Computation Research Group

CTGM No. 165
Feburary 1975

Revised
June 1975

A User's Guide to MORTRAN2

A. James Cook
and
L. J. Shustek

Computation Research Group
Stanford Linear Accelerator Center
Stanford, California, 94305

1. INTRODUCTION

The term MORTRAN, like FORTRAN, has several meanings, depending upon the context in which the term is used. MORTRAN has come to mean

- A Structured Language
- A Translator for that language
- A Macro-processor

The structured language is implemented as a set of macros which are used by the macro processor to translate the language into FORTRAN. The resulting FORTRAN program is then run like any other FORTRAN program. User-defined macros are easily added to the standard (language-defining) set of macros so that the language is "open-ended" in the sense that extensions to the language may be made at any time by the user. Extensions have ranged from very simple ones like matrix multiplication, to complex ones like those which define new data types.

The user need not concern himself with the method of implementation or the macro facility in order to take advantage of the structured language which is provided by the standard set of macros. The features of this language include

- Free-field (column and card boundaries may be ignored)
- Alphanumeric labels of arbitrary length
- Comments inserted anywhere in the text
- Nested block structure
- Conditional statements which may be nested
  (IF, IF-ELSE, and ELSEIF)
- Loops (repetitively executed blocks of statements) which test for termination at the beginning or end or both or neither
  (WHILE, UNTIL, FOR-BY-TO, LOOP and DO)
- EXIT (jump out of) any loop
- NEXT (go to NEXT iteration) of any loop
- Multiple assignment statements
- Conditional (alternate) compilation
- Program listing features include
  Automatic printing of the nesting level
  Automatic indentation (optional) according to nesting level
- Abbreviations for simple I/O statements
- Interspersion of FORTRAN text with MORTRAN text

The user may elect to override the standard set of macros and write a set which defines another, perhaps "problem oriented", language.

The MORTRAN2 processor is a FORTRAN program of approximately 800 statements; the macros which define the structured language require about 50 cards. The 1966 ANSI standard has been observed throughout, so that transportability of the processor is assured. Certain non-FORTRAN characters (such as the semicolon and the apostrophe) are used as delimiters in the MORTRAN language, but these are read in by the MORTRAN2 processor during initalization as part of the standard macro set and may be changed to suit various machines without modifying the processor.

2. CODING RULES

MORTRAN programs may be written without regard to column or card boundaries. Statements may begin anywhere on the input line (card-image), and may end anywhere on the same line or on a succeeding line. The end of a statement is determined by a semicolon (;). This feature permits "free-field" or "free-form" programming. Normally, only the first 72 columns of the input line are intepreted as program text, but this can be changed. (See appendix A.)

Character strings comprising Hollerith fields are enclosed in apostrophes (as in 'THIS IS HOLLERITH DATA'). If an embedded apostrophe is desired as a character within a quoted string, use a pair of apostrophes to represent each such embedded apostrophe (as in 'DON''T').

Comments in MORTRAN are enclosed in quotation marks (as in "COMMENT") and may be inserted anywhere in the program (except in character strings or macros).

In MORTRAN, an alphanumeric label is a character sequence of arbitrary length enclosed in colons (as in :TOMATOES:). The characters which comprise the sequence may be any combination of letters and digits. An alphanumeric label may be used anywhere a FORTRAN statement label is allowed.

Multiple blanks (a sequence of two or more blanks) in a MORTRAN program are equivalent to a single blank except in quoted strings, where all blanks are preserved, and in macros (discussed in section 5).

Summary of coding rules:

- Terminate statements with a semicolon (;).
- Enclose comments in quotation marks (").
- Enclose labels in colons (:).
- Enclose character strings in apostrophes (').
- Blanks may be inserted freely except in labels, character strings and user-defined macros.

It should be pointed out that any extensions provided by a particular FORTRAN compiler may be used, provided that they do not conflict with MORTRAN's coding conventions. However, if transportability of the the programs being written in MORTRAN is a consideration, the ANSI FORTRAN standard should be adhered to. The standard set of macros which define the language described in this Guide do not generate non-ANSI FORTRAN.

3. STRUCTURE

A. Statements

FORTRAN may be regarded as a subset of MORTRAN, since (with a minor exception noted in appendix D) any valid FORTRAN statement becomes a valid MORTRAN statement when

- it is terminated by a semicolon, and
- continuation marks (if any) are deleted.

B. Blocks

A MORTRAN block is a sequence of MORTRAN statements enclosed in the special characters < and >, which we will call "brackets". The left bracket may be read "begin" and the right bracket may be read "end". Let

S1; S2; S3;...Sk;...Sn; (3.1)

be a sequence of statements. The sequence becomes a block when it is enclosed in brackets

< S1; S2; S3;...Sk;...Sn; > (3.2)

(The ellipses (...) are meta-symbols indicating arbitrary repetition. The brackets are not meta-symbols; they are delimiters in the MORTRAN language.)

Blocks may be nested. That is, any of the statements in a block may be replaced by a block. For example, in (3.2) we could replace Sk by a block and write

< S1; S2; S3;...< T1; T2; T3;...Tm; >...Sn; > (3.3)

The block containing the sequence T1;...Tm; is completely contained, or nested, within the block containing the sequence S1;...Sn;. We will frequently write an ellipsis enclosed in brackets <...> to denote a block.

Example of a block: < X=Y; CALL SUB(A); B=1; >

C. Conditional Statements

The simplest form of a conditional statement in MORTRAN is written

```
IF e <...>                                    (3.4)
```

where e is an arbitrary logical expression, and the ellipsis enclosed in brackets denotes a block as described above. If e is true then the statements in the block are executed. If e is false, control is transferred to the first statement following the block. For example:

```
IF A .LT. B < C=D; E=F; > G=H;
```

If A is less than B then the statements C=D and E=F are executed after which G=H is executed. If A is not less than B control is transferred directly to the statement G=H.

Next in complexity is the IF-ELSE statement, which is written

```
IF e <...> ELSE <...>                         (3.5)
```

If e is true then the statements in the first block are executed and control is transferred to the statement following the second block. If e is false then the statements in the second block are executed and control is transferred to the statement following the second block. For example, consider

```
IF A .LT. B <C=D; E=F;>
     ELSE   <G=H; I=J;>

K=L;
```

If A is less than B the statements C=D and E=F are executed after which control is transferred to the statement K=L. If A is not less than B the statements G=H and I=J are executed after which control is transferred to the statement K=L.

Consider

```
IF A.EQ.B < X=Y;>
```

Here, the block to be executed whenever A is equal to B consists of the single statement X=Y;. An alternate form acceptable in MORTRAN is the standard FORTRAN logical IF

```
IF (A.EQ.B) X=Y;
```

IF-ELSE statements may be nested to any depth. Even so, the IF-ELSE is not really adequate (in terms of clarity) for some problems that arise. For example, consider the following "case analysis" problem:

Suppose that we have

four logical expressions, p, q, r, and s,
and five blocks of statements, A, B, C, D, and E.

