A Macro Preprocessor for a FORTRAN Dialect

John Gary

Department of Computer Science
University of Colorado
Boulder, Colorado
80302

August 1975

This is a revision of Report #CU-CS-054-74, August 1974.

This design was performed under ARPA Grant AFOS-74-2732.

0. Introduction. Our objective is to provide a macro preprocessor for a language similar to FORTRAN. This is a "mesh operator" language (PDELAN) intended for the construction of finite difference codes for partial differential equations. It is described in a companion report [8]. The PDELAN compiler generates a FORTRAN object program. These ideas have not been implemented at the time of this writing.

The macro preprocessor could be used with a modified version of FORTRAN in which blanks are delimiters and certain "keywords" such as IF, FORMAT, DO, etc., are also reserved words which cannot be used as names by the user. The syntax of the macro preprocessor is intended to be natural to a FORTRAN programmer. Its most frequent application, such as the propagation of COMMON declarations throughout subroutines, should be easy to remember and use. It would place the error messages from the PDELAN compiler in the original source code. PDELAN contains structured control statements such as the following [13]:

IF ... THEN ... ELSE ... ENDIF

REPEAT ... UNTIL ... ENDREPEAT

The macro preprocessor permits long names (up to 29 characters) which are shortened to 6 characters on output (with name conflicts avoided). It also permits, through conditional macro expansion, the generation of code which is more machine independent. It is not intended to allow user defined language extension, except trivial extensions. It seems to us to be too difficult to include good error diagnostics in a macro extension. Also the macro extensions are probably too slow. Thus we place the processor for the mesh language in a compiler which follows the macro preprocessor. This compiler for the PDELAN language is described elsewhere [8].

This is a continuation of work started at NCAR in 1970. A first version of PDELAN for solving partial differential equations was developed by Gary and Helgason [1]. A set of graphics commands was added to PDELAN by Löcs and Gary [2]. This code is in light use at NCAR (fewer than 10 people use it). However Helgason wrote an improved version of the macro preprocessor called FRED which has been further improved by Dave Kennison [5]. FRED contains macro capability, subscripts bounds checking, a "TIDY" feature to renumber and indent a deck, and other features. However, it does not contain difference operators or graphics commands. Here we propose a macro facility somewhat different from FRED. It will not have the subscript bounds checking or the TIDY feature. It will have the capability to modify and generate tokens, perform conditional macro expansion, and execute macro-time expressions. It will be based on the PDELAN language in which blanks are delimiters and the keywords are reserved. PDELAN contains many FORTRAN features such as COMMON, SUBROUTINE, the same type of data structures (or lack of them), and the same I/O structure. However, it does have structured conditional and repetitive statements such as the PASCAL language or the preprocessors RATFOR, FLECS, MORTRAN, IFTRAN [14].

One of our main objectives is to make the macro syntax appear natural to an experienced FORTRAN programmer. Thus the macro definitions are similar to subroutines and the macro calls are similar to a FORTRAN function call. The macro-time statements and conditional compilation are similar to FORTRAN although we did use the IF ... THEN ... ELSE

construction. We feel our macro preprocessor is easier for a FORTRAN programmer to learn and remember than the macro preprocessor for FORTRAN called MP/1 by Macleod [6] or the preprocessor imbedded in the LLL FORTRAN [3]. MP/1 is more powerful since it is a pattern matching macro and it has a good set of macro-time commands. The LLL macros do not have much macro-time capability. The macro processors designed for FORTRAN (FLECS, MORTRAN, IFTRAN, RATFOR) that we are familiar with do not have enough macro-time commands or sufficiently flexible macros.

