"	no
illegal next	"	no
missing quote	a quote string must be complete on a single input line	no
unexpected brace of EOF	self-explanatory	no
unbalanced parentheses	"	no
invalid FOR clause	"	no
for stack oflo, check reinit clause	probably a missing closing right parenthesis	yes
INCLUDE´s nested to deeply	self-explanatory	no
cannot open INCLUDE	check file availability	no
token too long	self-explanatory	no
name too long	macro name too long	yes
definition too long	macro definition too long	yes

too many definitions	too many macro definitions	no
warning: possible label confilict	user label >=23000	no
illegal ELSE	ELSE but no IF	no
stack overflow in parser	too complicated a control structure in RATFOR	yes
illegal right brace	left brace missing?	no
unexpected EOF	Ratmac unfinished when an end-of-file was found	no
undefined macro	self-explanatory	no
call stack overflow	too many nested macro calls	yes
arg stack overflow	too many and/or too long macro arguments	yes
evaluation stack overflow	not enough space to evaluate a macro	yes
illegal macro name	missing terminating colon	no
XMACRO not found	tried to delete an undefined macro	no
illegal flag	FLAGON: or FLAGOFF: argument must be an upper case letter	no
DEFINE missing left parenthesis	self-explanatory	no
DEFINE non-alphanumeric name	"	no
DEFINE missing comma	"	no
DEFINE missing right parenthesis	"	yes

INDEX

Index cont.