Hortran3 User's Guide

A. James Cook

Computation Research Group 88
Stanford Linear Accelerator Center
Stanford California, 94305
December 16, 1982

Working Paper

Do not quote, cite, abstract, or reproduce without prior permission of the author(s).

DISCLAIMER NOTICE

This document and associated documents and programs, and the material and data contained therein, were developed under the sponsorship of the U.S. government. Neither the U.S. nor the U.S.D.O.E., nor the Leland Stanford Junior University, nor their employees, nor their respective contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any liability or responsibility for accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represents that its use will not infringe privately-owned rights. Mention of any product, its manufacturer, or suppliers shall not, nor is it intended to, imply approval, disapproval, or fitness for any particular use. A royalty-free, non-exclusive right to use and disseminate same for any purpose whatsoever is expressly reserved to the U.S. and the University.
CONTENTS

1. INTRODUCTION

PART I. The Programming Language

Features
Directives

2. PROGRAMMING CONVENTIONS

Form, Line Length, Strings, Comments
Labels, Blanks, Summary of Conventions

3. STRUCTURE

Statements
Blocks
Control
Sequential control
IF
UNLESS
ELSE
ELSEIF
FORTRAN IF
Summary of IF, ELSE, and ELSEIF
Iterative Control
WHILE
LOOP
UNTIL
FOR
DO
The "compact" DO
"Forever" loops
EXIT
NEXT
Multi-level Control
Multi-level EXIT
Multi-level NEXT
EXECUTE local procedure

A.James Cook

December 16, 1982
4. CONTROL SUMMARY

- Summary of Iterative Control .................................................. 25
- Summary of Sequential Control .............................................. 26
- GOTO :label: ............................................................................. 26

5. MISCELLANEOUS FEATURES

- Multiple Assignment ..................................................................... 27
- Input-Output Abbreviations ....................................................... 28

PART II. The Definition Language ............................................... 28

6. INTRODUCTION TO NOTATIONAL TRANSFORMATIONS ....................... 29

- A useful example ......................................................................... 29

7. TRANSFORMATION RULES .......................................................... 32

- The Simplest Example ............................................................... 32
- The Template Part ....................................................................... 33
  - Parameter Markers .................................................................. 33
  - Specifiers ................................................................................ 35
    - Forward references in Specifications .................................... 37
    - Specifying "Optional" ............................................................ 37
- The Value Part ............................................................................. 38
  - Parameter References ............................................................. 38
  - Left-to-right Substitution ....................................................... 39
- NULL Parameters ......................................................................... 40
- Primitive Evaluation Functions ............................................... 41
  - Order of evaluation .................................................................. 41
  - The REDUCE primitive ........................................................... 42
    - REDUCED text vs. REPLACED text .................................... 42
  - The COMPUTE primitive .......................................................... 43
    - Constant expressions ........................................................... 43
    - Constant Logical Expressions .............................................. 44
  - Evaluation-time IF-ELSE .......................................................... 45
  - String Expressions .................................................................... 46
    - The primitives NULL and EXIST ......................................... 47
- Recursive Application of Rules .................................................. 48
  - Unwanted Recursion ................................................................ 50
- The Scope of Rules ..................................................................... 51
- Removing Rules and Specifiers .................................................. 52
- Internal Counters ....................................................................... 53
- The LENGTH Primitive ............................................................. 54
  - Appending text ......................................................................... 56
- Special Symbols ......................................................................... 58
- The Reserved Symbol .................................................................. 58
- Other "special" symbols ............................................................. 58
- Terminators and Delimiters ....................................................... 59
  - The Apostrophe ....................................................................... 59
# Mortran User's Guide

## Appendix

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A. PROCESSOR CONTROL</td>
<td></td>
</tr>
<tr>
<td>Free-form directives</td>
<td>62</td>
</tr>
<tr>
<td>Column-one-restricted directives</td>
<td>64</td>
</tr>
<tr>
<td>B. PRIMITIVE SPECIFICATIONS</td>
<td>65</td>
</tr>
<tr>
<td>Qualifiers</td>
<td>65</td>
</tr>
<tr>
<td>Quantifiers</td>
<td>66</td>
</tr>
<tr>
<td>Alternatives</td>
<td>67</td>
</tr>
<tr>
<td>Literals</td>
<td>68</td>
</tr>
<tr>
<td>C. PRIMITIVE EVALUATORS</td>
<td>69</td>
</tr>
<tr>
<td>REDUCE</td>
<td>69</td>
</tr>
<tr>
<td>COMPUTE</td>
<td>69</td>
</tr>
<tr>
<td>Counter primitives</td>
<td>70</td>
</tr>
<tr>
<td>Parameter Primitives</td>
<td>70</td>
</tr>
<tr>
<td>Miscellaneous primitives</td>
<td>71</td>
</tr>
</tbody>
</table>

---

A. James Cook

December 16, 1982
1. INTRODUCTION

This Guide is divided into two parts:

- Part I -- The Programming Language.
- Part II -- The Definition Language.

The programming language (more accurately, the notation) is defined by set of transformation rules. Transformation of Mortran3 notation into FORTRAN notation results from the application of these rules by the Mortran3 processor.

The definition language is used to define new programming notations. Briefly, the user supplies a prototype of a new notation, and a prototype of the FORTRAN notation into which his notation is to be transformed. The details of how this is done is the subject of Part II.

"Language" versus "Notation"

We are compelled by usage to describe "languages", even though that term (as it is applied in computer-related contexts) has been abused to the point of becoming virtually meaningless.1 Current use of the term "high-level language" usually implies the existence of a program called a compiler. The compilers referred to in this document are FORTRAN compilers; they transform FORTRAN notations into "machine-dependent" binary notations.

The binary notations into which a FORTRAN program is transformed (translated) depend upon the computer that will "execute" the program. Consequently, a different FORTRAN compiler is required for each different computer on which FORTRAN programs will be "run".

We regard FORTRAN compilers as hardware-extensions. From this point of view, the portability of FORTRAN programs derives from the fact that virtually all "maxi" and "mini" computers, and indeed, many "micro" computers have been extended in this sense: they have FORTRAN compilers.

The Processor

The Mortran3 processor exists in two forms:

- Form I: A FORTRAN program of less than 2000 lines, and
- Form II: A Mortran program of less than 600 lines.

1 Examples: Machine-language (meaning either binary notation or assembler notation), Machine-oriented-language (meaning a notation for writing compilers), High-level language, Procedural language, Non-Procedural language, Structured language, et cetera, ad absurdum.
When Form I is compiled it produces a processor. This processor can, given the necessary rules, transform Form II into Form I. Given other rules, the processor can transform Form II into notations other than FORTRAN.

Precompilation

The Mortran3 processor (as described here) is a precompiler. A precompiler's output is a compiler's input.

A program P in Mortran3 notation

The Mortran3 processor

The program P transformed into a program in FORTRAN notation

A FORTRAN compiler

Portability

The portability of a FORTRAN program depends largely upon the extent to which the program conforms to the (1966 ANSI) FORTRAN standard.

The Mortran3 "language" does not produce FORTRAN statements that violate the 1966 ANSI FORTRAN standard. Consequently, Mortran3 programs may be made portable to the same extent that FORTRAN programs may be made portable.

---

2 The Mortran3 processor may be used for other purposes. See the Mortran3 Reference Manual.

3 The precision with which arithmetic operations are performed is not dealt with here.

4 FORTRAN 77, at this time, is not in widespread use. Pascal seems a more likely candidate for the "portable language" for the near future.

A.James Cook

December 16, 1982
PART I. The Programming Language

FEATURES

Free-form
• Column and line (card) boundaries may be ignored.
• Continuation marks are not required.

Labels
• Alphanumeric labels of arbitrary length are accepted.
• FORTRAN statement labels (numbers) are accepted.

Local Procedures.

Nested block structure
Controlled execution of blocks:
• Sequential control
  IF, ELSE, ELSEIF, UNLESS, and GOTO.
• Iterative control (Loops)
  WHILE, UNTIL, FOR, LOOP, and DO.
  EXIT (jump out of) any loop
  NEXT (go to next iteration of) any loop
  Tests for loop termination may be made at the
  beginning of a loop
  end of a loop
  both beginning and end of a loop
  none of the above (i.e. "Forever" loops)
• Multi-level control statements.
  Multi-level iterative control.
    EXIT (alphanumeric label)
    NEXT (alphanumeric label).
  Multi-level sequential control.
    EXECUTE Local Procedure.
    GOTO alphanumeric label.

Comments may be inserted anywhere in the program text.

Operators
• Relational operators may be denoted by: <, <=, =, ~=, =>, and >.
  The operators <= and => may also be denoted by =< and >= respectively.
  The symbols ≤, ≥, and ≈ may be used if the user's display device
  is capable of representing them.

• Logical operators may be denoted by: & (and), | (or), and ¬ (not).

Conditional (alternate) compilation

Abbreviations
Multiple assignment statements
Input/output statements may be abbreviated.
DIRECTIVES

Directives do not involve transformation of notations, rather they are instructions to the processor.

Listing Directives

- Automatic printing of the nesting level.
- Automatic indentation according to nesting level (optional).
- Interleaving of the Mortran3 Source program as COMMENTS in the generated FORTRAN program (optional).
- Inclusion of Mortran3 COMMENTS as COMMENTS in the generated FORTRAN program (optional).

Other Directives

- FORTRAN source text may be interspersed with Mortran3 source text.
- Mortran source input-unit switching.
- Mortran source input-unit rewinding.
- Mortran input line-length may be set by the programmer.
- The number of columns of the input line that are interpreted as program text may be set by the programmer.
- Automatic termination of Mortran comments at the end of an input line.
- Tracing of the transformation process.
2. PROGRAMMING CONVENTIONS

Form

Statements may begin anywhere on a line, and may end anywhere on the same line or on a succeeding line.

- Continuation marks are not required (nor permitted).
- The end of a statement is determined by a semicolon (;).

This convention (with the convention for blanks described below) permits free-form programming in which column and line boundaries may be ignored.

Line Length

The line length $L$ may be set by the programmer: $20 \leq L \leq 120$. (Default: 80.)

The number $C$ of columns to be interpreted as program text may be set by the programmer: $10 \leq C \leq L$. (Default: 72.)

Strings

A string is a sequence of characters enclosed in apostrophes (as in 'TWINE'). An apostrophe within a string is denoted by a pair of apostrophes (as in '''Twas brillig, and the slithy toves... '').

On output, Mortran3 will convert strings to the 1966 FORTRAN standard nH format if directed to do so by the programmer. (See appendix A).

- Example: 'DON''T' will be converted to 5HDON'T

Comments

Comments are enclosed in quotation marks (as in "BALDERDASH!"). A comment may contain any character except a quotation mark.

Comments may be inserted anywhere in the program text except:
- Within a string a comment becomes part of the string.
- In transformation rules. (See Part II.)

Example: CALL SUB (A "INPUT", B "OUTPUT"); Is equivalent to: CALL SUB (A, B);
**Labels**

An alphanumeric label is a sequence of characters enclosed in colons (as in :CONSERTATIVE:).

- The characters may be any combination of letters and digits.
- The length is arbitrary.

