SLAC Computation Group
Stanford, California

CGTM NO. 150
July 1973

OBSOLETE —
SEE MORTRAN II CGTM 165

A User's Guide

to

MORTRAN

(MORTRAN I)

A. James Cook

ABSTRACT

MORTRAN is a FORTRAN language extension. Its features include:

- free-field format
- alphanumeric statement labels
- comments inserted anywhere in the text
- multiple assignments
- implied looping and block structure
- FOR-BY-TO, WHILE, UNTIL, and IF-THEN-ELSE statements
- abbreviations for some common FORTRAN constructions
- user-defined macro-instructions

The MORTRAN processor itself is written in ASA standard FORTRAN IV, so that it can be implemented on any computer that has a standard FORTRAN compiler.

INTRODUCTION

This document describes a language that incorporates many of the advantages of the more sophisticated high-level languages, while retaining most of FORTRAN's advantages as well. This language, called MORTRAN, has the unrestricted free-field card arrangement of the more advanced languages, as well as much of their flexibility, naturalness, and conciseness. As a result, this language is more convenient for coding, and is easier to read than FORTRAN. However, the basic structure of MORTRAN is very like FORTRAN so that programmers having FORTRAN experience can learn it quickly. (In fact, with a minor exception described in Section 11, all of FORTRAN can be considered a subset of MORTRAN, and the programmer may intersperse FORTRAN and MORTRAN text freely.)

The MORTRAN processor translates the MORTRAN input text directly to FORTRAN statements, which are then compiled by the user's FORTRAN compiler into machine language for execution. The MORTRAN processor uses four files, two for input and two for output: the input files are the initialization file and the source file, and the output files are the print file and the punch file.

1. MORTRAN begins by reading the initialization file, and sets itself up for processing. No further reference is made to this file thereafter.

2. The source file contains the MORTRAN source program and any appropriate MORTRAN control cards, of which the end-of-input control card (see below) is required.

3. The print file contains a listing of the source program, along with any diagnostics generated by MORTRAN. MORTRAN prints a number in the left margin of the listing for each card of the source program, which gives the "bracket nesting level" at the start of the card. Brackets (< and >) are heavily used in MORTRAN, and this information is helpful in keeping track of the logical structure of the program.

4. The punch file contains the generated FORTRAN program, which is to be directed to the user's FORTRAN compiler. As an aid, the MORTRAN source program is included among the generated statements on ordinary FORTRAN comment cards.

1. Description of Coding Rules

(A) MORTRAN Control Cards

MORTRAN control cards are distinguished from normal input cards by the appearance of a percent sign (%) in column one of the card image. MORTRAN recognizes four different control cards, of which only one is necessary. The four types are (1) the end-of-input card, (2) the FORTRAN interspersion switch card, (3) the line length reduction card, and (4) the macro-instruction definition card. Only the end-of-input card is required; it must be the last card of the input deck to MORTRAN, and it consists of the required percent sign in column one, and another percent sign (denoting end of input) in column two.

The other three types of MORTRAN control cards are described in sections 8, 9, and 12 below.

(B) Coding MORTRAN Text

MORTRAN is "free-field" in the sense that statements may begin anywhere on the 80-character input line (card image), and end anywhere on the same line or a succeeding line, with blanks imbedded freely.

1. All statements in MORTRAN are terminated by semicolons (;).

2. Comments are enclosed in quotation marks ("double quotes"), and may be inserted anywhere in the text.

3. Character strings comprising Hollerith fields are enclosed in apostrophes ('single quotes'). If an embedded apostrophe is desired as a character within a quoted string, use a pair of apostrophes to represent each such embedded apostrophe.