Now suppose that p, q, r, and s are to be tested sequentially. When the first TRUE expression is found we want to execute the statements in the corresponding block (p corresponds to A, q to B, etc.) and then transfer control to the statement following block E. If none of them is true we want to execute block E. Using nested IF-ELSE statements we could write

```
IF p <A>
ELSE < IF q <B>
       ELSE <  IF r <C>
               ELSE <  IF s <D>
                       ELSE <E>
                    >
            >
     >
```

While this does what we want, it is awkward because each ELSE increases the level of nesting. MORTRAN offers the ELSEIF statement as an alternative:

```
IF     p  <A>
ELSEIF q  <B>
ELSEIF r  <C>                                    (3.6)
ELSEIF s  <D>
ELSE      <E>
```

Using ELSEIF instead of ELSE allows all the tests to be written at the same nest level.

In summary, an IF statement may be optionally followed by any number of ELSEIF clauses which in turn may be optionally followed by a single ELSE clause.

D. Iteration

A MORTRAN loop is a block which is preceded by, and optionally followed by a "control phrase".

One such phrase is " WHILE e ". One of the loops we may write with this phrase is

```
WHILE e  <...>                                    (3.7)
```

The logical expression e is tested first. If e is true the block is executed and then control is returned to test e again. When e becomes false, control is transferred to the first statement following the block.

Another control phrase is " LOOP ". If we wanted to test at the end of the loop instead of the beginning, we could write

```
LOOP <...> WHILE e ;                              (3.8)
```

In (3.8) the block is executed first. Then, if the logical expression e is true, the block is executed again. When e becomes false control is transferred to the statement following the loop (that is, the statement following the " WHILE e ;").

The logical converse of the WHILE loop is the UNTIL loop.

```
UNTIL e  <...>                                    (3.9)
```

The logical expression e is tested first. If e is false the block is executed and then control is returned to test e again. When the logical expression becomes true, control is transferred to the first statement following the block. Similarly, the logical converse of (3.8) may be written by replacing the WHILE in (3.8) by UNTIL.

Tests for loop termination may be made at both ends of a loop. For example, if e and f are logical expressions

```
WHILE e <...> UNTIL f ;
WHILE e <...> WHILE f ;

UNTIL e <...> WHILE f ;
UNTIL e <...> UNTIL f ;
```

all test at both the beginning and the end. The above list is by no means exhaustive, but we must develop other "control phrases" in order to complete the discussion.

The iteration control phrases discussed above do not involve "control variables", that is, variables whose values are automatically changed for each execution of the loop. The following loop involves a control variable:

```
FOR v = e TO f BY g <...>                    (3.10)
```

where v is the control variable, and e, f, and g are arbitrary arithmetic expressions. The control variable v must be of type REAL or INTEGER, and may be an array element (subscripted variable). The value of any of the arithmetic expressions may be positive or negative. Moreover, the magnitudes as well as the signs of f and g may change during the execution of the loop.

The control variable v is set to the value of e and the test for loop termination (see below) is made. If the test is passed then the block is executed, after which v is incremented by the value of g and control is returned to the test. Note that the block is never executed if the test fails the first time.

The "test" for the termination of a FOR loop refers to the logical expression

```
g * (v-f) .GT. 0                             (3.11)
```

If the value of (3.11) is true then the test is said to have failed and control is transferred to the statement following the loop. Multiplication by g in (3.11) assures that loops in which the increment is (or becomes) negative will terminate properly.

The "FOR loop" (3.10) has two alternate forms

```
FOR v = e BY g TO f <...>                    (3.12)
```

and

```
FOR v = e TO f <...>                         (3.13)
```

In (3.13) no increment is given, so it is assumed to be one.

The iteration control phrase " DO I=J,K,N " also involves a control variable. In this case I,J,K, and N must all be of type INTEGER and may not be array elements or expressions. (These are the standard FORTRAN rules for DO loops). The following generates a standard FORTRAN DO loop:

```
DO I=J,K,N <...>                             (3.14)
```

There is one exception to the rule that loops must be preceded by control phrases, namely the compact DO-loop notation

```
<I=J,K,N;...>                                (3.15)
```

which generates a standard FORTRAN DO loop. This form permits compact notation for nests like <I=1,N1; <J=1,N2; <K=1,N3; A(I,J,K)=exp; >>>. (The use of the compact DO-loop notation is controversial; some people feel that it obscures the loop control. If desired, it can be be removed from the language by deleting a single macro from the standard set.)

The MORTRAN FOR and DO loops apply only to blocks of statements, not to I-O lists. The usual FORTRAN implied DO should be used within READ or WRITE statements.

There remains one more type of loop to be discussed. This loop is sometimes referred to as the "forever loop". One writes the forever loop in MORTRAN by preceding a block with the phrase "LOOP".

```
                    LOOP <...> REPEAT                    (3.16)
or simply           LOOP <...>                           (3.17)
```

The block is executed, and control is transferred back to the beginning of the loop. The optional phrase "REPEAT" in (3.16) is sometimes useful as a visual aid in locating the ends of deeply nested loops.

A reasonable question might be: "How do you get out of a forever loop? Or, for that matter, any of the loops?" One rather obvious way is to write

```
GO TO :CHICAGO: ;
```

where the label :CHICAGO: is on some statement (or block) outside the loop. If a convenient label doesn't already exist, creating one for the sole purpose of jumping out of the loop can be annoying and distracting. For the case in which the jump is to the statement following the loop, the GO TO may be replaced by the MORTRAN "EXIT;" statement, which is written

```
                    EXIT;                                (3.18)
```

or, with a conditional statement

```
                    IF (e) EXIT;                         (3.19)
or                  IF  e  <...EXIT;>                    (3.20)
```

In any MORTRAN loop, the occurrence of the statement "EXIT;" causes a transfer of control to the first statement following the loop in which it occurs.

A companion to the "EXIT;" statement is the "NEXT;" statement, which is written

```
                    NEXT;                                (3.21)
```

or, with a conditional statement

```
                    IF (e) NEXT;                         (3.22)
or                  IF  e  <...NEXT;>                    (3.23)
```

The occurrence of a "NEXT;" statement (which is short-hand for "go to the next iteration of this loop") in any MORTRAN loop causes a transfer of control to the beginning of the loop in which it occurs, incrementing the control variable (if any) before making the test for continuation in the loop. In loops which test at both ends of the loop, only the test at the beginning of the loop is made; tests at the end of the loop are made only when the end of the loop is reached. The tests of control variables in FOR and DO loops are considered to be at the beginning of the loop. (For more detailed information refer to the flow-charts in Appendix B.)

Any MORTRAN loop may be optionally preceded by a label. We will call loops which are preceded by labels "labeled loops". Any EXIT or NEXT statement may be optionally followed by a label. Any labeled loop may contain one or more statements of the form

EXIT :label: ; (3.24)

which transfers control to the first statement following the labeled loop. For example EXIT :ALPHA:; would transfer control to the statement following the loop which had been labeled :ALPHA:. This transfer of control takes place regardless of nesting, and thus provides a "multi-level" EXIT capability. The statement

NEXT :label: ; (3.25)

transfers control in the manner described above for the NEXT; statement.