:`THIS ABSURDLY LONG ALPHANUMERIC LABEL IS A VALID LABEL:`

- Alphanumeric labels are intended to supplant FORTRAN statement labels, but FORTRAN statement labels (numbers) are allowed in most contexts.

**Blanks**

A sequence of two or more blanks is equivalent to a single blank except in strings, where all blanks are preserved.

Blanks are significant in some Mortran3 contexts.

- :THIS LABEL: is equivalent to :THIS LABEL:
- :THIS LABEL: is not equivalent to :THISLABEL:

- DO/I=.N is acceptable in FORTRAN.
  In contrast, keywords must be followed by at least one blank in Mortran3.

**Summary of conventions**

- Terminate statements with a semicolon (;).
- Enclose comments in quotation marks (").
- Enclose labels in colons (:).
- Enclose character strings in apostrophes (').
- An arbitrary number of blanks may be substituted for a single blank except in strings.
3. STRUCTURE

3.1 STATEMENTS

A valid FORTRAN statement becomes a valid Mortran3 statement when:

- The statement is terminated by a semicolon,
- Continuation marks (if any) are deleted, and
- Hollerith fields are converted to strings.  

Subject to (3-1), FORTRAN may be regarded as a subset of Mortran3; Conversely, Mortran3 may be regarded as an extension of FORTRAN. Most of the "extensions" are related to block structure and to the control of the execution of blocks.

3.2 BLOCKS

A Mortran3 block is a sequence of Mortran3 statements enclosed in the special characters [ and ]. Let \( S_i \) \((i=1,n)\) be a sequence of Mortran3 statements:

\[
(3-2) \quad S_1; S_2; S_3; \ldots; S_n;
\]

The sequence (3-2) becomes a block when it is enclosed in brackets:

\[
(3-3) \quad [ \quad S_1; S_2; S_3; \ldots; S_n; \quad ]
\]

Blocks may be nested. That is, a block may contain one or more blocks. For example, one could insert the block [ \( T_i \) ] \((i=1,m)\) in (3-3) producing:

\[
[ S_1; S_2; \quad [ \quad T_1; T_2; T_3; \ldots; T_m; \quad ]; \quad S_3; \ldots; S_n; \quad ]
\]

The block [ \( T_i \) ] is completely contained or nested within the block [ \( S_i \) ].

Example of a block: [ X=Y; CALL WUNDER(BAR); B=E; ]

\[\text{For example, } 15\text{THIS THING is converted to 'THIS THING'.}\]
3.3 **CONTROL**

- An ellipsis enclosed in brackets [...] will denote an arbitrary block.
- A control phrase controls the execution of a block.
- A control clause is a control phrase followed by the block it controls.
- When referring to blocks:
  - executed means: control is transferred to the first statement within the block.
  - skipped means: control is transferred to the first statement following the block.

3.3.1 **Sequential control**

The simplest sequential control clause is the IF-clause.

3.3.1.1 **IF**

```
IF e [...]  
```

where e is an arbitrary logical expression
- If e is TRUE, the block is executed.
- If e is FALSE, the block is skipped.

<table>
<thead>
<tr>
<th>Example (J-5)</th>
<th>IF A=B &amp; C&gt;D [ C=D; E=F; ] G=H;</th>
</tr>
</thead>
<tbody>
<tr>
<td>FORTRAN</td>
<td>IF(A.NE.B .OR. C.LE.D)GO TO L1</td>
</tr>
<tr>
<td>generated</td>
<td>C=D</td>
</tr>
<tr>
<td>by MortranJ</td>
<td>E=F</td>
</tr>
<tr>
<td>from (J-5)</td>
<td>L1 CONTINUE</td>
</tr>
<tr>
<td></td>
<td>G=H</td>
</tr>
</tbody>
</table>

The label **L1** is a FORTRAN statement label (integer constant) generated by Mortran. One or more such labels is generated for each control phrase.
Notice that DeMorgan's laws have been applied in (3-5):

<table>
<thead>
<tr>
<th>The expression</th>
<th>-(A=B &amp; C&gt;D)</th>
</tr>
</thead>
<tbody>
<tr>
<td>becomes</td>
<td>A.NE.B..OR.C.LE.D</td>
</tr>
</tbody>
</table>

DeMorgan's laws will be applied to expressions with FORTRAN operators (.LT., .LE., .EQ., etc.) as well.

3.3.1.2 UNLESS

UNLESS e [...]

- If e is FALSE, the block is executed.
- If e is TRUE, the block is skipped.

3.3.1.3 ELSE

An IF-clause (or an UNLESS-clause) may be followed by an ELSE-clause:

IF e [...] ELSE [...]  

- If e is TRUE, the IF-block is executed and the ELSE-block is skipped.
- If e is FALSE, the IF-block is skipped and the ELSE-block is executed.

Example (3-7):

| IF A < B [ C=D; E=F; ] ELSE [ G=H; I=J; ] K=L | IF(A.GE.B)GO TO L1  
| C=D | E=F  
| GO TO L2  
| L1 CONTINUE  
| G=H | I=J  
| L2 CONTINUE  
| K=L |
3.3.1.4 ELSEIF

IF/ELSE-clauses may be nested to a depth of 50. But nesting IF/ELSEs is not always desirable; in some situations, one wants to avoid nesting. For example, consider the following "case analysis" problem:

\[
\text{(J-8)} \quad \text{IF} \; p \; \text{[A]} \; \text{ELSE} \; \text{[IF} \; q \; \text{[B]} \; \text{ELSE} \; \text{[IF} \; r \; \text{[C]} \; \text{ELSE} \; \text{[D]} \; \text{] } \; \text{]} \; \text{] s;}
\]

where \( p, q, \) and \( r \) are logical expressions
A,B,C, and D are blocks of statements
\( s \) denotes the first statement following the nest.

- If \( p \) is TRUE, block A is executed and control is transferred to \( s \).
- If \( p \) is FALSE and \( q \) is TRUE, block B is executed and control is transferred to \( s \).
- If \( p \) and \( q \) are FALSE and \( r \) is TRUE, block C is executed and control is transferred to \( s \).
- If \( p, q, \) and \( r \) are all FALSE, block D is executed, and control is transferred to \( s \).

Using "structured programming" principles to expose the level of nesting, we could re-write (J-8) as:

\[
\text{(J-9)} \quad \text{IF} \; p \; \text{[A]} \; \text{ELSE} \; \text{[IF} \; q \; \text{[B]} \; \text{ELSE} \; \text{[IF} \; r \; \text{[C]} \; \text{ELSE} \; \text{[D]} \; \text{] } \; \text{] ] s;}
\]

But (J-9) is awkward because each ELSE-clause increases the level of nesting. The ELSEIF-clause combines an ELSE-clause with an IF-clause:

\[
\text{IF} \; p \; \text{[A]} \; \text{ELSEIF} \; q \; \text{[B]} \; \text{ELSEIF} \; r \; \text{[C]} \; \text{ELSE} \; \text{[D]} \; \text{s;}
\]

Thus, the tests are made at the same nest level.

An arbitrary number of ELSEIF-clauses may follow an IF-clause.
In addition to the advantages mentioned above, the ELSEIF generates "cleaner" FORTRAN:

```
IF p [A] ELSEIF q [B] ELSEIF r[C] ELSE D s;
```

```
IF(-(p)) GO TO L1
A
GO TO L4
L1 IF(-(q))GO TO L2
B
GO TO L4
L2 IF(-(r))GO TO L3
C
GO TO L4
L3 D
L4 CONTINUE
```

The nested IF/ELSE example generates more "CONTINUE" statements:

```
IF p [A] ELSE IF q [B] ELSE IF r[C] ELSE D s;
```

```
IF(-(p)) GO TO L1
A
GO TO L6
L1 IF(-(q))GO TO L2
B
GO TO L5
L2 IF(-(r))GO TO L3
C
GO TO L4
L3 D
L4 CONTINUE
L5 CONTINUE
L6 CONTINUE
```
3.3.1.5 FORTRAN IF

The FORTRAN logical IF: is equivalent to:

\[
\text{IF (e) \quad \text{IF e [statement;]}}
\]

The IF-phrase is the preferred form because an ELSE-clause (or an ELSEIF-clause) may not follow a FORTRAN IF.

incorrect: IF(X=Y) CALL SOME(HELP); ELSE [ ...]
correct: IF X=Y [CALL SOME(HELP);] ELSE [ ...]

3.3.1.6 Summary of IF, ELSE, and ELSEIF

* An IF-clause may appear by itself.
* An IF-clause may be followed by an ELSE-clause.
* An IF-clause may be followed by an arbitrary number of ELSEIF-clauses.
* The last in a sequence of ELSEIF-clauses may be followed by an ELSE-clause
3.3.2 Iterative Control

A Mortran3 loop is a block which is preceded by and, optionally, followed by a loop-control phrase. (We will shorten this to "phrase" in most contexts.)

* A loop is terminated means: control is transferred to the first statement following the loop.

3.3.2.1 WHILE

```
WHILE e [...]
WHILE e [...] REPEAT
```

* REPEAT is optional.
* IF e is FALSE, the loop is terminated.
* IF e is TRUE (1 the block is executed and
  2 control is returned to test e again.

Example (3-11)
```
WHILE A<B [...]
L1 IF(A.GE.B) GO TO L2
... GO TO L1
L2 CONTINUE
```

3.3.2.2 LOOP

If one wanted to test at the end of a WHILE-loop instead of the beginning, one could write

```
(3-12) LOOP [...] WHILE e;
```

In (3-12) the block is always executed once. Then, while e is TRUE, the block is executed repeatedly. When e becomes FALSE the loop is terminated.
3.3.2.3 UNTIL

UNTIL e [...] 

- IF e is TRUE, the loop is terminated.
- If e is FALSE (1 the block is executed and
  (2 control is returned to test e again.

Example (J-14)  UNTIL X>Y [...] 

L1 IF(X.GT.Y)GO TO L2 
  ... 
  GO TO L1 
L2 CONTINUE

Tests for loop termination may be made at both ends of a loop. For example, if e1 and e2 are logical expressions, the following test at both the beginning and the end:

WHILE e1 [...] UNTIL e2 ; 
WHILE e1 [...] WHILE e2 ; 
UNTIL e1 [...] WHILE e2 ; 
UNTIL e1 [...] UNTIL e2 ;

The FORTRAN generated for the first example is:

WHILE e1 [...] UNTIL e2;

L1 IF(-(e1))GO TO L2 
  ... 
  IF(e2)GO TO L2 
  GO TO L1 
L2 CONTINUE

Any Mortran3 loop may be tested at both ends, including the ones we haven't described yet: three flavors of FOR loops, two flavors of DO loops and the "Forever" loop.
3.3.2.4 FOR

The loop control phrases discussed up to this point do not involve control variables, that is, variables that are automatically incremented (decremented) for each execution of the loop. The FOR loop involves a control variable.

```
 FOR V=Ax BY Ai TO Af [...]
```

- V is the control variable.
  - V may be a subscripted variable.
  - V must be of type REAL or INTEGER.

- Ax, Ai, and Af are arbitrary arithmetic expressions.
  - Ax is the initial value
  - Ai is the increment
  - Af is the final value

The values of Ax, Ai, and Af may be negative.