4. Sequences of statements, and certain special constructions, are enclosed in brackets (< and >).

5. The at-sign character (@) has special internal meanings in MORTRAN, and may be used only in quoted character strings.

Example 1.1:

```
COL 1 <--------------- INPUT CARD IMAGES ----------------> COL 80
|                                                              |

                        Y= SIN(THETA)
                          +COS(THETA)          - ( A  -
  "YOU"                                             BIG -
     "MAY"                                          LONG -
        "INSERT"                                     HAIRY -
            "COMMENTS"                                MESSY -
                 "ANY"                                 TERM )
  +A                "WHERE"
  +LOT                    "YOU"
  +OF                          "BUDDY"
  +JUNK       -(    -STILL             "DAMN"
                    -MORE                   "WELL"
                    -JUNK  / STUFF *          "PLEASE"  * AARGH) ;
```

is converted by MORTRAN to the following FORTRAN:

```
      COL 7 <------- GENERATED FORTRAN STATEMENT ------> COL 72
      |                                                      |

      Y=SIN(THETA)+COS(THETA)-(A-BIG-LONG-HAIRY-MESSY-TERM)+A+L
     XOT+OF+JUNK-(-STILL-MORE-JUNK/STUFF**AARGH)
```

Example 1.2:

```
      DIMENSION H(7); DATA H/'THIS IS A HOLLERITH FIELD.'/ ;
```

produces the FORTRAN statements

```
      DIMENSION H(7)
      DATA H/   26 HTHIS IS A HOLLERITH FIELD./
```

Note that apostrophes may also appear inside a quoted character string; for each apostrophe desired in the final string, write two successive apostrophes, as in the following example.

Example 1.3:

```
         DATA H/'''DON''T FORGET PAIRED QUOTES'''/;
```

produces the FORTRAN text

```
         DATA H/   28 H'DON'T FORGET PAIRED QUOTES'/
```

2. Alphanumeric Statement Labels

Any MORTRAN statement can be labeled with an alphanumeric character string of arbitrary length. Statement labels must precede the statement, and are enclosed in colons (:). Note that

labels (1) may contain any characters other than colons, semicolons, and apostrophes; (2) may begin with a digit (and may therefore be numeric); (3) may contain arbitrary numbers of blanks, which are ignored. Alphanumeric labels can be used in place of any valid application of FORTRAN numeric statement labels.

Example 2.1:

```
                                :DAMN: FORMAT(' IMPOSSIBLE');
 GOTO  ( :GOOD: , :BAD: , :RETCH: ) , N;
 GOTO    :NONE OF THOSE:;
                 :GOOD:A=B;                      GOTO:ANYWAY:;
                 :BAD:C=D;                       GOTO:ANYWAY:;
                 :RETCH: PRINT
                                :DAMN:;  STOP;
    :NONE OF THOSE: E=F;
          :ANYWAY: ET=CETERA;
```

is converted by MORTRAN to the following FORTRAN:

```
 60001 FORMAT(   11 H IMPOSSIBLE)
       GOTO( 60002 , 60003 , 60004 ),N
       GOTO 60005
 60002 A=B
       GOTO 60006
 60003 C=D
       GOTO 60006
 60004 PRINT 60001
       STOP
 60005 E=F
 60006 ET=CETERA
```

Example 2.2: Similarly, the MORTRAN statements

```
CALL SUB(X,Y,& :GOOD RETURN:); CALL BOOBOO; :GOOD RETURN: E=F;
```

are converted to the FORTRAN statements

```
      CALL SUB(X,Y,& 60007 )
      CALL BOOBOO
60007 E=F
```

3. Multiple Assignments

MORTRAN implements multiple assignments with the following statement:

```
<V1,V2,V3,...Vn> = expression;
```

where V1, V2, ..., Vn are any FORTRAN variables (subscripted or not) and the expression is any valid FORTRAN expression. The order of assignment is left to right, as illustrated in the following example:

Example 3.1:

```
<I,A,J> = SQRT(ZZZ/2.5);
```

is converted by MORTRAN into the following FORTRAN:

```
I=SQRT(ZZZ/2.5)
A=I
J=A
```

The usual FORTRAN rules concerning assignments and implicit conversions must be observed for each of the variables in the multiple assignment list.