Suppose we have a nest of loops which search some arrays. The outer loop has been labeled :SEARCH:, and two of the inner loops have been labeled :COLUMN:, and :ROW:. Now we may write

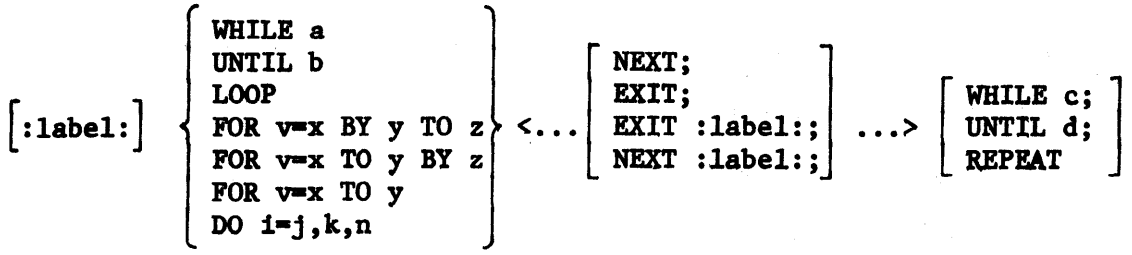
NEXT :ROW: ; or NEXT :COLUMN: ; or EXIT :SEARCH: ;

when the transfer involves more than one level of nesting, or

NEXT; or EXIT;

when only one nest level is involved. Of course, the form EXIT :label:; may also be used to exit a single level if desired.

We can now summarize MORTRAN loops in the following chart:



where a, b, c, and d are arbitrary logical expressions,
x, y, and z are arbitrary arithmetic expressions,
v is a subscripted or non-subscripted variable of type INTEGER or REAL,
j, k, and n are non-subscripted INTEGER variables or INTEGER constants, and
i is a non-subscripted INTEGER variable.

{ } indicates "choose one",

[ ] indicates "optional", and

... indicates a (possibly null) sequence of statements.

4. MISCELLANEOUS FEATURES

A. Multiple Assignment

It is sometimes useful to be able to assign the value of some expression or variable to several variables in a single statement. In MORTRAN one writes

```
/ p,q,r...,z / = e                                    (4.1)
```

where p,q,r,...,z are variables and e is an expression. The expression e is evaluated first and then assigned to the list of variables, proceeding from left to right. For example,

```
/ I, A(I,K), J / = SQRT(X/2.0);
```

produces the following FORTRAN statements:

```
I = SQRT(X/2.0)
A(I,K) = I
J = A(I,K)
```

It can be seen that some care must be taken to observe the order of assignments if type conversion is involved, or if (as in the example) the value of one variable affects assignment to another in the list.

B. I-O Abbreviations

Another instance in which the creation of a label can be annoying because it is used only once and contains no mnemonic information is the following:

```
READ (5,:NONSENSE:) i-o list;      :NONSENSE: FORMAT(format list);
```

In MORTRAN one may write

```
                    INPUT i-o list; (format list);     (4.2)
            or      OUTPUT i-o list; (format list);    (4.3)
```

whenever the input or output is to the standard FORTRAN input or output units (5 and 6 respectively). (If your FORTRAN compiler is already "extended" to allow the keywords INPUT and OUTPUT or if these keywords are already used in other contexts, the macros defining these statements may be removed or modified to use other keywords. Similarly, the input and output units may be easily changed to units other than 5 and 6.)

5. USER-DEFINED MACROS

A. String Replacement

Macro definitions are written in the following form:

```
%'pattern'='replacement'                    (5.1)
```

Macro definitions are not statements and therefore need not be terminated by semicolons. (If you put one in it will be ignored.) Macro definitions are "free field" in the sense that you may write more than one definition on one line, or extend one definition to several lines.

The pattern and replacement parts of a macro definition are character strings in the sense described in section 2. Since embedded strings are permitted in macro definitions, the usual rules regarding the doubling of apostrophes apply.

The simplest kind of macro is one which contains neither parameters nor embedded strings. For example, one could write

```
%'ARRAYSIZE'='50'                           (5.2)
```

after which all occurrences of the characters ARRAYSIZE in the program text would be replaced by 50. For example,

```
DIMENSION X(ARRAYSIZE); ... DO J=1,ARRAYSIZE <...>...
```

would produce the same FORTRAN program as if

```
DIMENSION X(50);... DO J=1,50 <...>...
```

had been written.

Blanks are generally not significant when searching for occurrences of the pattern in the program text. For example, the macro

```
%'SIGMA(1)'='SIGMA1'
```

would match the program text

```
SIGMA (1)
```

as well as

```
SIGMA(1)
```

In some cases it is desirable to require that one or more blanks be present in the program text in order that a match occur; this can be done by writing a single blank in the pattern part of the macro.

For example the macro

```
%'DUMP X;'='OUTPUT X; (F10.2);'
```

would match DUMP X;

but not Y = DUMPX;

Normally, the text generated by a macro is itself elegible for replacement by other macros, or even by the the same macro that generated the text. A potential danger to keep in mind in writing macros is the "macro explosion" or infinite macro loop phenomenon.

Suppose that we define a macro like

```
%'EQ'='EQUIVALENCE'                                    (5.4)
```

to provide a convenient abbreviation. Assuming that the first instance of EQ encountered in the program text is not something like A.EQ.B (which would be replaced by A.EQUIVALENCE.B), the first instance of EQ in the program text would be replaced by EQUIVALENCE the way we intended. Then, since the processor "re-scans" all generated text, the EQ in EQUIVALENCE would be re-matched and replaced again resulting in EQUIVALENCEUIVALENCE, and again, resulting in EQUIVALENCEUIVALENCEUIVALENCE, and so on until we run out of space, get an error message, and stop. The simplest way to avoid a macro explosion is to enclose that portion of the replacement part not to be re-scanned in quotation marks ("). We could rewrite (5.4) as

```
%'EQ'='"EQUIVALENCE"'                                  (5.4A)
```

and avoid the macro explosion; if we wrote

```
%'EQU'='"EQUIVALENCE"'                                 (5.4B)
```

we could also avoid matching the relational operator .EQ..

If we wrote %'AB'='CD' and %'CD'='AB' we would cause the processor to go into an infinite loop, not unlike inadvertent loops that are possible in any programming language. Unless you specifically want the replacement to be re-matched, it is a good idea to enclose the replacement part in quotation marks. Thus %'AB'='"CD"' and %'CD'='"AB"' would not cause this problem.

The characters appearing inside quoted strings are not elegible for macro replacement. For example, the statement

```
OUTPUT SETS; (' EQUIVALENCE SETS',10I5);
```

would be unaffected by (5.4).