Ax is assigned to V and the test for loop termination is made:

```
 The test: (J-15) Ai*(V-Af)>0
```

- If (J-15) is TRUE, the loop is terminated.
- If (J-15) is FALSE:
  1. the block is executed
  2. V is incremented by the value of Ai
  3. control is passed to the test.
- Multiplication by Ai corrects the sign of (V-Af).
Example (J-16)

\[
\begin{align*}
\text{FOR } X & = 0.4 \text { TO } -0.3 \text { BY } -0.01 \left[ \ldots \right] \\
X & = 0.4 \\
\text{GO TO } L3 \\
L1 & X = X + (-0.01) \\
L3 & \text{IF } ((-0.01)(X-(-3)) > 0) \text{GO TO } L2 \\
\ldots & \\
\text{GO TO } L1 \\
L2 & \text{CONTINUE}
\end{align*}
\]

This trivial example is presented to make the following points:

- The increment in (J-16) is negative
- The control variable X is assigned both zero and negative values.
- The loop is terminated only when \( X \leq -0.3 \). In the statements indicated by the ellipsis in (J-16), the programmer may change \( V \) or any of the variables of \( Ax, Ai, \) or \( Af \).

1) Changes in the variables of \( Ax \) will have no effect on the loop, since the assignment \( V = Ax \) is not re-executed.

2) Changing the value of the variables of \( Ai \) or \( Af \) might be useful for asymptotic processes.

Assuming a "straightforward" application of the FOR-loop: If \( Ai \) and/or \( Af \) are expressions whose evaluation is time-consuming, (and time is important) assignment of \( Ai \) and \( Af \) to temporary variables outside the loop may be indicated.

The FOR-loop has three forms:

\[
\begin{align*}
\text{FOR } V & = Ax \text { BY } Ai \text { TO } Af \left[ \ldots \right] \\
\text{FOR } V & = Ax \text { TO } Af \text { BY } Ai \left[ \ldots \right] \\
\text{FOR } V & = Ax \text { TO } Af \left[ \ldots \right]
\end{align*}
\]

where \( V, Ax, Ai, \) and \( Af \) are as above

- If no increment (BY \( Ai \)) is given the integer 1 is assumed.
3.3.2.5 DO

(3-17) \[
\text{DO } i=j,k,n \ldots 
\]

where \(i, j, k\), and \(n\) must conform to FORTRAN requirements for \(DO\) loops.

A standard FORTRAN \(DO\) loop is generated from (3-17). Therefore, \(i, j, k\), and \(n\) must all be of type INTEGER and may not be subscripted variables or expressions.

3.3.2.6 The "compact" \(DO\)

There is one exception to the rule that loops must be preceded by control phrases.

(3-18) \[
[ i=j,k,n; \ldots ]
\]

(3-18) generates a standard FORTRAN \(DO\) loop. This form permits compact notation for nests like the following example.

Example (3-19) \[
[I=1,H1; [J=1,H2; [K=1,H3; \ldots]]]
\]

\[
\begin{align*}
\text{DO } 10 & \text{ I=1,H1} \\
\text{DO } 20 & \text{ J=1,H2} \\
\text{DO } 30 & \text{ K=1,H3} \\
\ldots & \\
30 & \text{ CONTINUE} \\
20 & \text{ CONTINUE} \\
10 & \text{ CONTINUE}
\end{align*}
\]

This "oddball" notation has been severely criticized on the ground that it too "terse" and tends to obscure rather than clarify loop control. It is included for "compatibility" with Mortran2, and because the author likes it.
### 3.3.2.7 "Forever" loops

```fortran
LOOP [...] 
LOOP [...] REPEAT
```

REPEAT is optional.

The "forever" loop generates the following FORTRAN:

```fortran
L1 CONTINUE
...
GO TO L1
L2 CONTINUE
```

The "exit" from "forever" loops must be made by a statement within the loop. The label L2 is generated by MortranJ as the "target" for the EXIT, whose description follows.

### 3.3.2.8 EXIT

EXIT is "shorthand" for "jump out of this loop", where this loop refers to the innermost loop in which the EXIT appears.

```fortran
EXIT;
IF (e) EXIT;
IF e [...EXIT;]
```

In the "forever" loop EXIT generates "GOTO Lz", where Lz refers to the "target" mentioned above.

But EXIT may appear in any MortranJ loop.

In a MortranJ loop, EXIT causes a transfer of control to the first statement following the loop in which it occurs. For nested loops, the EXIT is from the innermost loop to the next level out.
3.3.2.9 NEXT

NEXT is "shorthand" for "go to the next iteration of this loop", where this loop refers to the innermost loop in which the NEXT appears.

```
NEXT;
IF (a) NEXT;
IF b [...]NEXT;
```

NEXT transfers control to the beginning of the loop in which it occurs.

- The control variable (if any) is incremented
- The test (if any) for loop termination is made.
- If tests at both ends of the loop are involved, only the test at the beginning of the loop is made.
- In FOR-loops and DO-loops, NEXT generates GOTO L1, where L1 refers to the label on the statement that increments the control variable.
- In DO-loops, the statement that increments the control is the CONTINUE that marks the end of the "range" of the DO.

Phrases control the execution of blocks. But notice that EXIT and NEXT refer to iterative control phrases, and not to sequential control phrases.

**Example:** UNTIL A=B [...]IF C=D [...]EXIT;] P=Q; [...] R=S; ...

In this example EXIT transfers control to the statement R=S, and not to the statement P=Q.
3.3.3 Multi-level Control

There are two categories: sequential, and iterative.

* Multi-level iterative control statements are:
  
  EXIT:label;
  NEXT:label;

* Multi-level sequential control statements are:
  
  EXECUTE procedure name
  LEAVE
  GOTO :label;

3.3.3.1 Multi-level EXIT

In nested loops, it may be desirable to EXIT more than one level. EXIT of more than one level of nesting may be made only from labeled loops. A labeled loop is a loop preceded by an alphanumeric label. A Multi-level EXIT is made by:

```
EXIT :label: ;
```

EXIT:label: transfers control to the first statement following the labeled loop. The transfer of control takes place regardless of nesting, and thus provides a "multi-level" EXIT capability.

```
10 CONTINUE
11 IF(mort)GOTO 12
   cherchez
   la verite
21 IF(eins.EQ.zwei)GOTO 22
   oktober
   IF(tod)GOTO 12
   IF(angst)GOTO 22
   fest
   GOTO 21
22 CONTINUE
   cest
   la vie
12 CONTINUE
   ho
   hum
```

A. James Cook

December 16, 1982
### 3.3.3.2 Multi-level NEXT

**NEXT** : label: transfers control in the manner described above for NEXT, but may be multi-level. The following example contains both single-level and multi-level EXITs and NEXTs.

```plaintext
:SEARCH: FOR V=Ax BY Ai TO AF
   [ ... ]
   :ROW: DO I=I,N
   [ ... ]
   :COL: DO J=I,M
   [ ... ]
   UNTIL 00
   [ ... ]
   IF(e1) EXIT :SEARCH ;
   [ ... ]
   IF(e2) NEXT :SEARCH ;
   [ ... ]
   IF(e3) NEXT :ROW ;
   [ ... ]
   IF(e4) EXIT :ROW ;
   [ ... ]
   IF(e5) NEXT :COL ;
   [ ... ]
   IF(e6) NEXT ;
   [ ... ]
   IF(e7) EXIT ;
   [ ... ]
   "end UNTIL"
   [ ... ]
   "end COL"
   [ ... ]
   "end ROW"
   [ ... ]
   "end SEARCH"
```

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>CONTINUE V=Ax GOTO 13</td>
</tr>
<tr>
<td>11</td>
<td>V=V+(Ax)</td>
</tr>
<tr>
<td>13</td>
<td>IF((Ai)*V-(AF))*GT.0)GOTO 12</td>
</tr>
<tr>
<td>20</td>
<td>CONTINUE DO 21 I=I,N</td>
</tr>
<tr>
<td>30</td>
<td>CONTINUE DO 31 J=I,M</td>
</tr>
<tr>
<td>41</td>
<td>IF(e8)GOTO 42</td>
</tr>
<tr>
<td>42</td>
<td>CONTINUE</td>
</tr>
<tr>
<td>31</td>
<td>CONTINUE</td>
</tr>
<tr>
<td>32</td>
<td>CONTINUE</td>
</tr>
<tr>
<td>21</td>
<td>CONTINUE</td>
</tr>
<tr>
<td>22</td>
<td>CONTINUE</td>
</tr>
</tbody>
</table>

A. James Cook  
- 21 -  
December 16, 1982
3.3.3.3 EXECUTE local procedure

A local procedure has the form:

```
name PROCEDURE [ ... ]
```

where `name` is alphanumerically.

A local procedure is "called" by the statement:

```
EXECUTE name;
```

Any number of EXECUTE statements may precede a procedure.

The "calls" must precede the Procedure:

```
... EXECUTE PHI ;
... EXECUTE PHI ;
... PHI PROCEDURE [ ... ]
...;
```

After the procedure is executed control is returned to the first statement following the EXECUTE statement. An EXECUTE statement must not follow a procedure:

```
... EXECUTE PHI ;
... PHI PROCEDURE [ ... ]
...;
```

- The first EXECUTE statement will "call" PHI, and will return correctly.
- The second EXECUTE statement will "call" PHI, but will not return correctly.

Multiple returns from a procedure are written:

```
LEAVE name;
```
Example of FORTRAN generated for procedures:

```
EXECUTE RHO;
EXECUTE PHI;
EXECUTE RHO;

PHI PROCEDURE [ ... LEAVE PHI; ... ]
RHO PROCEDURE [ ... LEAVE RHO; ... ]
```

Caveat!

Program units containing local procedures must avoid identifiers of the form:

```
K\text{Pn}
```

where \( n \) comprises 2 to 4 decimal digits.

(See the example above.)

Moreover, the program unit must not override the FORTRAN convention that identifiers beginning with "K" default to type INTEGER.
Examination of the FORTRAN generated for EXECUTE and PROCEDURE statements will reveal that:

1) It is not possible to "fall-into" a procedure, because a branch around the procedure is generated by MortranJ.

2) It is not necessary to issue LEAVE statements explicitly unless multiple returns are needed.

3) EXIT :name: is equivalent to LEAVE :name:. LEAVE should be used to avoid confusion with loops.

It should also be pointed out that "GOTO procedure :name:" will generate a FORTRAN compile-time error of "MISSING LABEL". (See the following section.)
4. CONTROL SUMMARY

4.1 SUMMARY OF ITERATIVE CONTROL

We can now summarize MORTRAN loops in the following chart:

<table>
<thead>
<tr>
<th>optional</th>
<th>choose one</th>
<th>optional</th>
<th>optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHILE e1</td>
<td>UNTIL e1</td>
<td>NEXT;</td>
<td>WHILE e2;</td>
</tr>
<tr>
<td>LOOP</td>
<td>FOR V=Ax BY Ai TO Af</td>
<td>EXIT;</td>
<td>UNTIL e2;</td>
</tr>
<tr>
<td>:label:</td>
<td>FOR V=Ax TO Af BY Ai</td>
<td>EXIT :label:;</td>
<td>REPEAT</td>
</tr>
<tr>
<td></td>
<td>FOR V=Ax TO Af</td>
<td>NEXT :label:;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DO i=j,k,n</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

where:

- e1 and e2 are logical expressions.
- V is a (subscripted) variable of type INTEGER or REAL.
- Ax, Ai, and Af are arbitrary arithmetic expressions.
  - Ax is the initial value.
  - Ai is the increment.
  - Af is the final value.
- i, j, k, and n are standard FORTRAN DO variables:
  - i is a non-subscripted INTEGER variable.
  - j, k, and n are non-subscripted INTEGER variables or INTEGER constants.