1. The syntax. In order to simplify the macro preprocessor a modified FORTRAN syntax is used in which blanks are delimiters and the keywords are reserved. Tokens can be recognized by a lexical scan ahead of the macro expansion and thus also ahead of the syntactical analysis. Macro calls are recognized by the appearance of a macro name with no special delimiting character or pattern matching required. The syntax of a macro definition is similar to that of a FORTRAN subroutine or function. The macro definitions can be stored as a string of tokens rather than a character string which should permit a more compact internal representation. The token types include integer and real constants, identifiers, and the operators or delimiters + - * / ** .. : ( ) =. The delimiters $ # and ' are also used. The first two are used in conjunction with the macros, the last is used to delimit character strings. The last three characters have a different form on the keypunch than the teletype form given here. Various period delimited operators are also used (.LE. .LT. .GT. .NE. .EQ. .AND. .OR. .NOT. .NULL.). Boolean constants (10B) and Hollerith constants ("ABCD" 4HABCD) are also used. Format specifications must be included as tokens (for example, 2E10.3,2P,2PE10.3). A nonblank character in column six denotes a continue card, which is almost the standard FORTRAN convention. Columns 1 thru 5 can be used only for statement labels.

In order to use a compiler writing system such as that described by Cohen [15], we might allow only identifiers, integers and character delimiters as tokens. These tokens can be recognized by a lexical scanner built into the compiler writing system.

The statements are almost "free-form". They may start anywhere on the card after column six. Column six is used to mark continuation cards. Only statement labels may appear in Columns 2 thru 5. The statement termination ";" can be used to place more than one statement on a card. The added statements can be labeled. If the first token of the statement is an integer constant, then this constant is a statement label. In order to produce better error diagnostics, it was decided not to attempt to make the macro preprocessor independent of the langage syntax. It is also an advantage if the macro preprocessor can recognize statement labels. Eventually we will even partially parse the input ahead of the macro explansion in order to indent the input source program according to the nesting level of the source statements.

2. The macros and macro-time statements. All the "macro-time" statements are preceded by a name which starts with the delimiter "%" or by the delimiter "%" itself. The character used for this delimiter is system dependent, it does not exit on the KRONOS timeshare system. These delimiters can be easily altered. There are two types of macro, a statement macro and an expression macro. The body of the statement macro consists of one or more statements. Such a macro name can be used as a statement in the PDELAN language. The macro name is an abbreviation for the macro body which replaces it prior to compilation of the PDELAN program. For example, consider the following abbreviation of a subprogram header block.

```
%MACRO  COMMONBLKA
     COMMON A,E,C,D,U
     COMMON /B/ H,DLX,DLY
     INTEGER HX,UX
%ENDMACRO
```

The body of the second type of macro is an expression. Both types of macro may have arguments. The following is an example of an expression macro definition and usage.

The definition is

```
%EXPMACRO FN(F,X) = EXP(F(X))
```

An example of the usable is

```
T = FN(SIN,W+1.)
```

The second statement would expand into

```
T + EXP(SIN(W+1.))
```

The definition of this type of macro is a single statement and therefore does not require the %ENDMACRO statement. More than one expression macro may be defined in a single statement. For example

```
%EXPMACRO UT = U(N1),  U2 = U(N2)
```

Macro calls may be nested, that is a macro definition may contain a call to a macro. Recursion is allowed, that is a macro may call itself. However, a macro definition may not contain another macro definition. A statement number may be placed on a statement macro call, for example

```
100  INITIALIZE
```

In this case a CONTINUE statement is generated ahead of the body of the macro INITIALIZE.

The scope of the macro definitions. When a macro definition is encountered the name of the macro is placed in the symbol table and its body is placed on a "definition stack." To avoid overflow of this stack it is possible to limit the scope of a macro definition. The command %MACROBLOCK indicates the start of a "block." All

macros defined within the block are regarded as local to the block. When the command %ENDMACROBLOCK is encountered all these local macros will be removed from the symbol table and their definitions popped from the definition stack. Macros not within such a block are regarded as global and cannot be removed from the definition stack. In addition, macros defined within a subprogram are available only within that subprogram. The %MACROBLOCK and %ENDMACROBLOCK commands cannot occur within a subprogram.

Macro formal parameters are local to the definition and can be used as names elsewhere in the source deck. A macro definition must appear ahead of its first usage or call.

3. Macro-time statements. These are declarations, expressions and the conditional control of macro expansion. These statements are executed at "macro-time," that is, when the expansion is performed. In this first version only integer variables can be declared at macro-time, and these variables must be global, they cannot be declared within a macro definition or a macro block. An example of a macro-time declaration of an integer variable is the following

```
%INTEGER  N,M
```

There must not be a blank following the %, it is part of the identifier.