B. Parameters in Macros

The pattern part of a macro definition may contain up to nine formal (or "dummy") parameters, each of which represents a variable length character string. The parameters are denoted by the symbol #. For example,

```
'EXAMPLE#PATTERN#DEFINITION'                    (5.5)
```

contains two formal parameters. The formal parameters are "positional". That is, the first formal parameter is the first # encountered (reading left to right), the second formal parameter is the second # encountered and so on. The corresponding actual parameters are detected and saved during the matching process. For example, in the string

```
EXAMPLE OF A PATTERN IN A MACRO DEFINITION      (5.6)
```

(assuming (5.5) is the pattern to be matched), the first actual parameter is the string "OF A", and the second actual parameter is the string "IN A MACRO". The parameters are saved in a "holding buffer" until the match is completed.

After a macro is matched, it is "expanded". The expansion process consists of deleting the program text which matched the pattern part of the macro and substituting for it the replacement part of the macro.

The replacement part may contain an arbitrary number of occurrences of formal parameters of the form #i (i=1,2,...,9). During expansion, each formal parameter #i of the replacement part is replaced by the i-th actual parameter. A given formal parameter may appear zero or more times in the replacement part. For example, the pattern part of the macro definition

```
%'PLUS #;' = '#1=#1+1;'                         (5.7)
```

would match the program text

```
PLUS A(I,J,K);                                  (5.8)
```

During the matching process the actual parameter A(I,J,K) is saved in the holding buffer. Upon completion of the matching process (that is when the semicolon in the program text matches the semicolon in the pattern part), the "expansion" of the macro takes place, during which the actual parameter A(I,J,K) replaces all occurrences of the corresponding formal parameter, producing

```
A(I,J,K)=A(I,J,K)+1;                            (5.9)
```

Note that the single formal parameter #1 occurs twice in the replacement part and therefore the single actual parameter A(I,J,K) occurs twice in the resulting string.

The program text which may be substituted for the formal (dummy) parameter is arbitrary except for the following restrictions:

1. It may not be text containing an end-of-statement semicolon. This restriction prevents "run-away" macros from consuming large parts of the program.

2. Parentheses and brackets must be correctly matched (balanced). This facilitates the construction of macros which treat expressions as indivisible units.

3. Quoted character strings are considered to be indivisible units. If the opening apostrophe of a character string is part of the actual parameter, then the entire string must be within the actual parameter.

C. Advanced Uses

It is not necessary to understand this section in order to make effective use of user-defined macros. Much of the available power of the processor could be lost, however, if this section is not clearly understood. First, we define some terms: a string in the program text which matches the pattern part of a macro definition will be called an "instance" of the macro. For example, (5.8) is an instance of (5.7) which expands into (5.9). "No-rescan text" will mean that portion of the replacement part of a macro which has been enclosed in quotation marks. For example, all of the replacement part of (5.4B) is no-rescan text. "Squashed text" will refer to program text in which all multiple blanks have been replaced by a single blank, and from which all comments have been deleted.

The following is a brief description of the matching process: A line (card) is read, squashed, and put into a buffer called the "expand buffer", where beginning with the first character of the buffer it is matched against each macro in turn. When an instance of a macro is found, the macro is expanded, and the matching process begins again with the first character of the newly expanded text. If no instance of any macro is found, then the first character of the expand buffer is sent to an output buffer and the matching process begins again with the next character in the expand buffer. If the first character of the expand buffer is a quotation mark, then all the characters following the quotation mark, up to the next quotation mark are sent to the output buffer. That is, all no-rescan text is sent immediately to the output buffer when it is encountered in the first position of the expand buffer.

A blank in the pattern part of a macro MUST be matched by a blank in the expand buffer for a successful match of the macro. If, on the other hand, the character in the pattern part of a macro is not a blank, a blank in the expand buffer is ignored. This convention allows the programmer to "force" a match to a blank in the incoming text, or to ignore the blank. Since multiple blanks are squashed on input (except in quoted strings), failure to squash the blanks in the pattern and replacement part of macros (which are quoted strings) would mean that patterns containing multiple blanks could not be matched. Therefore, multiple blanks are squashed from the pattern and the replacement parts of macro definitions, except for embedded strings. Stated in terms of the input text: blanks in the input text are ignored unless a "forced match" to a blank is being made, in which case a single blank in the pattern part of the macro effectively matches an arbitrary number of excess blanks in the input text.

Macro definitions are "stacked". That is, the most recently defined macro is the first candidate for matching. (Since the standard macros are read in first, the user's macros are matched before any of the standard set.)

Suppose that, for debugging purposes, we want to "dump" some variable (say X) every time it is assigned a value in the program. That is, every time any assignment is made to the variable X, we want to write on the FORTRAN output file something like

X ASSIGNED xxx

where xxx is the value which is being assigned to X in the program at execution time. We could do this by writing a macro like

```
%';X=#;' = '";X"=#1;  OUTPUT X; ('' X ASSIGNED '',E12.6);'      (5.10)
```

after which every assignment to X would cause X to be dumped. Note that

1. The first character in the pattern part is a semicolon. This was done to prevent matching assignments like SUMX=expression.
2. We have doubled the apostrophes which enclose the Hollerith data in the format-list part of the replacement.
3. We have enclosed the assignment statement in the replacement part in quotation marks to prevent it being re-expanded by the same macro.

While (5.10) is a useful macro, it suffers from the following fault: If we want to dump another variable (say Y) we must write another macro which is identical to (5.10) except that the X's are replaced by Y's. This suggests that we could write a macro which would create macros to dump any variable in the program. This is in fact possible, as follows:

```
%';TRACE#;'='%'';#1=##;''='"';#1"=##1;
          OUTPUT#1;('''' #1 ASSIGNED '''',E12.6);'''              (5.11)
```

Now, if the statement TRACE X; is put in the program, it will cause a macro like (5.10) to be generated. If we write TRACE Y; it will cause another macro to be generated which is like (5.10) except that the X's would be Y's. Careful examination will reveal that the replacement part of (5.11) is in fact the whole macro (5.10) with the following changes

1. All occurrences of X have been replaced by #1.
2. We have doubled the apostrophes which delimit the pattern and replacement parts of (5.10) because they are now inside a quoted string. The double apostrophes in (5.10) have been changed to quadruple apostrophes for the same reason.
3. It was necessary to generate a "#" in the replacement part. To do this we wrote "##" so that it could be distinguished from a formal parameter.

What we have illustrated here is a macro which generates macros. We leave to the reader's imagination the variety of uses to which this facility can be put.

5. CONDITIONAL COMPILATION

It is often the case that several versions of a program are maintained that differ only in certain well-defined respects. One possible technique for merging many versions into a single program is to use MORTRAN macro calls in place of the code that varies, and to define a set of macros that produce the different versions. The macro calls that are not being used for a particular version may be enclosed in comment symbols. The production of a particular version then becomes a matter of removing the comment symbols from a subset of the macro calls, and enclosing another subset in comment symbols. In this way, the subset that is enclosed in comment symbols becomes a form of documentation.

When the number of calls which must be altered in this way is not small, the macro technique becomes cumbersome. For these cases MORTRAN has a "conditional compilation" feature, which allows the generated code to be changed in many places by changing a small number of macro definitions.