The "compact" DO-loop does not "fit" in the above chart, however, the rules that apply to DO i=j,k,n [...] also apply to the "compact" DO-loop [i=j,k,n; ...]
4.2 SUMMARY OF SEQUENTIAL CONTROL

Sequential control statements are IF, IF-ELSE, ELSEIF, and GOTO.

4.2.1 GOTO :label:

This statement generates a standard FORTRAN GOTO statement. Mortran3 does not prohibit branching into the range of a loop, consequently:

- If the loop is a DO-loop, no error will be generated by Mortran3, but an error will be generated by most FORTRAN compilers.

- Similarly, GOTO procedure name will not generate Mortran3 errors, but will generate FORTRAN compile-time errors.
5. MISCELLANEOUS FEATURES

5.1 MULTIPLE ASSIGNMENT

produces:

\[
\begin{align*}
& V_1 = \text{expression} \\
& V_2 = \text{expression} \\
& V_3 = \text{expression} \\
& \quad \vdots \\
& V_n = \text{expression}
\end{align*}
\]

NOTICE to Mortran2 users: this has been changed.

Mortran2 produced:

\[
\begin{align*}
& V_1 = \text{expression} \\
& V_2 = V_1 \\
& V_3 = V_2 \\
& \quad \vdots \\
& V_n = V_{n-1}
\end{align*}
\]

This was changed because:

1) The Mortran2 version may cause conversion errors when the \( V_i \) are different types.

2) This new version is "what one would expect".
5.2 INPUT-OUTPUT ABBREVIATIONS

These abbreviations may be used to read and write from the "standard" units.

**Input**

```
INPUT i-o list;(format list);
```

Where *i-o list* and *format list* must meet standard FORTRAN requirements.

**Example:**

```
INPUT X,Y ; (2F9.3);
```

**Produces:**

```
READ(5, 10)X,Y
10 FORMAT (2F9.3)
```

**Output**

```
OUTPUT i-o list;(format list);
```

Where *i-o list* and *format list* must meet standard FORTRAN requirements.

**Example:**

```
OUTPUT Z ; ('VOILA LE Z','E12.6)
```

**Produces:**

```
WRITE(6, 20)Z
20 FORMAT ('VOILA LE Z','E12.6)
```

**INPUT** and **OUTPUT** may be generalized to include other logical i-o units. (see Part II).
PART II. The Definition Language

6. INTRODUCTION TO NOTATIONAL TRANSFORMATIONS

The programming language described in Part I is defined by a set of transformation rules. The transformation from Mortran notation to FORTRAN notation is the result of the application of these rules by the MortranJ processor.

The application of a rule involves:

- "Recognition" of a notational "form" and
- Substitution of FORTRAN "form(s)" for the "form" that was "recognized".

We will give precise definitions of these "fuzzy" statements later. Before going into that, we present a simple, useful example.

6.1 A USEFUL EXAMPLE

Suppose one has a program in which subscripts are calculated, and that errors will result if the subscripts are not within certain ranges. For example:

\[ j < j < 7 \]

One could insert IF, WRITE, and FORMAT statements that issue error messages if these variables are not within the given ranges. But such error-checking statements are obtrusive; they distract our attention from more important aspects of the program. A much less obtrusive (and easier to write) notation might be:

Example (6-1) \[ \text{ASSERT } j < j < 7 \]

One wants to transform notations like (6-1) into FORTRAN statements that would produce run-time error messages like:

\[ \text{ASSERTION "} j < j < 7 \text{" VIOLATED: } j = JJ \]
A more general form of (6-1) is:

\[(6-2)\quad \text{ASSERT digit1 < letter < digit2}\]

The FORTRAN notation one wants to substitute for (6-2) should depend upon the actual letters and digits corresponding to \(\text{digit1}\), \(\text{letter}\), and \(\text{digit2}\).

The transformation described above could be defined by:

\[
\begin{align*}
\text{REPLACE} \ (\text{ASSERT}(\text{DIGIT})<\{\text{LETTER}\}<\{\text{DIGIT}\}) \\text{WITH} \\
(6-3) \quad \{ \text{IF}( \neg((P1)<(P2) \& (P2)<(P3)) \) OUTPUT (P2); \} \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad 

A transformation rule is defined by (6-3). An application of (6-3) is illustrated in the following example:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Produces:</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASSERT 7 &lt; K &lt; 9 ;</td>
<td>IF(7.GE.K.OR.K.GE.9) WRITE(6,10)</td>
</tr>
<tr>
<td></td>
<td>10 FORMAT(' ASSERTION &quot;7&lt;K&lt;9&quot; VIOLATED: K =',I3)</td>
</tr>
</tbody>
</table>

"7" corresponds to the 1st (DIGIT) in (6-3)
"K" corresponds to the (LETTER) in (6-3)
"9" corresponds to the 2nd (DIGIT) in (6-3)

1) In (6-3), (DIGIT) and (LETTER) are parameter markers.
2) In the example, 7, K, and 9 are actual parameters.
3) There are three parameter markers in (6-3).
4) There are nine parameter references in (6-3)
   Five of them are denoted by (P2)
   Two of them are denoted by (P1)
   Two of them are denoted by (P3)
5) OUTPUT is a notation described in Part I.
6) The braced quantities (DIGIT) and (LETTER) are parameter markers.
The quantities DIGIT and LETTER are specifiers.
   * In this case the specifiers are primitives; they are "built-in".
But the programmer may define specifiers that suit his own needs. For example, a specifier may be defined that describes a variable name. A FORTRAN variable name is:

A letter followed by not more than 5 letters or digits.

A MortranJ specifier that could be used to "recognize" a FORTRAN variable name is defined by the following SPECIFY statement:

```
(6-4) SPECIFY FORTNAME AS LETTER (0,5)[LETTER|DIGIT] ;
```

Following the definition (6-4), (FORTNAME) could replace (LETTER) in the ASSERT rule definition above so that the ASSERT notation could be used with FORTRAN variable names instead of single letters. But something must be done about DIGIT also. The "something" follows:

```
SPECIFY CONSTANT AS (1,7)[DIGIT];
```

Now, one can re-write the transformation rule as:

```
REPLACE (ASSERT(CONSTANT)<(FORTNAME)<(CONSTANT))
WITH
(6-6) ( IF( ~(P1)<(P2) & (P2)<(P3) ) OUTPUT (P2); (' ASSERTION "(P1)<(P2)<(P3)" VIOLATED: (P2) = ',I9); )
```

Following the definition of (6-6), one might write:

```
ASSERT 1234 < INDEX < 5678;
```

Producing, at run-time:

```
ASSERTION "1234<INDEX<5678" VIOLATED: INDEX=98765
```

This introductory example is intended to whet your appetite for bigger and better things so that you will read Part II, which is excerpted from "The MortranJ Reference Manual" (by yours truly). And maybe even wade through the Reference Manual itself!
7. TRANSFORMATION RULES

The Mortran3 processor may be regarded as a device that accepts and applies transformation rules.

- A rule may be defined at any point in a program.
- A rule applies to all program text after it is defined. 
- A rule may define one or more rules. (Definitions may be nested.)

To define a rule, the programmer writes:

```
REPLACE (template) WITH (value)
```

- The keywords REPLACE and WITH are intended as a constant reminder that a rule defines a substitutive function. 
- Template describes a user-defined notation.
- Value comprises text (and/or a prescription constructing text) that will be substituted for the user-defined notation.
- The braces are delimiters.

7.1 THE SIMPLEST EXAMPLE.

In the simplest type of rule, both the template part and the value part are made up of literal symbols (symbols that denote themselves). This type of rule merely substitutes one symbol-string for another symbol-string.

Example (7-1)  

```
REPLACE (SIZE) WITH (22)
```

Rule (7-1) would transform all occurrences of (say):

```
DIMENSION A(SIZE)
```

into

```
DIMENSION A(22)
```

Rules like (7-1) are, by themselves, of limited usefulness. 

The user may define rules of considerable power and flexibility by including braced quantities within the template and value parts. If braced quantities are included in the template and value parts the braces must be balanced, just as parentheses must be balanced in FORTRAN expressions.

---

6 The scope of rule application may be restricted by the programmer (See Section 'The Scope of Rules' below.)

7 A function that is evaluated by substitution as opposed to transfer of control. See "Mortran3 Reference Manual", appendix E for a FORTRAN analogy.

8 They may be used like FORTRAN 77's PARAMETER statement.
Braced quantities in rule definitions denote function references.

- In the template part they denote recognition functions.
- In the value part they denote evaluation functions.

7.2 **THE TEMPLATE PART**

A template comprises literal symbols and, optionally, braced quantities.

- Templates must contain literal symbols.
- Templates may contain braced quantities.

The name within the braces is the name of a recognition function.
The name and the braces comprise a parameter marker.

7.2.1 **Parameter Markers**

A parameter marker may be thought of as a "place holder":

```
template (7-2) THIS (WORD) IS FALSE.
```

Template (7-2) is made up of:

- Literal symbols: "THIS IS FALSE."
- A parameter marker: (WORD)
- A specifier: WORD

A specifier is the name of a function. The function specifies criteria that
must be met by an actual parameter.

Assume that WORD has been specified (described below) as:

"a contiguous string of upper-case letters"

Then, (7-2) would match THIS SENTENCE IS FALSE. ; parameter: "SENTENCE".

- A template may contain up to 35 parameter markers.
- The parameter markers in a template are "positional". That is, they are numbered (by Mortran) from the left, starting with one.
template (7-3)  

(7-3) would match:  

1st actual parameter is "THIS"  
2nd actual parameter is "THAT"

Template (7-3) would also match:  
= (A.B)  
= (ANTIDISESTABLISHMENTARIANISM.DIALECTICALMATERIALISM)

Template (7-3) would not match:  
= (3.14) because 3 is not a word.  
= (A.EQ.B) because EQ.B is not a word.

One "tells Mortran" how to "recognize" a parameter by means of the SPECIFY statement, which is described in the next section.
7.2.2 Specifiers

A specifier is the name of a recognition function. The function is defined by a SPECIFY statement:

```
SPECIFY specifier AS specification;
```

- A specification is a boolean function.
  1. A specification is met or it is met-met.
  2. A specification comprises:
     a. Specifiers.
     b. Primitive specifiers.
     c. Primitive specifications.
        1) Specifiers are user-defined.
        2) Primitives are system-defined.
- A specifier is satisfied if its specification is met.

```
A template will not match a segment of text unless all of its specifiers are satisfied.
```

Example:
```
SPECIFY HEXDIGIT AS (0...F);
```
- This SPECIFY statement defines the specifier HEXDIGIT.
- HEXDIGIT would be satisfied by one of the symbols 0 through F. HEXDIGIT would not be satisfied by any other symbol.
- The parameter marker (HEXDIGIT) would "match" a single digit. The parameter marker (HEXDIGIT) would not "match" any other symbol.
- (0...F) is an example of a primitive specification.