Expressions involving the operators + - * / and integer variables are allowed. For example

```
%   M = (N+1)/2
```

The % alone denotes a macro-time replacement statement.

Conditional expansion. A Boolean expression can be evaluated at macro-time to control macro expansion. For example

```
%IF  N .LT. 0 THEN
     PRINT 250, X,Y,Z
     KC = KC+1
%ENDIF
```

If the Boolean is true then these two statements will be included in the code to be compiled, otherwise they will not. Note that THEN is a reserved word. The %IF THEN %ELSE %ENDIF compound statement must not overlap a macro definition, it must be completely within or completely outside a macro definition.

The Boolean expression can involve relational operations on macro-time integer expressions. Comparisons(.EQ. or .NE.) between character strings are allowed. For example

```
%MACRO  M(X)
   %IF X .EQ. A THEN CALL SUB(A): %ENDIF
.....
%ENDMACRO
```

If the actual argument of a macro call of this macro M is the token A, then the code

```
CALL SUB(A) ;
```

will be added to the output. These conditional statements can be nested. Note that %ELSE and %ENDIF must be used and not ELSE or ENDIF.

4. Generated symbols. Statement labels distinct from any used in the source code may be generated by use of the symbols #L(3) or #L(N) where N is a macro-time integer variable. Within each macro call #L(3) represents the same statement label different from the label generated by #L(3) on other calls of this macro. The label generated by #L(N) will depend on the value of N within a given call of the macro, and

will also be different on different calls of the macro. As an example consider

```
%MACRO SUM(S,A,N)
   S=0.
   DO #L(1) K-1,N
   #L(1) S=S+A(K)
   %ENDMACRO
```

A generated name, distinct from other names in the program, which starts with "I" is obtained from #I(1) or #I(N). For names starting with "E" the symbols #E(1) or #E(N) are included. These can be used to generate "local" variables within a macro call.

In addition the macro preprocessor will shorten long identifiers to 6 characters. This will be done by truncation provided no conflicts arise, otherwise the last character or characters will be modified until a unique name is obtained. This modification is carried out for each subprogram. Therefore external names should not exceed 6 characters.

5. Commands to control expansion. These are identifiers whose first character is %. They constitute a complete statement in PDELAN. Not all of these will be implemented initially.

%NOLIST - cease listing source code

%LIST - list source code

%NEWPAGE OR
%NEWPAGE(ID) - skip source listing to new page, Print identifier ID on top of page.

%LONGNAMES - at end of preprocessor source listing print longnames and their shortened form

%NAMEMAP - list line numbers where each identifier is used

%FINUS - end of source check

%MACROMAP - list line numbers where macros are definied or used.

%MACROBLOCK - indicate start of a block of macro definitions

%ENDMACROBLOCK - indicates the end of a block of macros. When this command is encountered the macros in the block will be popped from the macro definition stack. Therefore macros within the block are not defined following this card. This permits the use of "local" macros and may prevent overflow of the macro stack.

6. Error diagnostics. When an error is detected we will print the error message with the listing of the original source. If the error is inside a nested set of macro calls, then the error message will be printed after the innermost macro call. However, the name of each macro called within the nested set will be printed along with the line number from which it was called. We will try to print the innermost line of code along with a pointer to the token at which the scanner stopped, and of course a message to indicate the type of error. The error recovery will try to resume at the next statement inside the innermost macro. We are using recursive descent to parse the PDELAN variant of FORTRAN, and we may have some difficulty achieving good error recovery.

Note that good error messages and recovery probably requires the compiler which follows the macro expansion to be designed together with the macro preprocessor. Errors which the compiler finds should probably be printed in the original source code which is input to the macro preprocessor and not in the input to the compiler which is the output from the macro preprocessor. We regard the output of good error messages in this context as a difficult problem. Note that our mesh operator variant of FORTRAN (PDELAN) produces FORTRAN code as output which must then be input to a FORTRAN compiler. The code passed to the FORTRAN compiler by the mesh langage compiler should never contain any undetected errors. The user should not be required to inspect the FORTRAN output any more than a user need look at assembly langage output from a FORTRAN compiler.