The conditional compilation feature is a mechanism whereby a particular piece of program can be ignored (that is, treated as a comment) or not, depending on how a related macro has been defined. The "piece" can be as small as a single character, or as large as an entire program, and will hereafter be called a program "segment". To delimit a segment, the programmer decides on a unique string to begin the segment, and another to end it. Any number of segments may be delimited in this way, and whether or not the segments are compiled will depend on the macros that define the delimiters.

For example, suppose that certain segments are to be generated only in a "multiple-precision" version of a particular program, and other segments only for a "double-precision" version. One might use the strings "/MULTIPLE/" and "/ENDMULT/" to delimit the multiple-precision segments, and the strings "/DOUBLE/" and "/ENDDOUB/" to delimit the double-precision segments. A typical use might be:

```
/MULTIPLE/  DIMENSION A(10);      /ENDMULT/
/DOUBLE/    DOUBLE PRECISION A;   /ENDDOUB/
```

To determine which segments are to be compiled, the strings serving as delimiters should be defined as macros before any of the segments are used. The macros for the start-of-segment strings should have either the keyword "GENERATE" or the keyword "NOGENERATE" as the replacement text, and the macro for the end-of-segment strings should have the keyword "ENDGENERATE" as the replacement text. For example, to specify the multiple-precision version of the program the following definitions should be used:

```
%'/MULTIPLE/'='GENERATE'     %'/DOUBLE/'='NOGENERATE'
%'/ENDMULT/'='ENDGENERATE'   %'/ENDDOUB/'='ENDGENERATE'
```

To generate the double-precision version, the GENERATE and NOGENERATE keywords are interchanged.

Bear in mind that the strings to be used as delimiters will be macro patterns. They should be chosen so as to avoid unintentionally matching program text not involved in conditional compilation. Note that the replacement parts of both /ENDMULT/ and /ENDDOUB/ is the same keyword ENDGENERATE. The same delimiter (unique string) may be used as the terminating delimiter for all conditionally compiled segments. They were chosen to be different in the example in order to improve readability.

Conditionally-compiled segments may be nested within one another. This feature is particularly useful when the conditions are not mutually exclusive; whenever an outer segment is not being generated, all inner segments will also not be generated regardless of how the macros for the inner segments were defined.

APPENDIX A - CONTROL CARDS

Mortran control cards may appear anywhere within the program. Each control card begins with a "%" in column 1, and should not contain embedded blanks or program text.

Column 1
↓

%% Signals end of MORTRAN input. This is the ONLY control card that is required. All others are optional.

%F Switch to FORTRAN mode. (Initial mode is MORTRAN.) While in FORTRAN mode, cards are read and written without any processing. This feature allows the interspersion of FORTRAN and MORTRAN text. If this feature is used, all FORTRAN statement labels should be restricted to four digits (or less) in order to avoid possible conflict with MORTRAN generated statement labels, all of which are five digits long.

%M Switch back to MORTRAN mode. (Initial mode is MORTRAN)

%E Eject (start new page) in MORTRAN listing.

%L List. That is, turn on MORTRAN listing. (Initally ON)

%N Nolist. Turn off MORTRAN listing.

%In Indent n places per nest level in MORTRAN listing
where n = 0,1,2,...,99 (Initally 0)
(Leading blanks are automatically suppressed when n>0 so that "ragged" programs will be "straightened" by MORTRAN.)

%Cn Set input line width to n, where n=10,11,...,80. (Initially n=72.) Characters in columns n+1 thru 80 will appear in the MORTRAN listing, but will be ignored by the processor.

%An Annotation mode switch. Controls the generation of MORTRAN source as comments in the generated FORTRAN as follows:

n=0 Suppress MORTRAN text in FORTRAN file. (Initially n=0.)

n=1 Interleave MORTRAN text as comments in the FORTRAN program starting in column 2 and extending through column 80. If column 80 of the MORTRAN source line is not blank, a second comment line is generated containing the 80th character.

n=2 Interleave the MORTRAN text as comments in the FORTRAN program in columns 40 thru 80. Each MORTRAN source line will appear as two comment lines in the FORTRAN listing, each of which contains one half of the MORTRAN source line.

APPENDIX A - CONTROL CARDS

%Qn Quote switch. Controls MORTRAN comments as follows: (Initially n=0.)
n=0 Comments must be fully enclosed in quotation marks (").
n=1 All comments not closed at the end of each line will be closed by MORTRAN.

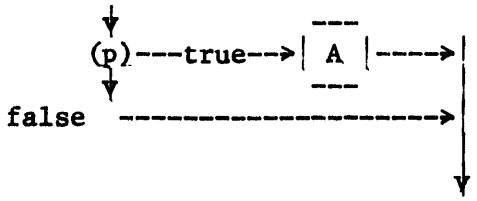
%Un Unit switch. Causes MORTRAN to switch to FORTRAN input unit n for further input, where n=1,2,3...99. When a %% card is read from unit n, input is switched back again to the unit from which the %Un was read. If another %Un is read before a %%, the current input unit is stacked and the MORTRAN input unit is again switched. This control card provides a facility similar to that implemented in other languages by a "COPY" or "INCLUDE" statement. It is particularly convenient for introducing standard declarations, common blocks, or macro definitions from a predefined external file.

APPENDIX B - FLOW CHARTS AND CODE GENERATION

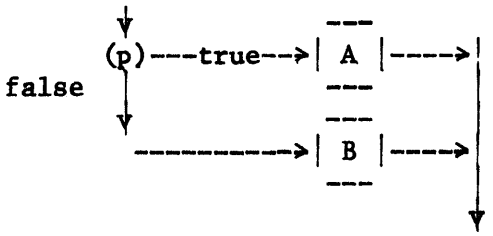
IF, ELSE, and ELSEIF

In the following, let p,q,r...,z stand for arbitrary logical expressions and A,B,C,...,N stand for arbitrary sequences of statements.

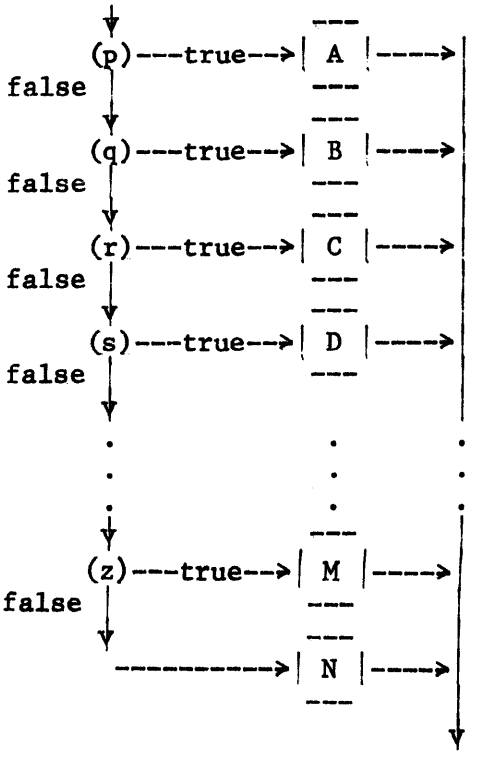
```
IF p < A >
```




```
IF    p     < A >

    ELSE    < B >
```