Example:
```
SPECIFY LCASE AS (a...z);
```
- This specify statement defines the specifier LCASE, which would be satisfied by one lower-case letter.
- The specification part of this specify statement is an example of a primitive specification, namely, a qualifier.

There are a number of primitive (system defined) specifiers. Three very useful ones are LETTER, DIGIT, and ALPHA; They are predefined by the system as follows:
(7-4) b SPECIFY DIGIT AS (0...9);  
c SPECIFY ALPHA AS [LETTER|DIGIT];

Notice that (7-4)-c is specified in terms of other specifiers, namely LETTER and DIGIT. LETTER and DIGIT comprise a list of two alternatives. ALPHA is satisfied if either LETTER or DIGIT is satisfied. The symbol "|" is the "alternation symbol". The square brackets delimit the list of two alternatives. (In general, the number of alternatives in a list is arbitrary.)

Square brackets also delimit entities that are quantified.

Example:

SPECIFY FNNAME AS LETTER(0,5)[ALPHA];

- The specification part contains a quantifier.
  a) The entity being quantified is enclosed in square brackets.
  b) The quantifier is enclosed in parentheses.
- The quantifier may be read
  a) "at least zero, but not more than five" or
  b) "optionally at most five".
- FNNAME would be satisfied by a letter followed, optionally, by at most 5 letters or digits.
- FNNAME would be satisfied by a standard FORTRAN variable name.

Example:

SPECIFY THING AS 'T'(2)[LETTER|DIGIT];

- THING would be satisfied by the letter T followed by exactly two symbols.
- Each of the symbols may be either a letter or a digit.
- The specification part in this example comprises
  a) the literal 'T',
  b) two specifiers LETTER and DIGIT,
  c) one quantifier, (2)
  d) a list comprising two alternatives, [LETTER|DIGIT].
7.2.2.1 Forward references in Specifications

Forward references are permitted in specifications. For example,

\[
\begin{align*}
\text{SPECIFY QUADRUPED AS } & (4)\{\text{LEG}\}; \\
\text{SPECIFY LEG AS } & \{\text{LIMB} \mid \text{EXTREMITY} \}; \\
\end{align*}
\]

In (7-5) LEG is referred to before it is defined; LEG is a forward reference.

Forward references may increase the computation done during transformations.

It should also be pointed out that QUADRUPED will be satisfied by any of 16 \((=2^4)\) possible arrangements of LIMB and EXTREMITY.

If we write "L" for "LIMB" and "E" for "EXTREMITY", then the following arrangements would satisfy the specifier QUADRUPED.

\[
\begin{align*}
L & L L L, L L E L, L L E E, \\
L & E L L, L E L E, L E E L, L E E E, \\
E & L L L, E L L E, E L E L, E L E E, \\
E & E L L, E E L E, E E E L, E E E E
\end{align*}
\]

7.2.2.2 Specifying "Optional"

There are two ways that a specification may be made "optional".

\[
\begin{align*}
(7-6)-a & \text{ SPECIFY APPENDAGE AS } \{\text{TAIL}\}^*; \\
(7-6)-b & \text{ SPECIFY APPENDAGE AS } (0,1)\{\text{TAIL}\}; \\
\end{align*}
\]

Both (7-6)-a and (7-6)-b specify that APPENDAGE is "optional". In other words, the text "TAIL" will satisfy the specifier APPENDAGE, but APPENDAGE will be satisfied whether or not the text "TAIL" is present.

- If "TAIL" is present, it becomes the actual argument.
- If "TAIL" is absent, the actual argument is NULL.

More about this in the subsection 'NULL Parameters' below.
7.3 **THE VALUE PART**

The value part comprises any or all of the following:

- Literal symbols
- Strings
- Braced quantities
- Evaluation-time IF-THEN-ELSEs.

Braced quantities (not in strings) always denote function references. The simplest of the function references are the parameter references.

7.3.1 **Parameter References**

A parameter reference is a primitive function that refers to an actual argument. The first of these is denoted by:

```
(Pn)
```

\[1 \leq n \leq 35\]

The primitive function invoked by (Pn) returns actual argument n.

- n is the parameter reference number
- n corresponds to the relative position (from the left) of the parameter marker.
- The number of parameter markers may not exceed 35, but any number of parameter references may appear in the value part.
- The parameter reference number n may not exceed the number of parameter markers. (If it does the error message "NO SUCH ARGUMENT IN TEMPLATE" will result.)

Other primitives that refer to actual parameters are:

```
(LENGTH n)
(EXIST n)
(NULL n)
```

- (7-7)-1 returns the length of actual argument n.
- (7-7)-2 and -3 refer to the "existence" or "non-existence" of the nth actual argument.

Briefly, the parameter "exists" if its length is greater than zero, and is "null" if its length is equal to zero. But see the section "NULL Parameters" below.
7.3.1.1 Left-to-right Substitution

Normally, the value part of a rule is substituted for the template part on a strict left-to-right (L-R) basis.

\[
(7-8) \quad \text{REPLACE } (((\text{LETTER}) \cdot \text{LETTER}))) \\
\text{WITH} \quad \{ \{P1\}(1) \cdot \{P2\}(1) \\
+ \{P1\}(2) \cdot \{P2\}(2) \\
+ \{P1\}(3) \cdot \{P2\}(3) \}
\]

- The template part of rule (7-8) contains two parameter markers.
- The value part of rule (7-8) contains six parameter references.
  Three of them refer to the first actual parameter.
  Three of them refer to the second actual parameter.

Application of (7-8) to the

<table>
<thead>
<tr>
<th>program text</th>
<th>(A.B)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A(1)*B(1) + A(2)*B(2) + A(3)*B(3)</td>
</tr>
</tbody>
</table>

Since (7-8) contains only literal symbols, parameter markers, and parameter references, application of (7-8) involves the following steps:

1. The template part matches the text (A.B).
2. The parameters A and B are saved and the text "(A.B)" is deleted.
3. Beginning at the point where the text "(A.B)" was deleted, the value part is substituted. The substitution proceeds left-to-right and symbol-by-symbol until a parameter reference is encountered.
4. The actual parameter referred to is inserted at the reference point.
5. Left-to-right substitution resumes until the next parameter reference is encountered.

Upon completion of the substitution, the newly substituted text is re-scanned to determine whether other rules may be applied to it.

Simple left-to-right substitution may be altered by the programmer through the use of primitive evaluation functions and evaluation-time IF-ELSEs described below.
7.3.1.2 **NULL Parameters**

The programmer may define notations whose parameters are described in considerable detail by means of SPECIFY statements.

In particular, he may specify that a parameter is "optional". One of the ways he may do this is illustrated in the following example:

\[
\begin{align*}
\text{(7-9)-a:} & \quad \text{SPECIFY BIPED AS } (2)[\text{LEG}]; \\
\text{(7-9)-b:} & \quad \text{SPECIFY APPENDAGE AS } (0,1)[\text{TAIL}]; \\
\text{(7-9)-c:} & \quad \text{SPECIFY APPENDAGE AS } [\text{TAIL}'];
\end{align*}
\]

Both (7-9)-b and (7-9)-c specify that APPENDAGE is "optional". (7-9)-a specifies that BIPED requires exactly 2 occurrences of LEG. Now consider the template fragment:

\[\ldots\{\text{BIPED}\}{\text{APPENDAGE}}\ldots\]

If the specifier BIPED is satisfied, and the "\ldots" parts are satisfied, then the template matches regardless of whether or not the specifier TAIL is satisfied. In other words, the specifier APPENDAGE will always be satisfied.

If a specifier has been defined as optional, then:

- The actual parameter EXISTS if it contains at least one symbol.
- The actual parameter is NULL if it contains no symbols.

(NULL = does not EXIST = contains no symbols.)

Now, in the value part of a rule, the programmer may well want to know whether the BIPED has a tail or not, and to take action accordingly. The programmer may test for the existence of a parameter that has been declared optional. More about this following the evaluation-time IF-ELSE.

A.James Cook

- 40 -

December 16, 1982
Simple left-to-right substitution may be altered by:

- Evaluation function references.
- Evaluation-time IF-THEN-ELSEs.

### 7.4 PRIMITIVE EVALUATION FUNCTIONS

A primitive evaluation function reference is denoted by:

```
(NAME argument)
```

- `NAME` is the name of the function and
- `argument` is a symbol-string.

The argument of an evaluation function may contain other function references. In other words, function references may be nested. Nested primitive references (excepting parameter references) are evaluated from the innermost outward.

#### 7.4.1 Order of evaluation

Let `P`, `Q`, and `R` denote primitive function names, and let an ellipsis denote an arbitrary symbol-string.

```
(7-10)   (P...(Q ...(R ...) ...) ...) ...
```

In (7-10), the primitive `R` is evaluated first. The argument of `Q` then includes the symbol-string resulting from `R`'s evaluation, and so on until `P` is evaluated. (Each evaluation results in the removal of the name and delimiting braces of the function.)

Let upper-case letters denote the names of primitive functions, lower-case letters denote symbol-strings, and boldface letters denote evaluated symbol-strings.

Then, `(P ab(Q cd(R ef) gh) ij)` is evaluated as follows:

- (after `R` is applied.)
- (after `Q` is applied.)
- (after `P` is applied.)
7.4.2 The REDUCE primitive

First, we illustrate with a contrived example involving J rules:

\begin{verbatim}
(7-11)-a  REPLACE (xxx) WITH ((nJ)
  -b  REPLACE (n) WITH (2)
  -c  REPLACE (/Z3) WITH (ONE,TWO,THREE!)
\end{verbatim}

Rules (7-11) will transform "xxx" into "123", but "123" will not be transformed into "ONE,TWO,THREE!". Rules (7-11) will be applied in the following steps:

1) "xxx" is replaced by "lnJ" by (7-11)-a.
2) "n" is replaced by "2" by (7-11)-b.

As pointed out in section "L-R Substitution" above, the substituted text is "re-scanned" to determine whether other rules may be applied. But the text is re-scanned from the point at which the last substitution was made. In this example, "123" will not be matched because the re-scan begins at "23" after (7-11)-b is applied. We can force the application of (7-11)-c by re-writing (7-11)-a to invoke REDUCE:

\begin{verbatim}
(7-11)-a-REV  REPLACE (xxx) WITH (REDUCE((lnJ))
\end{verbatim}

Then, "xxx" will be transformed into "ONE,TWO,THREE!" in the following steps:

1) "xxx" is reduced to "lnJ" by (7-11)-a-REV.
2) "n" is reduced to "2" by (7-11)-b.
3) "123" is reduced to "ONE,TWO,THREE!" by (7-11)-c.
4) "xxx" is replaced by "ONE,TWO,THREE!"

7.4.2.1 REDUCED text vs. REPLACED text.

The distinction between reduced and replaced is an important one.