4. Blocks

A block is a sequence of MORTRAN statements enclosed in angle brackets, < and >. Let S1;S2;...Sn; represent a sequence of any valid MORTRAN statements. Then a block is represented in MORTRAN by

```
<S1;S2;...Sn;>
```

Blocks may be nested, in that any of the statements in the sequence may themselves be composed of blocks.

5. Conditional Statements

The simplest form of conditional statement in MORTRAN is written

```
<IF> logical-expression <T1; T2; ... Tm;>
```

Example 5.1: The MORTRAN statement

```
<IF> A.GT.B <A=B; CALL ARRGH;>
```

would be translated into the following FORTRAN:

```
      IF(.NOT.(A.GT.B)) GOTO 10001
      A=B
      CALL ARRGH
10001 CONTINUE
```

Note that by omitting the <IF> and enclosing the logical expression in parentheses, this simple form can also be written

```
(logical-expression) <T1; T2; ...Tm;>
```

It can be seen that the simple form

(logical-expression) T1;

is equivalent to the standard FORTRAN Logical IF statement, where T1 is the single statement to be executed if the logical-expression is true.

MORTRAN implements a general IF-THEN-ELSE conditional statement with the following:

<IF> logical-expression <true-block> <ELSE> <false-block>

where only the "true-block" of statements is executed if the logical-expression is true, and the "false-block" of statements is executed if it is false. The logical-expression must be a valid FORTRAN logical expression, and the statements in the blocks may be any valid MORTRAN statements including other blocks and other conditional statements.

Example 5.2: The MORTRAN text

(logical-expression) <true-block> <ELSE> <false-block> produces

```
      IF(.NOT.(logical-expression))GOTO 10005
      true-block
      GOTO 10006
10005 CONTINUE
      false-block
10006 CONTINUE
```

Note that the <ELSE> should appear only after a block following an <IF>; improper uses of <ELSE> will usually result in undefined or duplicate labels in the generated FORTRAN text.

As in the simple cases above, the <IF> preceding the logical expression is optional and may be omitted, in which case the logical expression must be enclosed in parentheses:

(logical-expression) <true-block> <ELSE> <false-block>

MORTRAN also implements an UNLESS-THEN-ELSE statement, which is the logical converse of the IF-THEN-ELSE conditional:

<UNLESS> logical-expression <false-block> <ELSE> <true-block>

As in the case of the IF-THEN-ELSE conditional statement, the <ELSE> and the following block are optional, in which case the statement has the form

<UNLESS> logical-expression <false-block>

It is of course possible to nest blocks and conditional statements in MORTRAN, as the following example will illustrate. Suppose we write

```
(logical-expression-1) <(logical-expression-2) <block1>  block2>
```

then MORTRAN would generate

```
       IF(.NOT.(logical-expression-1))GOTO 10002
       IF(.NOT.(logical-expression-2))GOTO 10003
       block1
 10003 CONTINUE
       block2
 10002 CONTINUE
```

Examples

```
(ALOG(E) .EQ. SQRT(F)) <E=F**2;> <ELSE>
    <(E.LT.0.5) <E=5.0; Q=E**3;>
          <ELSE> <GO TO :SOMEWHERE:>>

<UNLESS> A.EQ.B+SIN(C) <A=COS(C); B=A;>
```

6. Looping

Looping is the repeated execution of a block of statements. MORTRAN allows both explicit and implicit looping.