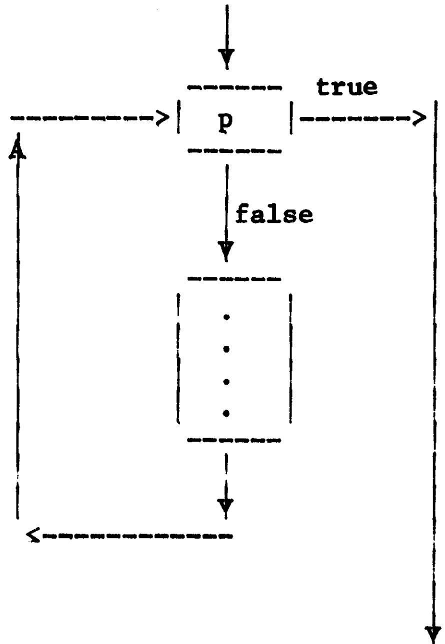



```
IF       p  < A >

ELSEIF   q  < B >

ELSEIF   r  < C >

ELSEIF   s  < D >

   .     .    .
   .     .    .
   .     .    .

ELSEIF   z  < M >

ELSE        < N >
```



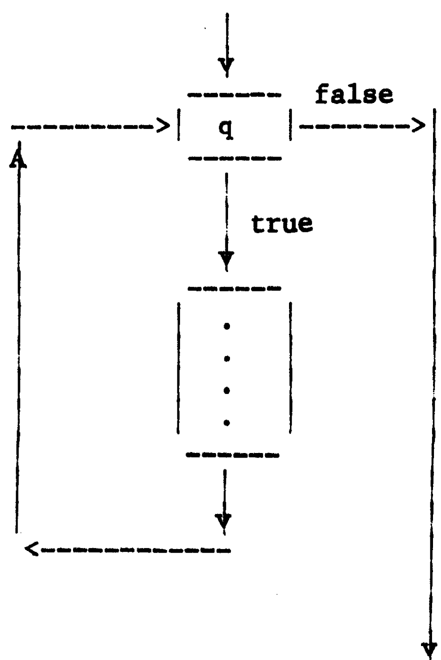
UNTIL p <...>

```
alpha IF(p) GO TO beta
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      GO TO alpha
 beta CONTINUE
```



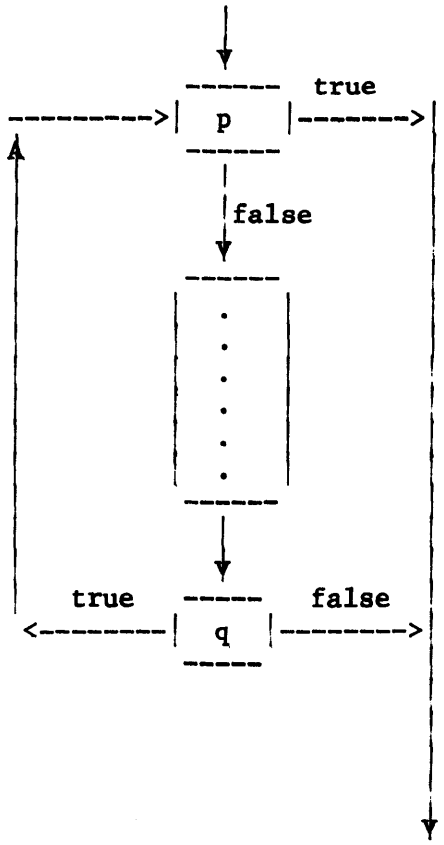
WHILE q <...>

```
alpha IF(.NOT.(q))GO TO beta
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      GO TO alpha
 beta CONTINUE
```



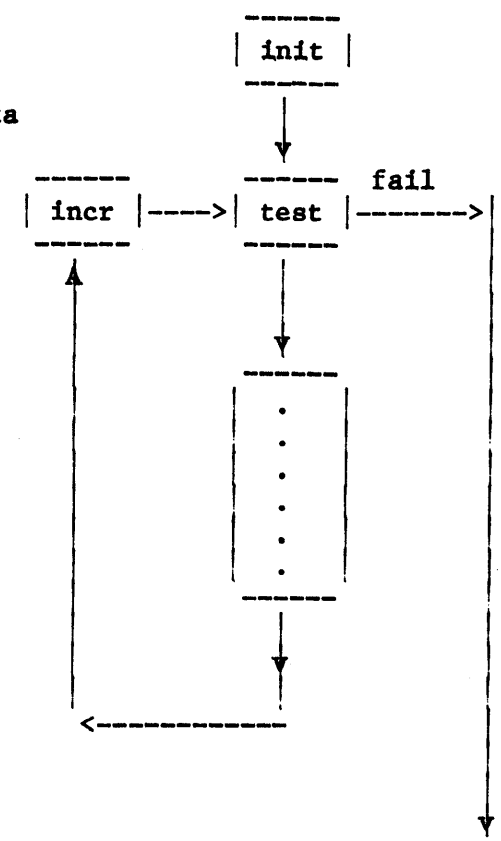
```
UNTIL p <...> WHILE q;
```

```
alpha IF(p) GO TO beta
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      IF(.NOT.(q))GO TO beta
      GO TO alpha
 beta CONTINUE
```



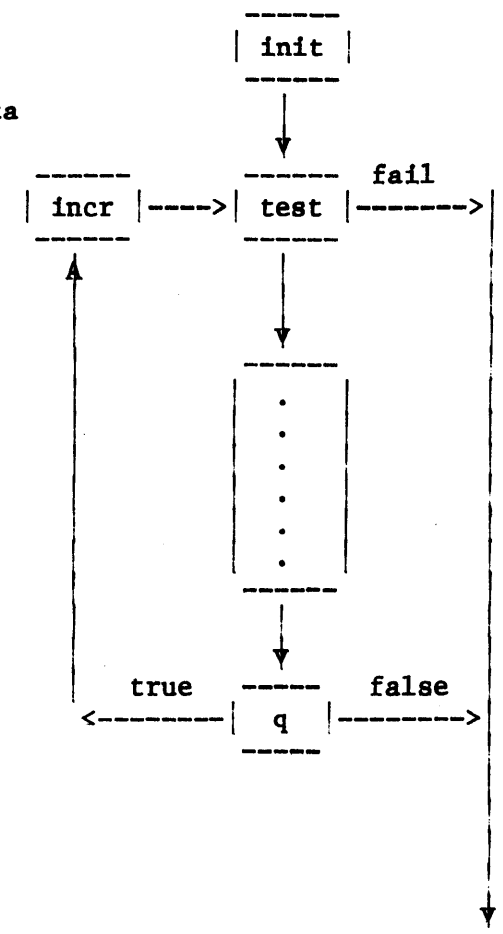
```
FOR v = e TO f BY g <...>
```

```
      v = e
      GO TO gamma
alpha v = v + g
gamma IF(g*(f-v).GT.0)GO TO beta
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      GO TO alpha
 beta CONTINUE
```



