

MORTRAN2, A MACRO-BASED STRUCTURED FORTRAN EXTENSION

A. James Cook and L. J. Shustek
Computation Research Group
Stanford Linear Accelerator Center
Stanford University, P. O. Box 4349, Stanford, California 94305

ABSTRACT

A language that permits a relatively easy transition from FORTRAN programming to structured programming is described. This language, called MORTRAN2, is a FORTRAN language extension that is further extensible by user-defined macros. The language is implemented by a pre-processor (which is in fact a macro-processor) written in standard FORTRAN. The output of the pre-processor is also FORTRAN, so that transportability of both the pre-processor and its generated programs is assured.

INTRODUCTION

The computing world is awash with a new fad: Structured Programming. There is no consensus as to what structured programming is but some of its more zealous advocates assert that those who write non-structured programs are at best in a state of sin and at worst guilty of criminal obfuscation. It has been said that it is impossible to write a "good" program in FORTRAN. To those who have written the overwhelming majority of scientific and engineering programs in FORTRAN over the last fifteen years, these pronouncements are a trifle exaggerated. There are several reasons why FORTRAN has not met the sudden and awful death that its detractors insist it deserves. Among the reasons are

- Availability. Compilers exist for most computers, even many "mini" computers.
- Efficiency. Many of the FORTRAN compilers are peerless in producing efficient machine code.
- Universality. It is the most widely known scientific and engineering language.

Admittedly, FORTRAN has some weak points. A number of FORTRAN pre-processors have been written to provide FORTRAN language extensions which correct some of these weaknesses, but they are lacking in one or more of the following respects: transportability (of the pre-processor and its output), extensibility (of the language), or compatibility (with existing FORTRAN programs). In addition to these technical constraints, we impose what might be called an aesthetic requirement: the language should support the development of programs that are easy to read and to modify. MORTRAN2 satisfies the technical constraints, and provides facilities that permit programs to be written that satisfy the aesthetic requirement. The macro facility in MORTRAN2 provides the programmer with a means of defining symbols to represent a sequence of operations in a way that augments the sub-program facility provided by FORTRAN. The macro facility may also be used to extend the language (for example, to include new data structures, and new operators on those data structures).

We will show that by successively eliminating some of the objectionable features of FORTRAN one is led easily and naturally to a language that permits structured programs to be written, debugged, and maintained.

MORTRAN2

We will dispose of FORTRAN's small annoyances first. Counting the number of characters in a Hollerith data field is error-prone and should be unnecessary. The rule in MORTRAN2 is

- Enclose Hollerith data in apostrophes (as in 'PARAMETERS'). If an imbedded apostrophe is needed, represent it by a pair of apostrophes (as in 'DON'T' which becomes SHDON'T).

FORTRAN has some annoying rules about "column seven" and "continuation marks". Forget all such rules, and substitute the single rule:

- Terminate all statements with a semicolon (as in X=Y;), and ignore column and card boundaries.

FORTRAN rules regarding comments do not allow the comments to be placed where they would be most meaningful. Put comments anywhere you like, but

- Enclose all comments in quotation marks (as in "COMMENT").

FORTRAN statement labels have no mnemonic value. We abolish them and use alphanumeric labels instead. A label in MORTRAN2 is an alphanumeric character string of arbitrary length enclosed in colons (as in :TOMATOES:). In MORTRAN2,

- Use an alphanumeric label (in any context) where you would ordinarily use a FORTRAN statement label.

So much for eliminating the minor annoyances. (The FORTRAN programmer need learn only the above four rules in order to start writing programs in MORTRAN2, so that gradual transition from FORTRAN to MORTRAN programming is possible.) We will now consider FORTRAN's more serious deficiencies:

*Work supported by the U. S. Atomic Energy Commission.