(A) Implicit Looping

MORTRAN accomplishes implicit looping with the control statements:

```
<WHILE> logical-expression <block>

<UNTIL> logical-expression <block>
```

The "logical-expressions" above can be any valid FORTRAN logical expressions. For the <WHILE> statement, the block of statements is executed only while the logical expression is true, whereas in the case of the <UNTIL> statement the block is executed while the logical expression is not true. Thus, the <UNTIL> statement is simply the converse of the <WHILE> statement. The test as to whether to execute the block is made before the block is entered, so that it is possible never to enter the block. For example, consider the following:

Example 6.A.1: The statement

```
<WHILE> logical-expression <block>
```

produces the FORTRAN statements

```
10014    IF(.NOT.(logical-expression))GOTO 10015
         block
         GOTO 10014
10015    CONTINUE
```

The <UNTIL> statement produces similar FORTRAN code:

```
10016    IF(logical-expression)GOTO 10017
         block
         GOTO 10016
10017    CONTINUE
```

The statements comprising the blocks may be any valid MORTRAN statements including blocks, IF-THEN-ELSE, UNLESS, and other looping control statements. Thus, looping can be nested to any depth. There are no restrictions for branching into or out of a block under implicit looping control.

(B) Explicit Looping

MORTRAN implements explicit looping with the <FOR> and <DO> statements. The <FOR> statement has three forms, and the <DO> statement has four:

```
<FOR> var = expr1 <BY> expr2 <TO> expr3 <block>
<FOR> var = expr1 <TO> expr3 <BY> expr2 <block>
     <FOR> var = expr1 <TO> expr3 <block>

<DO> index = m1, m2, m3 <block>
  <DO> index = m1, m2 <block>
  <index = m1, m2, m3; block>
    <index = m1, m2; block>
```

These two statements differ in a very important respect: the MORTRAN <DO> statement generates an exactly equivalent FORTRAN DO statement, whereas the MORTRAN <FOR> statement generates a more complex and general set of statements. Omitting the <BY> parameter "expr2" in the <FOR> statement, and omitting the parameter "m3" in the <DO> statement, are equivalent to specifying a default value of 1 for the increment parameter.

Example 6.B.1: The MORTRAN text

```
<DO> K = N1, N2 <block>
```

produces the FORTRAN text

```
         DO 10011 K=N1,N2
         block
10011    CONTINUE
```

To show why the <FOR> statement is more general, the following example shows the expansion of a typical MORTRAN <FOR>:

Example 6.B.2: <FOR> var = expr1 <BY> expr2 <TO> expr3 <block>

generates the FORTRAN statements

```
          var = expr1
          GOTO 10018
10019     var = var + (expr2)
10018     IF((expr2)*((var)-(expr3)).GT.0)GOTO 10020
          block
          GOTO 10019
10020     CONTINUE
```

In the <FOR> statement, the induction variable "var" represents any valid FORTRAN real or integer variable (subscripted or not) and expr1, expr2 and expr3 are any valid FORTRAN arithmetic expressions. This statement causes "var" to be set initially to the value of expr1. Then, if this value minus the current value of expr3 times the sign of expr2 is not positive, the block is entered. After the sequence of statements in the block is completed, "var" is incremented by the current value of expr2, and if the preceding condition is still met with this new value for "var", the block is re-entered. The variable "var" is thus incremented and the block is re-executed until the the condition is not met. Since the test is made <u>before</u> entering the block, it is possible to never enter the block. The expressions expr2 and expr3 are re-evaluated on each test, so that it is possible to change their values as well at the value of "var" within the block. There are no restrictions on branching into or out of the block.

The MORTRAN DO-statement is completely analogous to the FORTRAN DO-statement. As in the FORTRAN DO-loop, the induction variable, "index", is set to the value of M1 and the block is executed. "index" is then incremented by M3 and the block is re-executed until the value of "index" exceeds that of M2. M3 is optional, and if omitted is assumed to have the value +1. The block is always executed at least once. The parameters that control the loop must obey the same restrictions as the corresponding FORTRAN DO-loop parameters: they must be integer constants or non-subscripted integer variables. Also, the usual FORTRAN restrictions regarding branching into or out of the block are present.