REDUCE rescans \textit{all} of its argument until all rules have been applied to the argument. When no further reduction is possible, the reduced string replaces the matched string.

If the argument of REDUCE contains other invocations of REDUCE then the innermost is re-scanned first. (This is not unlike nested FUNCTION references in FORTRAN.)
REDUCE is automatically invoked by the COMPUTE primitive.

7.4.3 The COMPUTE primitive

Again, the contrived introductory example:

\[
\begin{align*}
(7-12)-a & \quad \text{REPLACE (TWO) WITH (2)} \\
-b & \quad \text{REPLACE (SIX) WITH (6)} \\
-c & \quad \text{REPLACE (APE) WITH ((COMPUTE \, 3\times(TWO+SIX)))}
\end{align*}
\]

Rules (7-12) will transform "APE" into "24" in the following steps:
1) TWO is reduced to 2 by (7-12)-a.
2) SIX is reduced to 6 by (7-12)-b.
3) The constant expression "3\times(2+6)" is evaluated.
4) The value is converted to the two characters "2" and "4".
5) The text "24" replaces the text "APE".

- COMPUTE returns the value expressed as a digit-string.
- COMPUTE's argument must be reducible to a constant expression.
  (IF it is not reducible, various error messages will result)

7.4.3.1 Constant expressions

Constant expressions comprise integer constants and operators.

An expression like:

\[
\begin{align*}
\text{DOGS} & > \text{CATS} \\
\text{POODLES+CORGIES} & > \text{SIAMESE+ALLEY}
\end{align*}
\]

could reduce to (say):

\[
\begin{align*}
5+7 & > 2+32768
\end{align*}
\]

and then to (say):

\[
\begin{align*}
5+7 & > 3.2768
\end{align*}
\]

this last is a constant expression.

Constant expressions are either integer expressions or logical expressions.

A constant expression comprises:

<table>
<thead>
<tr>
<th>Integer constants</th>
<th>(contiguous digit-strings)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic operators</td>
<td>(+, -, *, /, **)</td>
</tr>
<tr>
<td>Relational operators</td>
<td>( &lt;, &lt;=, =, &gt;=, &gt; )</td>
</tr>
<tr>
<td>or</td>
<td>(LT, LE, EQ, NE, GE, GT )</td>
</tr>
<tr>
<td>Logical operators</td>
<td>( AND, OR, NOT )</td>
</tr>
<tr>
<td>or</td>
<td>( &amp; ,</td>
</tr>
<tr>
<td>Parentheses</td>
<td></td>
</tr>
</tbody>
</table>

(The FORTRAN operators .AND., .EQ., .NE., etc. are NOT acceptable in this context.)
7.4.3.2 Constant Logical Expressions

For constant logical expressions COMPUTE returns:
the digit 1 (TRUE) or
the digit 0 (FALSE)

Assuming rules (7.12)-a and (7.12)-b above,

(COMPUTE TWO LT SIX ) returns 1 (TRUE)
(COMPUTE TWO < SIX ) returns 1 (TRUE)
(COMPUTE TWO EQ SIX ) returns 0 (FALSE)
(COMPUTE TWO=SIX+6 > 30-TWO ) returns 0 (FALSE)
(COMPUTE TWO*SEX+6 > 30-TWO ) issues error message if "SEX" is not reducible. Returns 0.

NOTE

An arithmetic expression that evaluates to zero is not distinguishable from a logical expression that evaluates to FALSE. Therefore, it is possible (but not advisable) to perform arithmetic operations on logical expressions.
7.4.4 Evaluation-time IF-ELSE

The evaluation-time IF-ELSE is written:

\[
\text{[IF] \hspace{1em} \text{predicate} \hspace{1em} \text{[true-part]} \\
\text{[ELSE]} \hspace{1em} \text{[false-part]}}
\]

- predicate is (1 a constant expression, 9
  (2 a string expression
  (3 the primitive \text{EXIST}
  (4 the primitive \text{NULL}

- true-part and false-part are symbol strings.

The IF-ELSE in a programming language controls the order of execution of the statements in a program.

This evaluation time \text{[IF]-[ELSE]} controls the substitution of symbol-strings. This \text{[IF]-[ELSE]} should be regarded as a function that returns the symbol-string \text{false-part} if predicate is FALSE, and the symbol-string \text{true-part} otherwise.

- Evaluation-time \text{[IF]-[ELSE]}'s may be nested.

\(^9\text{More precisely, it is an expression that must be reducible to a constant expression. (See subsection 'Constant Expressions'.)}\)
7.4.4.1 String Expressions

String expressions involve two types of strings:

1) Literals enclosed in apostrophes (like 'THIS').
2) Parameter markers enclosed in apostrophes (like '(P1)').

A parameter reference enclosed in apostrophes converts the actual argument to a string. String expressions are written:

```
S1 relation S2
```

- S1 must be a string
- S2 may be (1 a string, or (2 a specifier
- relation is (1) the symbol "="
  (2) the symbol "~"

If S1 is not a string, unpredictable errors will result. If S2 is not a string, it is assumed to be a specifier.

In summary, string expressions may contain

- Strings
  - Literals
  - Parameter references
- Specifiers

If both arguments are strings, the strings are compared.

If the first argument is a string and the second argument is a specifier, then if the string satisfies the specifier and if the relation is "=" then TRUE is returned.

Undefined specifiers in string expressions result in an error message and a value of FALSE for the predicate.

The following are valid string expressions involving strings only:

- `'(P1)'='(P2)'` TRUE when parameter 1 is identical to parameter J.
- `'(P1)'~'(P2)'` FALSE when parameter 1 is identical to parameter J.
- `'(P1)'='THIS'` TRUE when parameter 1 is the literal THIS.
- `'THIS'='(P1)'` TRUE when parameter 1 is the literal THIS.
- `'(P1)'~'THIS'` TRUE when parameter 1 is not the literal THIS.
The following string expressions involve a string and a specifier:

- 'P(F)\)FNAME\] TRUE when parameter F satisfies the specifier FNAME.
- 'P(F)\)FNSME issues an error message if FNSME has not been defined in a SPECIFY statement and returns FALSE.

7.4.4.2 The primitives NULL and EXIST

Actual parameters may be NULL (contain no symbols). This can happen when a specifier has been defined as optional. (See subsection NULL Parameters.) The primitives EXIST and NULL may be used to determine the status of actual arguments.

\[
\begin{array}{|c|}
\hline
\text{(EXIST n)} \\
\text{(NULL n)} \\
\hline
\end{array}
\]

n refers to the nth actual parameter

- (EXIST n) returns TRUE if actual argument n is not null.
- (EXIST n) returns FALSE if actual argument n is null.
- (NULL n) returns TRUE if actual argument n is null.
- (NULL n) returns FALSE if actual argument n is not null.

These primitives are especially useful in evaluation time IF-ELSE predicates.
7.3 **Recursive Application of Rules.**

A rule may be evaluated recursively. **Direct**\(^\text{10}\) recursion results when the name of the rule appears in the value part of the rule. We give two illustrations:

- The first illustrates the transformation of a functional notation for factorial into an expression.
- The second calculates the factorial.

First, we transform the notation: \( !d \) (where \( d \) is a digit) into the notation: \( d!(d-1)!(d-2)\ldots!(1) \)

**REPLACE \((!\text{DIGIT}))\right\)**

\[(7-13)\]

\[\text{WITH } \begin{cases} \text{IF } \{\text{PI}\}>1 & \{\text{PI}\} * ! \{\text{COMPUTE } \{\text{PI}\}-1\} \\ \text{ELSE} & 1 \end{cases}\]

Rule (7-13) will transform: \( !9 \) into: \( 9*8*7*6*5*4*3*2*1 \)

in the following steps:

- First step: \( !9 \) is replaced by \( 9*8 \)
- Second step: \( 8 \) is replaced by \( 7*6 \)
- Third step: \( 6 \) is replaced by \( 5*4 \)
- Fourth step: \( 4 \) is replaced by \( 3*2*1 \)

Rule (7-13) is not intended to be useful; It is intended to be a simple illustration of recursive application of a rule.

- The name \( (!) \) appears in the value part of \( (7-13) \) with a new argument, to wit: \( !\{\text{COMPUTE } \{\text{PI}\}-1\} \).

- When \( \{\text{PI}\}=1 \), the predicate returns FALSE, and the digit "!" is substituted, preventing further recursion.

---

\(^\text{10}\) Indirect recursion can also take place when two or more rules "invoke" one another.

A. James Cook

December 16, 1982
In the second example, the factorial is computed by nested invocations of COMPUTE:

```plaintext
REPLACE [[[DIGIT]]] WITH [[[IF] (P1)>1 [[[COMPUTE (P1)!{COMPUTE (P1)-1} ] ] [ELSE] [1]]]
```

Rule (7-14) will transform 15 into 120 as follows:

15 is reduced:
(COMPUTE 5!*14) (nest level=1)

14 is reduced:
(COMPUTE 5*(COMPUTE 4!*13)) (level=2)

13 is reduced:
(COMPUTE 5*(COMPUTE 4*(COMPUTE J!*12))) (level=3)

12 is reduced:
(COMPUTE 5*(COMPUTE 4*(COMPUTE J*(COMPUTE 2*(COMPUTE 1!)))))) (level=4)

Now, because the argument of 1! is not greater than 1, recursion stops and 1 is substituted. The nest is then evaluated from the innermost outward:

```plaintext
(COMPUTE 5*(COMPUTE 4*(COMPUTE J*(COMPUTE 2*(COMPUTE 1!))))
(COMPUTE 5*(COMPUTE 4*(COMPUTE J*2)))
(COMPUTE 5*(COMPUTE 4*6))
(COMPUTE 5*24)
120
```

When computation is complete (level=0), replace the text "15" with "120".

If the ELSE part of (7-14) is removed, 15 will not return 120 as one might expect, it will return 0, and COMPUTE will issue the error messages:

```
#WARNING INVALID C-EXPRESSION #STRANGE (2*)
```

Rule (7-14) (without the ELSE part) attempts to compute:

```
(5*(4*(3*(2!)))
```

which is meaningless. Rather than abandon the rule, it issues error messages and computes:

```
(5*(4*(3*0))
```

which may not be what was intended. Thus, the ELSE part of (7-14) serves a purpose other than assuring the correct value for arguments of ! that are less than or equal to zero.
7.3.1 Unwanted Recursion

An improperly defined rule can rapidly fill Mortran's workspace or waste time in infinite recursion.

Space-filling example (7-15)

\[
\text{REPLACE(A)WITH(AA)}
\]

Rule (7-15) will replace the first "A" encountered in the input text with "AA", and then replace the first "A" of "AA" with "AA" resulting in "AAA", and so on until the remaining workspace is filled. Then Mortran will issue the message "MORTTRAN WORKSPACE EXHAUSTED", and stop.

For those cases where one actually wants this type of rule, one should use the primitive EMIT (see appendix A). In the case cited above, one should write:

\[
\text{REPLACE (A) WITH ((EMIT AA))}
\]

Time-wasting example (7-16)

\[
\text{REPLACE(A)WITH(A)}
\]

Example (7-16) will repeatedly replace "A" with "A" until the replacement limit (default 100) is reached. Then Mortran will issue the error message "100 RULES EVALUATED WITH NO OUTPUT", and turn on TRACE. TRACE is turned on so that the user may find the unwanted recursion without re-submitting the job. (See appendix A for a description of TRACE.)
7.6 THE SCOPE OF RULES

A rule is global\(^{11}\) to the text following its definition unless the scope is restricted by the processor directives\(^{12}\) !LOCAL and !END LOCAL.