FORTRAN has no "block structure", and "structure" is the thing we are trying to achieve. Let us define a block as a sequence of statements which are delimited in some manner, and see how it can be used. Let

```
S1; S2; S3;...Sn; (1)
```

be a sequence of statements. The sequence becomes a block when it is enclosed in the special symbols < and > which we will call "brackets". The brackets are not meta-symbols; they are delimiters in the language. We make (1) into a block by writing

```
<S1; S2; S3;...Sn;> (2)
```

We can use such a block to circumvent the FORTRAN deficiency which disallows more than one statement following a logical IF statement. Let e be an arbitrary logical expression. If we write

```
IF e <S1; S2; S3;...Sn;>
```

all of the statements in the block are executed if and only if e is TRUE. If we write

```
IF e <T1; T2; T3;...Tn;>
ELSE <F1; F2; F3;...Fn;>
```

the statements T_i are executed if e is TRUE, and the statements F_i are executed if e is FALSE.

To save space we will write an ellipsis enclosed in brackets to denote a block. That is, we will write <...> instead of (2).

Another thing we can do with our definition of a block is to simplify the way we write loops. For example

```
WHILE e <...>
```

repetitively executes the statements in the block while e is TRUE, and

```
UNTIL e <...>
```

repetitively executes the statements in the block until e becomes TRUE.

```
DO i=j,k,n <...>
```

where i, j, k, and n are the standard FORTRAN DO parameters, generates a standard FORTRAN DO loop.

FORTRAN DO loops have certain deficiencies. The "control variable" must be a non-subscripted integer variable; the other DO parameters must be non-subscripted integer variables or unsigned INTEGER constants. All DO parameters must always be positive. In MORTRAN2 one may write

```
FOR v=a TO b BY c <...> (3)
```

where a, b, and c are arbitrary arithmetic expressions, and v is a variable of type REAL, INTEGER, or DOUBLE PRECISION which may be a simple variable or an array element (subscripted variable). We will call (3) a "FOR loop"; it has

two alternate forms

```
FOR v=a BY b TO c <...>
FOR v=a TO b <...> (4)
```

Since (4) has no increment, it is assumed to be one.

MORTRAN2 implements the "forever" or "endless" loop with

```
LOOP <...> REPEAT (5)
```

The optional word REPEAT in (5) is an aid to readability which may also be used with any of the other loops.

In MORTRAN2, a loop is simply a block which has been preceded by, and optionally followed by a "control phrase". The control phrases which begin a loop are WHILE... UNTIL... FOR..., DO..., and LOOP. The control phrases which may optionally follow a loop are WHILE..., UNTIL..., and REPEAT.

In order to "jump out" of a loop one may write something like

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

where the label :CHICAGO: precedes some statement or block that is outside the loop. This can be annoying if a convenient label does not already exist and one must be created for the sole purpose of leaving the loop. MORTRAN2 offers an alternative. The statement

```
EXIT;.
```

causes control to be transferred to the first executable statement following the loop in which the EXIT appears. The statement

```
NEXT;
```

causes control to be transferred to the next iteration of the loop in which the NEXT appears, incrementing the control variable (if any) before performing the test for continuation in the loop.

Any loop may be optionally preceded by a label. The EXIT and NEXT statements may be optionally followed by a label, in which case, the transfer of control is made with respect to the loop bearing that label. For example, suppose that the outermost loop of a nest of loops has been labeled :SEARCH:, and that two of the interior loops have been labeled :COLUMN: and :ROW:. We may write

```
NEXT :COLUMN: ; or NEXT :ROW: ;
```

to transfer control to the next iteration of the corresponding loop, or

```
EXIT :SEARCH: ;
```

to transfer control to the statement following the outermost loop.

USER-DEFINED MACROS

At any point in a MORTRAN2 program one may define a macro by writing

%'pattern'='replacement' (6)

where "pattern" and "replacement" are character strings optionally including parameters and imbedded character strings. (Since imbedded character strings are permitted, the rule regarding imbedded apostrophes must be observed when writing macros.)

In the simple (parameter-less) form, a macro does simple text substitution; all occurrences of the pattern string in the program are replaced by the replacement string.