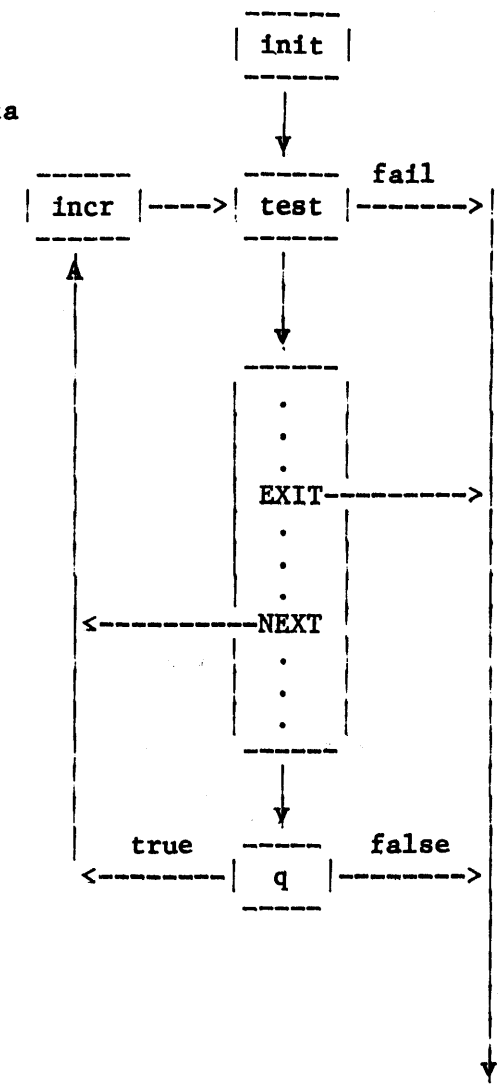
```
FOR v = e TO f BY g <...> WHILE q;
```

```
      v = e
      GO TO gamma
alpha v = v + g
gamma IF(g*(f-v).GT.0)GO TO beta
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      IF(.NOT.(q))GO TO beta
      GO TO alpha
 beta CONTINUE
```



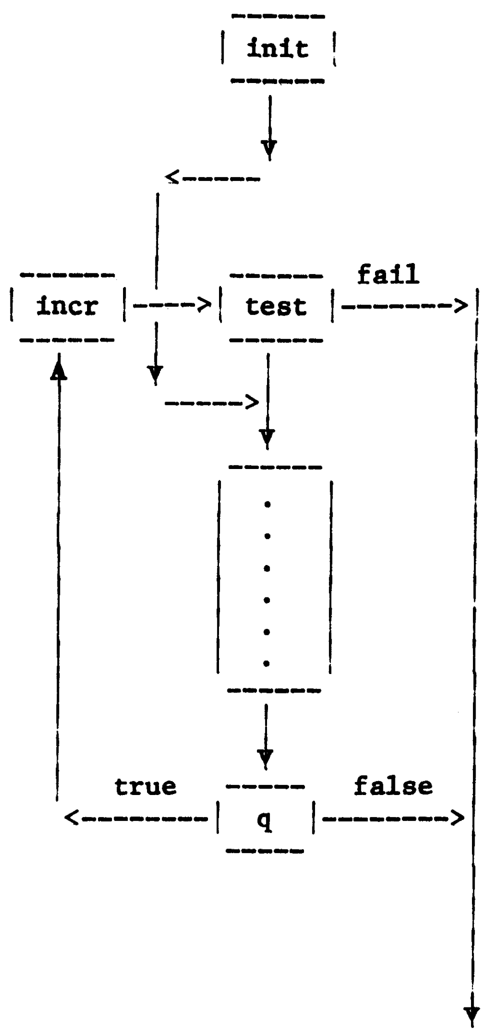
```
FOR v = e TO f BY g <...EXIT...NEXT...> WHILE q;
```

```
      v = e
      GO TO gamma
alpha v = v + g
gamma IF(g*(f-v).GT.0)GO TO beta
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      GO TO beta
      .
      .
      .
      GO TO alpha
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      IF(.NOT.(q))GO TO beta
      GO TO alpha
 beta CONTINUE
```



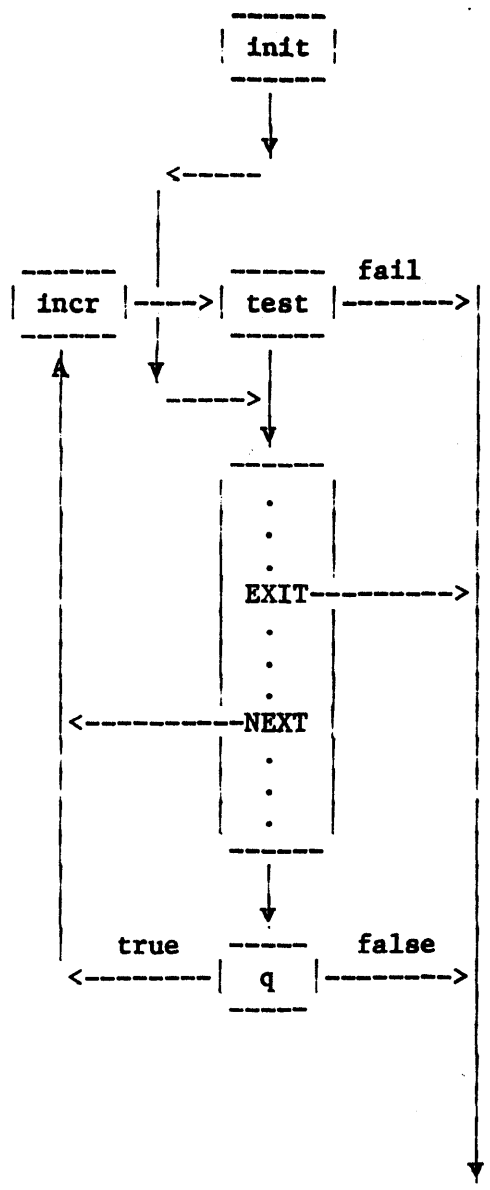
```
DO I=J,K,N <...> WHILE q;
```

```
      DO alpha I=J,K,N
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      IF(.NOT.(q))GO TO beta
alpha CONTINUE
 beta CONTINUE
```



```
DO I=J,K,N <...EXIT...NEXT...> WHILE q;
```

```
      DO alpha I=J,K,N
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      GO TO beta
      .
      .
      .
      GO TO alpha
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
      IF(.NOT.(q))GO TO beta
alpha CONTINUE
 beta CONTINUE
```



APPENDIX C

COMPATIBILITY WITH MORTRAN1

1. Reserved words versus bracketed keywords.

MORTRAN1 users will have noted that keywords (FOR, DO, WHILE, etc.) are not enclosed in brackets in this guide, and that the "reserved word" (i.e. non-bracketed) macro set for MORTRAN2 will not accept bracketed keywords. This does not mean thet programs written using the bracketed keywords of MORTRAN1 must be re-written. A set of macros is available that implements the old form. If the bracketed keywords are preferred, the corresponding set of macros should be used; otherwise, the program should be converted to the new form.