The !LOCAL directive (in effect) marks all subsequent rules as "temporary". The !END LOCAL directive deletes all "temporary" rules, and recovers the space they occupied.

This feature is useful in a number of ways, among them:

* Rules may defined that apply only within a SUBROUTINE or FUNCTION subprogram.

* Because rule definitions may be nested, the creation and elimination of rules that "live" only to carry out some temporary need is facilitated.

* In the current version !LOCAL directives may not be nested. (This will be corrected in the next version.)

---

\(^{11}\) Applies everywhere.

\(^{12}\) See appendix D for a list of directives.
7.7 REMOVING RULES AND SPECIFIERS

There are two ways of removing Rules and Specifiers:
1) By defining them within a LOCAL range as described above.
2) By the statements:

```
REMOVE RULE 'template'
REMOVE SPECIFIER 'specifier'
```

The recovery of the space left by REMOVED rules or specifiers is automatic in the LOCAL case, but is not automatic for explicit REMOVE statements. The user can force the recovery of "dead" space by the directive !RECOVER. In either case, the recovery procedure prints the following:

```
nnn  LOCATIONS RECAINED BY RECOVERY
mmm  LOCATIONS REMAINING
pp   PERCENT RULE CAPACITY REMAINING
```

Recovery is also automatically invoked whenever the Rule and Specifier storage space is exhausted.
7.8 INTERNAL COUNTERS

There are 35 user-accessible counters, 1...9, and A...Z.

Each of these counters can hold one integer.

The user may access these counters with the following primitives:

(INCR c)  increment counter c, c=(1...9,A...Z)
(DECR c)  decrement counter c, c=(1...9,A...Z)
(SETR c=a) set counter c to the value of expression a.
(COPY c)  returns value of c (base 10) as a digit string.
(COPY c BASE b) returns value of c (base b) as a digit string.

The expression e must be reducible to a constant integer expression. A simple example will illustrate the uses of COPY.

```
SPECIFY NUM AS (1,0)[DIGIT];

(7-17) REPLACE [SHOW DIGITS IN (ARB) BASE (NUM)]
       WITH ( (SETR A=(P1))
              DIGITS OF (P1) BASE 10 ARE: (COPY A);
              DIGITS OF (P1) BASE (P2) ARE: (COPY A BASE (P2));
        )
```

Examples of the application of (7-17) follow:

The text: SHOW DIGITS IN 2**8-1 BASE 2;
          yields: DIGITS OF 2**8-1 BASE 10 ARE: 255
          yields: DIGITS OF 2**8-1 BASE 2 ARE: 11111111

The text: SHOW DIGITS IN 8**3-1 BASE 8;
          yields: DIGITS OF 8**3-1 BASE 10 ARE: 511
          yields: DIGITS OF 8**3-1 BASE 8 ARE: 777

The text: SHOW DIGITS IN 8**4-1 BASE 16;
          yields: DIGITS OF 8**4-1 BASE 10 ARE: 4095
          yields: DIGITS OF 8**4-1 BASE 16 ARE: FFF

The text: SHOW DIGITS IN 8**9/2+4 BASE 36;
          yields: DIGITS OF 8**9/2+4 BASE 10 ARE: 40
          yields: DIGITS OF 8**9/2+4 BASE 36 ARE: 14

The text: SHOW DIGITS IN 220 BASE 22;
          yields: DIGITS OF 220 BASE 10 ARE: 220
          yields: DIGITS OF 220 BASE 22 ARE: A0
7.9 **THE LENGTH PRIMITIVE**

The LENGTH primitive is written

\[ (\text{LENGTH } r) \]

where \( r \) is a parameter reference number.

A parameter reference number is a number base 35 denoted by \( 1 \) through \( Z \). If there are fewer than \( r \) parameter markers, the error message

NOSUCH ARG. IN TEMPLATE OF RULE

will result.

We give three examples of its use:

- SPECIFY WORD AS \((1,11)[\text{LETTER}]\);
  
  REPLACE \((\text{COUNT(WORD)})\) WITH \(((P/I)\text{ HAS }\text{LENGTH }1)\text{ LETTERS})\)

  Example: COUNT THIS yields: THIS HAS 4 LETTERS

- SPECIFY NAME AS \((1,4)[\text{LETTER}]\);
  
  REPLACE \((|\text{NAME}|)\)
  
  WITH \(\{\text{IF} (\text{LENGTH }I)=2 [\text{ABS}((P/I))]\text{ IF} (\text{LENGTH }I)=3 [\text{LENGTH}((P/I))]\text{ IF} (\text{LENGTH }I)=4 [\text{DETERM}((P/I))]\}\)

  Example: \(I|XX| + J|YYY| + K|ZZZ|\)
  
  yields: \(I\text{ABS}(XX) + J\text{LENGTH}(YYY) + K\text{DETRM}(ZZZ)\)
The third example is an extension of the COPY example from the previous chapter.

```
SPECIFY NUM AS (1,8)[DIGIT];
REPLACE (SIZE([ARB])) WITH ( LENGTH 1 )
REPLACE (SHOW DIGITS IN [NUM] BASE [NUM])
WITH ( (SETR A=(P1))
    THE SIZE((COPY A BASE (P2)))
    DIGITS OF (P1)((P2)) ARE:
    (COPY A BASE (P2))
)
```

The text: SHOW DIGITS IN 255 BASE 2
yields: THE 8 DIGITS OF 255(2) ARE: 11111111

The text: SHOW DIGITS IN 511 BASE 8
yields: THE 3 DIGITS OF 511(8) ARE: 777

The point of this example is that the ancillary rule SIZE is necessary because the argument of LENGTH is a parameter reference number. In other words an instance of the template SIZE((ARB)) must be recognized in order to determine the length of its argument.
7.18 APPENDING TEXT.

Text may be appended to the value part of an existing rule. There are two forms:

(7-18) \texttt{APPEND (value) TO (template)}

and

(7-19) \texttt{APPEND 'value' TO 'template'}

The two forms differ in one respect only: the latter preserves ALL blanks in its template part.

APPEND can produce two effects:

1) It can append value to the value part of a previously defined rule whose template part is identical to template.

2) It can define a new rule if no rule with a template identical to template exist. In other words, it defaults to:

\texttt{REPLACE (template) WITH (value)}

NOTE: In searching for a rule with template identical to template, the entire template (including parameter markers) is compared, as opposed to normal rule application. In other words, in this case parameter markers are treated as if they were literals.

For example, consider the following:

(7-20) 1 \texttt{APPEND (FOUR) TO (LINCOLN)}

2 \texttt{APPEND (SCORE AND SEVEN) TO (LINCOLN)}

Assuming that no rule whose template is LINCOLN exists, then

(7-20)-1 Defines a rule whose template is LINCOLN and whose value is FOUR.

(7-20)-2 Appends SCORE AND SEVEN to FOUR, so that the value of LINCOLN becomes FOURSCORE AND SEVEN.

Notice that leading and trailing blanks have been deleted and that embedded blanks have been reduced to a single blank in the appended text. This is done so that the value part will conform to the "canonical" form for rule definition.
If it is desired that blanks be preserved in the appended text, the alternate form of APPEND should be used. Using the alternate form with the same example:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>APPEND 'FOUR' TO 'LINCOLN'</td>
</tr>
<tr>
<td>(7-21) 2</td>
<td>APPEND 'SCORE AND SEVEN' TO 'LINCOLN'</td>
</tr>
<tr>
<td>3</td>
<td>APPEND 'YEARS AGO' TO 'LINCOLN'</td>
</tr>
</tbody>
</table>

Again, assuming that no rule whose template is LINCOLN exists:

If one denotes a blank by a lower case "b", then the final value of LINCOLN produced by (7-21) would be:

```
FOURSCORE and bSEVEN bYEARS bbb AGO b
```

or

```
FOURSCORE AND SEVEN YEARS AGO
```

In summary:

- Text may be APPENDED to a previously DEFINED rule.
- The alternate form may be used after the "standard" form, and vice versa.
7.11 SPECIAL SYMBOLS

7.12 THE RESERVED SYMBOL

One symbol is reserved for "system" use and may not be used by the programmer in any context whatever, save one: it may be changed. The default setting for the reserved symbol is "a". If this symbol must be used for other purposes, (e.g. the operating system demands its use), the reserved symbol may be changed. The reserved symbol may be set by the processor directive

!SETRRESERVED (where "r" denotes an ASCII symbol.)

For example, !SETRRESERVED would cause the symbol "a" to become the reserved symbol. Subsequent to !SETRRESERVED, the symbol "a" would become available for other purposes.

7.13 OTHER "SPECIAL" SYMBOLS

In addition to the reserved symbol, Mortran3 recognizes a number of symbols (in some context) as "special". There are three categories:

a) Delimiters:
   Parentheses ------ ()
   Braces ------------ {}  
   Brackets --------- []  
   Quotation mark --- "
   Apostrophe

b) Two directive indicators:
   Bang -------------- !
   Percent ----------- %

c) One terminator:
   Semicolon --------- ;

13 The apostrophe may also be treated as an "ordinary" symbol. Mortran3's treatment of the apostrophe depends upon a toggle that may be set by the !STRING processor directive. See appendix A and also the subsection "The Apostrophe" below.
7.13.1 Terminators and Delimiters

We will use the terms delimiter and terminator in the following sense:

- A terminator marks the end of an object
- Delimiters mark both the beginning and the end of an object.

For example, an exclamation point terminates this sentence! In contrast, a remark (like this) is often delimited by parentheses. But a remark, like this, may also be delimited by commas. In written English, a "parenthetical remark" is not necessarily a remark within parentheses; it may be a remark set off by commas.

Some delimiters (like parentheses) employ separate, symmetric symbols\(^\text{14}\) to mark the beginning and end of the delimited text. The FORTRAN "language" uses parentheses as delimiters in many contexts, but makes little use of terminators. The FORTRAN "END statement" terminates (sub)programs. But statements are not explicitly terminated. Rather, they are implicitly terminated at the end of a line unless the following line is a "continuation line".

Mortran3 "language statements" are terminated by semicolons as described in Part One.

In contrast, REPLACE(template)WITH(value) is not a statement; it is a function definition. It is not terminated by a semicolon, because both the template and value parts may contain semicolons. Both the template and value parts are delimited by braces; consequently neither may contain unbalanced braces.

7.13.1.1 The Apostrophe

The apostrophe is not the same (conceptually) as a quotation mark. But most FORTRAN compilers have been extended\(^\text{15}\) to include the apostrophe as a "legal"

---

\(^{14}\) The French language employs the symbols "\(<\)" and "\(>\)" to delimit quotations. Typesetters in most languages employ separate, symmetric symbols to delimit quotations. But "programming languages" do not. One reason is that the most widely used character sets, EBCDIC and ASCII, do not define "left quotation mark" and "right quotation mark".