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' (7)

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

(assuming (7) 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 has been successfully 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 formal parameters of the form #i (i=1,2,...,9). During the expansion process, 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

%'INCREMENT #;' = '#1=#1+1;' (8)

would match the program text

INCREMENT A(I,J,K);

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), the expansion of the macro takes place, during which the actual parameter "A(I,J,K)" replaces all occurrences of the corresponding formal parameter, and producing

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

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.

Space does not permit more elaborate examples demonstrating the full power of macros. One particularly useful feature is the ability to write macros which generate other macros. For this purpose, the rule regarding the doubling of an apostrophe to denote an imbedded apostrophe is extended to include the symbol # used to denote a formal parameter. That is, if the replacement part of a macro is (or contains) a macro definition, the formal parameters in the replacement part which are within the contained macro are represented by ##.

MORTRAN1

MORTRAN2 is an elaboration of MORTRAN1 which has been in use at the Stanford Linear Accelerator Center for almost two years. Some very large programs have been written in MORTRAN for the analysis of high energy physics data. Our experience to date shows that the time required to pre-process MORTRAN programs is roughly equal to the time required by the FORTRAN (IBM, level H) compiler to compile the resulting program. In both MORTRAN1 and MORTRAN2 the programmer may insert segments of FORTRAN code in a MORTRAN program or segments of MORTRAN code in a FORTRAN program.

OTHER APPROACHES

Modification of the "front end" of the FORTRAN compiler to accept MORTRAN programs seems attractive at first because of the obvious advantage of avoiding the pre-processor step with the implied increase in speed as well as other advantages. The disadvantages are: loss of transportability of the processor, loss of transportability of the programs which are output from the processor, vastly increased maintenance problems, introduction of new uncertainty with respect to suspected bugs in the compiler or (heaven forbid) the pre-processor.

The approach we have taken is to regard FORTRAN as a machine-independent "assembler language" (the only such language available today), and the pre-processor as a compiler whose object code is FORTRAN.

CHARACTER SETS AND TRANSPORTABILITY

There exists an incredible number of different character sets and internal representations for those character sets. MORTRAN2 solves the problem by reading in the character set during initialization. All that is necessary to change any of the special characters used by MORTRAN2 as delimiters is to change the characters on a single card. No reference is ever made in the processor to any particular character set or to any internal representation. Moreover, since the delimiters are matched by macro patterns, the user may use words as delimiters instead of special characters. For example, the word "BEGIN" may be used instead of the left bracket, and the word "END" may be used instead of the right bracket.

RESERVED WORDS VERSUS BRACKETED KEY-WORDS

MORTRAN2 offers the programmer a choice of writing programs using reserved words, or enclosing his keywords in brackets. Reserved words have the advantage that they are easily typed, but the disadvantage that they may not be (safely) used as variable names. Bracketing the keywords makes them easy to locate in the program listing and allows the programmer to use keywords as variable names if he wishes.

EXTENSIBILITY

The language described in this paper is defined by a set of about fifty macros (requiring about as many cards). In order to re-define the language, one need only write a new set of macros. For example, the choice of key-words or reserved words described above requires a very simple modification of the standard (language-defining) set of macros.

User-defined macros may be easily added to the standard set to extend the language. For example, macros have been written that permit simple operations on matrices so that explicit loops need not be written. More elaborate macros have been written that define dynamically allocated linked-list data structures. Some very useful macros have been written that might have been included as part of the language, but were not on the grounds that they are (1) available as an option and (2) would "clutter up" the language if they were made an integral part of it. The macros are "stacked", that is, the last read in is the first scanned for matches in the program text, so that the user may override specific parts of the standard macro set without re-writing all of it.

READABILITY

Some aids to readability have already been mentioned: alphanumeric labels, the fact that comments may be inserted anywhere in the program

text, and the fact that the language is "free-field". In addition MORTRAN automatically prints the nesting level, and optionally indents the listing according to the nesting level. Automatic indentation can be very helpful in exposing the structure of complex programs.

CONCLUSION

A structured language is one which has (minimally) a clearly defined nested block structure, and facilities for controlling the execution of those blocks. MORTRAN makes the only machine-independent language available today into such a structured language.