2. Line length.

In order to change the input line length, MORTRAN1 used a %C followed by an X in the last column to be processed. This method was found to be error-prone and awkward. The form of this control card in MORTRAN2 is %Cn where n=10,11,...,80.

3. Macro Definitions.

A. "No-rescan" macros.

The way a "no-rescan" macro is defined has been changed to a new, more flexible format (see section 5.A).

B. More than one macro definition on a line.

EVERY macro definition is now preceded by a percent-sign (%). If you have several definitions on a line, simply change the commas which separate the definitions to percent-signs.

4. Blanks.

MORTRAN1 deleted ALL blanks from the program text. MORTRAN2 changes multiple blanks (sequences of 2 or more blanks) to a single blank. This fact is important in macro definitions (see section 5.A) and in labels. Just remember that:

MULTIPLE BLANKS ARE CHANGED TO A SINGLE BLANK

For example, :THIS LABEL: matches :THIS LABEL: but not :THISLABEL:.

5. Switching between FORTRAN and MORTRAN.

Now %F switches to FORTRAN and %M switches back to MORTRAN. (See Appendix A.)

APPENDIX D

Converting old FORTRAN programs to MORTRAN

Old FORTRAN programs may be mechanically converted to MORTRAN by the following steps:

1. Terminate all statements with a semicolon.

2. Delete any continuation marks.

3. Convert any Hollerith field in the program by enclosing the field in apostrophes and deleting the nH field descriptor that precedes the field. This is necessary because of the possibility that the field may contain sequences of two or more blanks which would be reduced to a single blank by the MORTRAN processor.

4. Enclose comment lines in quotation marks. (Another possibility is to include a %Q1 control card (See appendix A) and then change all occurrences of C in column 1 to a quotation mark.

5. Terminate the program with a %% control card. This is necessary because the ANSI FORTRAN standard makes no provision for sensing an end-of-file condition.

Obviously, such a mechanical conversion will add nothing to the structure or mnemonic content of a program. Readability and mnemonic content must be provided by the programmer.

APPENDIX E

MORTRAN ERROR MESSAGES

The messages are arranged in order of the expected frequency of occurrence. The explanations for each error are arranged in order of the most probable cause.

***** MISSING RIGHT BRACKET
This message is issued at the end of the MORTRAN listing. Check the nesting level on the MORTRAN listing to locate the error.

***** UNCLOSED LOOP OR BLOCK
This message is similar to the above, except that it was possible to be more explicit.

***** UNCLOSED STRING
The program ended without the closing apostrophe for a character string.

***** UNCLOSED COMMENT
The program ended whithout the colsing quotation mark for a comment.

***** MACRO BUFFER OVERFLOW
Missing right colon on alpha label.
Too many labels, or excessively long labels.
Too many user-defined macros for available space.

***** INPUT BUFFER OVERFLOW
Unclosed character string. Check the column to the left of the nesting level to locate the beginning of the string.
Statement too long.

***** LABEL STACK UNDRFLOW
An excess of right brackets or a missing left bracket. This message will usually be accompanied by a negative nesting level in the MORTRAN listing. Unlike the MISSING RIGHT BRACKET message, this message is issued when the extra right bracket is encountered.

***** MACRO STACK UNDRFLOW
An error in a user-defined macro.

***** EXPANSION BUFFER OVERFLOW
Macro explosion. (see section 5.A)
User defined macro error.
User defined macro(s) expansion too large for buffer.

***** MACRO STACK OVERFLOW
User defined macro error
Nesting level exceeds 50.

***** LABEL STACK OVERFLOW
User defined macro error
Nesting level exceeds 50.

MORTRAN ERROR MESSAGES (cont'd)

***** BAD NUMERICS ON CONTROL CARD
A control card which required a number had invalid digits or a number that was outside the allowable range for the control card.

***** BAD PARAMETER NUMBER
The character following a # in the replacement part of a macro definition is not a digit from 1 to N, where N is the number of formal parameters in the pattern part of the macro.

***** ILLEGAL MACRO PATTERN
The pattern part of a macro definition may not be a null string.

***** PARAMETER BUFFER OVERFLOW
The sum of the lengths of the actual parameters exceeds the space available for them.

```
 MORTRAN 2.0      (VERSION OF 6/24/75)

      %U5
   0
   0    "  TEST PROGRAM USING RESERVED KEYWORDS "
   0
   0     % 'ARRAYSIZE' = '100'
   0     % 'PRIMESIZE' = '3000'
   0
   0    INTEGER SIEVE1;
   0    INTEGER PRIMES(PRIMESIZE);
   0    INTEGER COUNT;
   0
   0    COUNT=SIEVE1(10000, PRIMES);
   0    OUTPUT COUNT;(I8,' PRIMES LESS THAN 10,000'//);
   0    OUTPUT (PRIMES(I),I=1,COUNT);(1X,10I10);
   0    STOP;
   0    END;
   0
   0    INTEGER FUNCTION SIEVE1(MAX, PRIMES);
   0
   0        "  GENERATE PRIMES
"  0
"  0           USES ALGORITHM 310(A1), PRIME GENERATOR 1,
"  0           B. A. CHARTRES, CACM 10(SEPT 1967), 569
"  0        "
   0
   0    INTEGER MAX; INTEGER PRIMES(PRIMESIZE);
   0    INTEGER Q(ARRAYSIZE),DQ(ARRAYSIZE);
   0    INTEGER I,SVSZ,PRMCNT,N;
   0    LOGICAL PRIME;
   0
   0        "  INITIALIZE "
   0
   0    /PRIMES(1),SVSZ,PRMCNT/=2; PRIMES(2)=3; Q(2)=9; DQ(2)=6;
   0
   0        "  NOW LOOK FOR PRIMES "
   0
   0    :PRIME TEST:
   0    FOR N=5 TO MAX BY 2 <
   1        PRIME=.TRUE.;
   1        FOR I=2 TO SVSZ <           " RUN THE NUMBER N THROUGH THE SIEVE Q "
   2            IF N .EQ. Q(I) <        " CAUGHT BY THE SIEVE? "
   3                 PRIME=.FALSE.;
   3                 Q(I)=N+DQ(I);
   3                 IF I .EQ. SVSZ <
   4                       SVSZ=SVSZ+1; Q(SVSZ)=PRIMES(SVSZ)**2;
   4                       DQ(SVSZ)=2*PRIMES(SVSZ);
   4                       NEXT :PRIME TEST:;
   4                       >
   3                 >
   2            >
   1        IF (PRIME) < PRMCNT=PRMCNT+1; PRIMES(PRMCNT)=N; >
   1        >
   0    SIEVE1=PRMCNT;
   0    RETURN; END;
      %%
      %%
 0 MORTRAN ERRORS ENCOUNTERED
```