\(^{15}\) The 1966 FORTRAN standard does not include the apostrophe in its character set and does not define a "string" or a "literal" data type. In section 4.2.6, the standard says: "Hollerith Type. A Hollerith datum is a string of characters. This string may consist of any characters capable of representation by the processor. The blank character is a valid and significant character in a Hollerith datum."
character, and they employ the apostrophe as a delimiter for "literal" data or "strings". MortranJ's predecessor, Mortran2, delimited its "patterns" (templates) with apostrophes. And thereto hangs a tale of horror. To illustrate, consider the following form of the Universal Truth:

It don't matter!

We may write this in a FORMAT statement using the (extended) FORTRAN convention, as:

'It don't matter!'

Here, literal data is delimited by apostrophes, and a single apostrophe is denoted by a pair of apostrophes. But suppose we want to quote ol’ Joe as having said: "It don't matter!".

'O1' Joe said: '"It don’t matter!''

Here, a single apostrophe delimits the outer quotation and a pair of apostrophes denote a single apostrophe and also delimit the inner quotation, while four apostrophes denote an apostrophe within the inner quotation. If we quote ol’ Pete quoting ol’ Joe, we get:

'O1' Pete said: '"O1' Joe said: '"'It don’'t matter!''''

Using symmetric "French quotes" we could re-write the latter as:

<Ol' Pete said: <Ol' Joe said: <It don't matter!>>>

MORAL: Usin' a single symbol as a delimiter ain't a 'specially good practice.16

16 FORTRAN delimits logical and relational operators with ".". This does not improve the readability of FORTRAN predicates. Compare (./A.<.2) to (A.GT..1.AND..2.GT.A).

A. James Cook
- 60 -
December 16, 1982
Mortran2 fell into the trap illustrated by the above examples. Mortran2 "patterns" (templates) were delimited by apostrophes, following the extended FORTRAN convention for strings.

Mortran3 has eliminated this horror to a large extent, but vestiges remain. In particular, Mortran3 must still accept and process FORMAT statements using this convention. Moreover, Mortran3 must permit user-defined transformation rules that generate FORTRAN-compatible data for FORMAT statements.

Mortran3's interpretation of the apostrophe depends upon a toggle that may be set by the programmer. The toggle is turned "ON" by the processor directive "!STRINGS;" and is turned "OFF" by the processor directive "!NOSTRINGS;". The default setting is "ON", which means that the apostrophe is a delimiter of strings. "OFF" means that the apostrophe is treated like any other symbol.
Appendix A

PROCESSOR CONTROL

There are two types of processor control directives:

- Free-form
- Column-one-restricted.

Free-form directives may appear anywhere on a line. Any number of Free-form directives may appear on a line.

Column-one-restricted directives MUST begin in column one and only one directive per line is recognized. (Column-one-restricted directives are a hold-over from Mortran2, and will be eliminated in future versions.)

A.1 FREE-FORM DIRECTIVES.

!ANNOTATE; Interleave Mortran source in FORTRAN output.
Mortran statements become COMMENTs in FORTRAN output.

!NOANNOTATE; Turn off !ANNOTATE.

!COMMENTS; Print Mortran comments as FORTRAN comments.
Mortran comments are output to FORTRAN file with 'C' in column one.

!NOCOMMENTS; Turn off !COMMENTS.

!LIST; Turn on Mortran listing

!NOLIST; Turn off !LIST.

!DEFINE; Print out rules at point of definition.

!NODEFIN; Turn off !DEFINE.

!LABELS n; Reset FORTRAN statement label generator to n.
!LOCAL; Begin "local" rule definitions. Local rules apply
to the text between the !LOCAL and !END LOCAL
directives.

!END LOCAL; Remove all rules defined after BEGIN LOCAL,
and recover space.

!RECOVER Recover workspace occupied by "dead" rules.

!SEQUENCE n; n=1, Put the MortranJ source from columns L-8 thru L
into columns 73 thru 80 of the FORTRAN text.
(where L is the line length.)

n=2, Put MortranJ generated sequence numbers
into columns 73 thru 80 of the FORTRAN text.

n=(anything else), Put blanks
into columns 73 thru 80 of the FORTRAN text.

!TRACE n; n=2 print template and parameters
each time a rule is applied,
NOT including nested applications.

n=3 print template and parameters
each time a rule is applied,
INCLUDING nested applications.

n=6 print template and parameters
each time a rule is ATTEMPTED,
INCLUDING nested applications.
(Produces VOLUMINOUS output. Not normally used.)
A.2 COLUMN-ONE-RESTRICTED DIRECTIVES.

%An For n=1 same as !ANNOTATE; above
  n=2 same as !NOANNOTATE; above

%Bn Set input buffer size to \( 80 < n < 132 \)

%Cn Set "significant column" to \( 10 < n < \text{buffersize} \)

%Dn For n=1 same as !DEFINE; above
  n=2 same as !NDEFINE; above

%F Set FORTRAN mode

%In Set automatic indentation to n columns per nesting level.

%L same as !LIST; above

%M Return to Morran mode

%N same as !NLIST; above

%Q Terminate comments at end-of-line.

%Rn Rewind Unit n

%Sxn Same as !SEQUENCE n; above.

%Tn Same as !TRACE n; above

%Un Switch to FORTRAN logical unit number n for MorranJ input
Appendix B

PRIMITIVE SPECIFICATIONS.

There are four types of primitive specifications:

- QUALIFIERS
- QUANTIFIERS
- ALTERNATIVES
- LITERALS

These primitives provide the interface to the text containing user-defined notations. The Mortran processor defines a "recognition function" for each user-defined SPECIFY statement. The specifier part becomes the name of the function. When a specifier is used as a parameter marker in a REPLACE statement, the recognition function is invoked. Therefore, ALL user-defined recognition functions (specifiers) must (ultimately) be resolvable in terms of these four primitives.

B.1 QUALIFIERS

\[(S_1 \ldots S_2)\]

where: \(S_1 \) and \(S_2\) are symbols,
and \(S_1 \leq S_2\)

This primitive\(^\text{17}\) qualifies a single text symbol. Let \(T\) be a text symbol. Then,

\[(S_1 \ldots S_2)\] returns \(\text{set}\) if \(S_1 \leq T \leq S_2\) and \(\text{not-set}\) otherwise.

\(^{17}\) This primitive depends upon Mortran's internal representation of symbols. In the present context it suffices to note that \(0 < 1 \ldots < 9 < A < B \ldots < Y < Z\). See Appendix C.
B.2 QUANTIFIERS.

\[(\text{min, max})[\text{specification}]\]
\[(\text{exact})[\text{specification}]\]

\(\text{min, max, and exact must be reducible to constant expressions.}\)^\[18\]

- Quantifiers are enclosed in parentheses.
- The specifications to which they apply are enclosed in square brackets.
- \((\text{min, max})\) returns \textit{set} for no fewer than \textit{min} nor more than \textit{max} occurrences.
- \((\text{exact})\) returns \textit{set} for exactly \textit{exact} occurrences.

---

\[18\] See the Subsection "Constant Expressions".

A. James Cook - 66 - December 16, 1982
B.3 **ALTERNATIVES**

A list of alternatives is written:

```
[ A1 | A2 | ... | An ]
```

Where the Ai are alternative specifications.

- The entire list returns *set* or *not-set*.
- The alternatives are examined left-to-right.
- If an alternative returns *set*, the remaining alternatives are skipped.
- If the list is exhausted, *not-set* is returned.
- The number of alternatives is not limited.
- An alternative may not be NULL (empty), but it may be the empty string: `.`. An entire list of alternatives may be specified as "optional" by including the empty string as the last alternative. Because the empty string is always satisfied, it must be the last of the alternatives.
- A list of alternatives may quantified.
  
  * In particular, \((0,1)[A1|A2|...|An]\)
  
  is equivalent to \([A1|A2|...|An|\"")

- An unquantified list of alternatives:
  
  \([...|...|...]\)

  is equivalent to \((1)[...|...|...]\)
B.4 LITERALS

LITERALS in specifications are written:

```
'string'
```

where string is an arbitrary symbol-string.

Within string, a single apostrophe must be represented by a pair of apostrophes.

Literals in specifications should not be confused with literal symbols in rule definitions.

Literal symbols in rule definitions (REPLACE statements) are NOT enclosed in apostrophes.

- In a rule that does not contain apostrophes:
  - In the template part, all symbols not enclosed in braces are literal symbols.

- In a rule containing apostrophes,

  (1) There must be an even number of apostrophes.
  (2) The doubling rule applies for embedded apostrophes.
  (3) In the template part the apostrophes must match the text.
  (4) In the value part the apostrophes are generated.

In summary:

In specifications a string denotes a sequence of literal symbols.

In rule definitions a string denotes a literal in the FORTRAN sense.
Appendix C

PRIMITIVE EVALUATORS

All primitive evaluation function references\textsuperscript{19} are denoted by:

\[
(n\text{ame }\text{arg})
\]

where \texttt{name} is the name of the function and \texttt{arg} is an arbitrary string of symbols.

Primitive evaluation function references may be nested. The "depth" of the nesting is limited only by the amount of "work-space" available to Mortran. (In other words, there is no fixed limit on the nesting level.)

Evaluation is from the innermost of the nest outward.

\textbf{C.1 REDUCE}

\{(REDUCE arg)\} Recursively apply all (system and user) rules to the argument until no rule is applicable.

\textbf{C.2 COMPUTE}

\{(COMPUTE arg)\} Invoke REDUCE.

REDUCE\texttt{d} argument should be constant expression.
- If it is, returns value of constant expression.
- If it is not, issues error message and returns 0.

\textsuperscript{19} Except the evaluation-time IF-ELSE.
C.3 COUNTER PRIMITIVES

There are 35 user-accessible counters, 1...9, and A...Z.

Each of these counters can hold one binary integer.

The following primitives refer to these counters.

(IMCR c) Increment counter c, c=(1...9,A...Z)
(DECR c) Decrement "
(SETR c=e) Set counter c to the value of expression e.
(COPY c) Returns value of c (base 10) as a digit string.
(COPY c BASE b) Returns value of c (base b) as a digit string.

C.4 PARAMETER PRIMITIVES

(Pn) Returns the nth actual parameter.

(LENGTH n) Returns the number of symbols in actual parameter n.

(EXIST n) Returns 1 (true) if actual parameter n is not NULL\(^{20}\)
          Returns 0 (false) if actual parameter n is NULL

(NULL n) Returns 1 (true) if actual parameter n is NULL
         Returns 0 (false) if actual parameter n is not NULL

---

\(^{20}\) Recall that a reference (Pn) may be NULL if the nth parameter marker has been specified as optional.

A. James Cook

December 16, 1982
C.5 MISCELLANEOUS PRIMITIVES.

(WAIT arg) Returns arg verbatim.

This primitive may be used to override the 'innermost-outward' order of evaluation.

This primitive delays, for one pass, the application of embedded rules. It does not prevent application.

(EMIT (arg)) Prevents application of any rule to arg.

(Appplies to system and user-defined rules.)

Useful for issuing error messages containing function references.

Also useful for preventing unwanted recursion.

The WAIT primitive delays application; the EMIT primitive prevents application.