In MORTRAN, the following implied notation can also be used for DO-looping:

```
<index = m1, m2, m3 ; block>
   <index = m1, m2; block>
```

When this notation is used as the "block" following an IF construction, <u>both</u> brackets are needed. Thus,

<u>Example 6.B.3</u>: `(N .GT. 0) << K = 1, N; A(K) = B; >>` produces

```
          IF(.NOT.(N.GT.0))GOTO 10062
          DO 10063 K=1,N
          A(K)=B
10063     CONTINUE
10062     CONTINUE
```

Examples:

```
<FOR> KK = M1 <BY> M3 <TO> M2 <A(KK) = B(KK);>

<DO> KK = M1, M2, M3 <A(KK) = B(KK);>

<KK = M1, M2, M3; A(KK) = B(KK);>

<FOR> I = M1 <BY> -M2 <TO> M3 <A(I) = B(I); C(I) = D(I);>

<FOR> R = 5.*X**3 <BY> -SQRT(Q) <TO> -SIN(THETA) <Q = COS(R);
    THETA = ATAN(R**2/Q);>
```

7. I/O Abbreviations

MORTRAN allows simple abbreviations for FORTRAN input and output statements referring to the standard system input and output files, by the statements:

```
<R> list; (format field);
<W> list; (format field);
```

where "list" is a valid FORTRAN I/O list, and the "format field" contains valid FORTRAN format codes. The format to be applied to the abbreviated read or write must immediately follow the read or write statement. As in FORTRAN, the I/O list is optional. This abbreviation is useful, for example, for inserting debugging WRITE statements; note that the format field generates a statement which cannot be referenced by other I/O statements. FORMAT statements may be named by alphanumeric labels, as illustrated in Example 2.1.

Example 7.1: `<R> (CARD(I),I=1,80); (80A1);`

8. Mixing MORTRAN And FORTRAN

FORTRAN text can be interspersed freely with MORTRAN text. This is accomplished by switching the MORTRAN processor to a "FORTRAN mode" by inserting a line (card) that is blank except for a percent sign (%) in the first column. While in the FORTRAN mode, the processor simply copies the input text to its output file for input to the FORTRAN compiler. MORTRAN does no processing on the text it reads while in FORTRAN mode. At any point in reading the input stream, the processor can be switched back to the MORTRAN mode by inserting another line (card) that is blank except for the percent sign in the first column. This %-line simply acts as a toggle for switching the processor back and forth between the two modes; it always starts in the MORTRAN mode.

Example 8.1

```
"THIS IS SOME MORTRAN TEXT" A=EXP(FXB);
%
C    NOW IN FORTRAN MODE
          CALL AJCLJS(JHF,JBE)
          CALL SLACCG(SF)
%
"NOW BACK IN MORTRAN MODE"          WHAT=A(RELIEF);
```

This switching can take place anywhere in the input text, and need not be confined to program or subprogram boundaries. However, if there is a mode switch within a program or subprogram, all user-defined FORTRAN numeric statement labels should be less than 10000 within that routine, because MORTRAN uses five-digit generated labels.

9. Line Length Reduction

As mentioned above, MORTRAN allows totally free-field input in columns 1 to 80 of the card image. This default line length can be arbitrarily changed at any point in the input stream by inserting a MORTRAN control card containing a percent sign in column 1, a letter "C" in column 2, and a non-blank character in the column which is to be the last column scanned by MORTRAN in the following source cards. Thus, the control card

```
%C                    X  <---   (X is in column 20)
```

would set the line length to 20 characters. This will be the new line length until it is again changed in the same manner. The MORTRAN processor will ignore columns on each line in excess of the line length. This excess part of each line may be used for comments or other descriptive information.

10. Diagnostics and Errors

The MORTRAN processor checks for certain simple errors and inconsistencies in its input text, and issues the following error message when they are detected:

```
**********ERROR n
```

Here, n is an integer describing the type of error:

n = 1 missing right colon (:), or label length too great for internal storage

n = 2 missing left bracket (<), or too many right brackets (>); or, missing right bracket (>), or too many left brackets (<)

n = 3 missing semicolon (;), or out of space

n = 4 unclosed quoted string (missing apostrophe)

n = 5 invalid syntax in a macro-instruction definition

A missing semicolon (;) usually results in FORTRAN errors that are easy to detect. A missing quotation mark (") will result in comments being treated as program and vice versa; such errors are usually easy to find. The maximum bracket (<,>) nesting level allowed by MORTRAN is currently set to 80.

11. Exception for Hollerith Fields

There is a single exception to the rule that all valid FORTRAN statements are also valid MORTRAN statements. MORTRAN will not process correctly explicit Hollerith fields that contain blanks. Statements of the form

nH character string

will have all embedded blanks within the character string suppressed. In MORTRAN, all Hollerith fields (character strings) must be enclosed in apostrophes ('single quotes'), as in Example 1.2. In those situations where explicit Hollerith fields are necessary, the MORTRAN processor can be switched to its FORTRAN mode for correct handling of those statements.

12. User-Defined Macro-Instructions

MORTRAN is implemented as a set of macro-instruction definitions that are matched against the user's input text. Because MORTRAN is written to accept a very general set of macro-instruction definitions, it is possible for the MORTRAN programmer to make use of this facility to define his own macro-instructions.

A user-defined macro-instruction is written on a single MORTRAN control card, in the following form:

%'pattern-recognition-text'='substitution-text'

The percent sign must appear in column one as usual. The first quoted string of characters following the percent sign is the pattern-recognition-text, or pattern-text. This is the pattern of characters to be recognized by MORTRAN as a request for replacement by the substitution-text. The pattern-text is followed by an equal sign. This is followed by a second quoted character string, which defines the substitution-text.

To give a simple example of a macro-instruction definition that illustrates some of the terminology and notation, suppose we want to define a macro-instruction which will allow the

programmer to define the length of an array at the time the program is being compiled. If we let the characters "ARRAYSIZE" stand for the desired array length, then we could write a macro-instruction definition of the following form:

```
%'ARRAYSIZE'='4000'
```

The pattern-recognition-text is the characters "ARRAYSIZE", and the substitution-text is the characters "4000". Thus, the sequence of MORTRAN statements

```
%'ARRAYSIZE'='4000'
DIMENSION ARRAY(ARRAYSIZE); <DO> I=1,ARRAYSIZE <ARRAY(I)=0.;>
```

would produce the FORTRAN statements

```
      DIMENSION ARRAY(4000)
      DO 10026 I=1,4000
      ARRAY(I) = 0.
10026 CONTINUE
```

It can be seen that all occurrences of the string "ARRAYSIZE" have been replaced by the string "4000".

13. Advanced Uses of the Macro-Instruction Facility

The features of MORTRAN described up to this point allow the programmer considerable flexibility and ease in writing programs. The facility to be discussed in the following sections is somewhat more difficult, and need not be understood in order to make effective use of MORTRAN.

To be able to make more elaborate uses of macro-instructions, it is important to understand the scanning mechanism used by MORTRAN in matching pattern-texts against input character strings. First, MORTRAN scans the input stream, deleting comments and blanks, up to the next semicolon. Then the statement is scanned for matches to the pattern-texts of user-defined macro-instructions (if any) followed by the standard MORTRAN definitions. If any match occurs, the matched part is replaced by its corresponding substitution-text. One of two things can then occur: if (as is usual) the substituted text is to be rescanned, pattern matching begins anew starting with the first character of the generated text; otherwise, if no rescanning is desired, the generated text is put out, and pattern matching resumes at the point in the input following the text segment that matched the pattern.

To give a more elaborate example of a macro-instruction definition, suppose we wish to define an "ASSIGN" statement, so that we can write statements like

```
<ASSIGN> SQRT(B) <TO> A;
```

The MORTRAN macro-instruction definition which implements this statement is:

```
%'<ASSIGN>#<TO>#;'='#2=#1;'
```

The pattern-recognition-text is the string of characters "<ASSIGN>#<TO>#;", and the substitution-text is the string of characters "#2=#1;".

The number-sign character (#) is used in two distinct and important ways. When it occurs in the pattern-text, the character "#" represents a variable-length character string. In this example, it will match any character string in the input stream which follows the characters "<ASSIGN>", up to the first subsequent occurrence of the character "<". Thus, the "#" character may be thought of as denoting a parameter in the pattern-text. Similarly, the second "#" in the pattern-text matches whatever character string appears in the input stream following the characters "<TO>" up to the first semicolon thereafter. In this example, there are two parameter strings in the macro-instruction definition.

In the substitution-text, the positions into which the character strings matching the parameters are to be substituted are indicated by the character "#" followed by a single decimal digit: the string matching the first "#" in the pattern-text is denoted by "#1", the second by "#2", and so forth. In this example, the substitution-text specifies that the second parameter string is to be followed by an equal sign, which is then followed by the first string, which is then followed by a semicolon. Thus, the MORTRAN statements

```
%'<ASSIGN>#<TO>#;'='#2=#1;'
"TEST <ASSIGN> MACRO"          <ASSIGN> SQRT(B) <TO> A;
```

would generate the MORTRAN text

```
A=SQRT(B);
```

Note that the blanks in the input statement have been deleted: otherwise, the blanks would have appeared in the generated text, as "A= SQRT(B) ;". It is important to remember that blanks are removed from the input stream as they are read by MORTRAN.

Now, suppose that this macro-instruction definition had been written without the angle brackets on the words "ASSIGN" and "TO":

```
%'ASSIGN#TO#;'='#2=#1;'
```

Then, the MORTRAN statements

```
%'ASSIGN#TO#;'='#2=#1;'
"TEST ASSIGN MACRO"            ASSIGN SQRT(B) TO A;
```

would generate the unexpected (and incorrect) MORTRAN text

```
ASSIGNSQRT(B)TOA;
```

for the following reason. In scanning the input text, MORTRAN would correctly match the characters "ASSIGN" to the start of the definition. Then, the characters "SQR" would match the first parameter character "#". This is because MORTRAN terminates the variable-length scan for a parameter when it finds the first character to match the single character following the "#" in the pattern. However, the next character in our sample pattern-text, the letter "O", does not match the next character from the input stream, "(". Because MORTRAN does no "backtracking" in its pattern matching, it will assume that this pattern does not match the input stream, and will discard all tentative results before trying another pattern.

Even if such a pattern-matching problem could be avoided, this definition of an ASSIGN-TO macro-instruction would have to be used with caution, because "ASSIGN" and "TO" are valid in other FORTRAN contexts. Assuming the above definition, the statement

```
ASSIGN(I,J) = STORE(K);
```

would generate the text

```
BE(K)=(I,J)=S;
```

which is probably not what was desired.

Another brief example of a macro-instruction definition will illustrate a problem that can arise if great care is not used in writing the definition. Suppose we are writing a program in which a statement like "K=K+1;" appears many times, and we want to use the shorthand notation "+K;" instead. It might be tempting to write a definition of the form %'+#;'='#1=#1+1;', in which the pattern-recognition-text is "+#;". That is, we want to recognize a plus sign, followed by some string of characters representing the variable to be incremented, followed by a semicolon.

To show why this leads to trouble, we will examine each step of the substitution process. Suppose the input stream contains the characters "+JJ;". This will match the pattern-text of our definition; the "#" in the pattern-text will be matched by the two characters "JJ". On substituting them in place of "#1" in the substitution-text, the output string would contain the characters "JJ=JJ+1;", as desired. Now, observe that the final three characters of this piece of generated text are a match for the original pattern! In this example, MORTRAN will rescan the generated text, and again match "+1;", so that those three characters would be replaced by the new segment of substituted text, creating the string "JJ=JJ1=1+1;". This, in turn, will match the pattern-text again, the generated text will become

"JJ=JJ1=11=1+1;", and so on, indefinitely. MORTRAN will eventually run out of space, and come to an untimely end.

There are two means available to avoid such catastrophes. The first, and the best, is to observe that (because blanks and comments are removed from the input) the character preceding the characters "+JJ;" must be a semicolon; thus if the macro-instruction definition was written %';+#;'=';#1=#1+1;', the indefinite recursion would be avoided. The second means available is to tell MORTRAN explicitly that the generated text is not to be rescanned after it has been created. This is indicated by using two equal signs to separate the pattern-text and the substitution-text: thus we could write %'+#;'==#1=#1+1;' and the recursion would be avoided. The latter technique is not recommended in general, however, because you may lose one of the more important abilities of MORTRAN. Because MORTRAN is itself defined by a set of macro-instructions, it often depends on many levels of expansion in order to be able to create the correct output text. If at some point in this process no rescan is done, some important segment of text might not be properly expanded.

14. Macro-Instruction Considerations

The following aspects of the macro-instruction facility should be kept in mind when writing macro-instruction definitions:

(1) Blanks and comments (enclosed in quotation marks) are deleted from the input stream as it is read. Thus comments are not searched for matches to the pattern-text.

(2) The character "@" has special meanings in MORTRAN, and should not be used anywhere, except in quoted strings. (Further details are given in the MORTRAN Installation and Maintenance Guide.)

(3) The order of scanning of macro-instruction definitions is "new before old", so that a macro-instruction may be redefined as many times as you like. Old definitions are not discarded, however, so that it is possible to run out of space if too many definitions are used.

(4) Macro-instruction definitions may occupy more than one (80-character) card image, and the length of the definition is not controlled by the line-length reduction control card (%C). If more than one card is needed, the definition continues in column 1 of the following card; do not put a percent sign in column 1.

(5) More than one macro-instruction definition may appear on a single card. The definitions are separated by a comma, as in

```
%'A'='B','C'='D','E'='F'
```

The last definition is terminated by a blank.

(6) A missing apostrophe in a macro-instruction definition will usually result in some sort of disaster. In addition, apostrophes (paired or not) may not appear within the body of either the pattern-text or the substitution-text.

(7) If there are blanks in the pattern-recognition-text, they can only be matched by blanks in generated text, since blanks have been squeezed out of the primary input stream.

(8) Excess parameters in the substitution-text (such as the use of "#9" in a macro-instruction containing only one "#" symbol in the pattern-text) will usually cause major errors.

(9) The "#" symbol in the macro-instruction pattern-text will match text between balanced parentheses (including any special characters) other than an apostrophe or a semicolon. An apostrophe or semicolon always stops the scan for text to match the "#" symbol. The pattern-text therefore cannot contain an embedded apostrophe or semicolon. (Text in balanced parentheses is collected so that subscripted variables and function calls, for example, will match the same pattern-text as a simple variable.)

(10) Paired equal signs (==) separating the pattern-recognition-text and the substitution-text mean that the generated substitution-text is not to be rescanned immediately upon generation.

(11) To facilitate pattern matches for macros, an implicit semicolon is placed ahead of the input stream initially.

(12) Quoted character strings (enclosed in apostrophes) are not scanned for matches to pattern-recognition-texts.

(13) MORTRAN does not backtrack in its pattern matching. Thus, the macro-instruction definitions

```
%'F?'='<FOR>','T?'='<TO>'
F?INDEX=1T?100; "SHOULD TURN INTO: <FOR>INDEX=1<TO>100;"
```

will not work correctly, since the characters "F?" will generate "<FOR>"; upon rescanning, this will fail to match the standard definition of a <FOR> statement, since the "<TO>" has not yet been substituted in the rest of the statement.