7. Desirable additions. We should allow array % variables and % variables of real or double precision type. For example

```
%REAL ARRAY  A, B(10)
%INTEGER  XM(10)
```

The usual FORTRAN functions should be available within the % statements. Also, % variables could be initialized within their declarations. For example

```
%REAL CPI = 4.*ATAN(1.)
DATA PI/CPI/
```

The DATA statement in the output program sets the value of PI to π. Note that a % variable which appears outside a % statement is transformed to a character string in the output program. Thus, CPI becomes 3.1416... using the number of digits carried by the machine.

A macro-time repetition command. This is the

```
%REPEAT ... % UNTIL ... %ENDREPEAT
```

command. The Boolean expression following %UNTIL is the same type as that used with the %IF command. Note that we have not included labeled % statements or a %GOTO. This is probably a mistaken position from which we will have to retreat.

The & macro formal parameter marker. A formal parameter in a macro definition can be indicated by an identifier as in a FORTRAN subroutine. An integer constant or an INTEGER variable prefaced by a & could also be used. For example &(1) or &(N). This type of notation is found in many macro processors. For example suppose the macro has a variable number of arguments and is to generate a subroutine call for each argument.

```
%INTEGER N
%MACRO PLOTV
 % N=1
 %REPEAT
   CALL PLOT(&(N))
   % N=N+1
 %UNTIL &(N) .EQ. NULL
%ENDMACRO
```

Then the macro call

```
PLOTV(U,V)
```

would generate the statements

```
CALL PLOT(U)
CALL PLOT(V)
```

Note that we have added a macro-time repetition command

%REPEAT ..... %UNTIL .....

We have also added a reserved word to represent the null token, namely NULL.

Symbol table information. Our macro preprocessor is coupled to the compiler for PDELAN. This compiler has a symbol table containing information such as arithmetic type, array dimensions, etc. This information should be available within % statements. This can be done by providing another function call for the % statements (a suggestion of Tom Wright from NCAR). For example

```
% N1 = ARITHTYPE (&(1)
% N2 = DIMEN(&(1))
% N3 - DIMEXTENT(U,2)
```

The macro actual parameter substituted for &1 must be a single token which is an identifier. If the variable U is declared

```
REAL  U(20,30,10)
```

and &(1) = U, then N2 = 3 and N3 = 30.

A parenthesized list could be used as an actual argument. For example, consider the macro call

MAC(X,(A,B),Y)

In this &(1) would be replaced by X, &(2) by A,B. The parentheses are dropped. Also &(2).1 is A and &(2).2 is B. A similar construct is used in the macro preprocessor of Macleod [6].

A second type of expression macro. This type of macro allows macro-time conditional statements to be used to select the expression which defines the macro. A new type of statement is allowed which permits a concatenation of tokens. This statement is defined by occurrence of the macro name on the left of the "=" with a string of tokens separated by blanks on the right. The macro name can also appear on the right. The macro name represents a string of tokens which is null when the macro is entered. The contents of this string can be changed by the concatenation statement as the sequence of macro-time statements within the expression macro are executed. Only macro-time statements are allowed in the body of this expression macro. The contents of the string when the %RETURN is executed define the macro. For example consider the following definition of an A format specification. Here NWORD is a global %INTEGER variable. This type of macro is distinguished from the previous %EXPMACRO by the absence of the "=" in the %EXPMACRO statement.

```
%EXPMACRO AFORM
     %IF NWORD .EQ. 4 THEN
          %  AFORM = 20A4  %ENDIF
     %IF NWORD .EQ. 10 THEN
          % AFORM = 8A10 %ENDIF
     %  AFORM = FORMAT (AFORM)
     %RETURN
%ENDEXPMACRO
```

The following macro may not produce the same result as the one above.

```
%EXPMACRO AFORM
     %INTEGER N
     % N = 80/NWORD
     % AFORM = FORMAT(N A NWORD)
     %RETURN
%ENDEXPMACRO
```

When a macro-time %INTEGER variable appears in the expression macro string it is replaced by the character string which represents its value. If NWORD = 4, then the token string (N A NWORD) consists of 5 tokens (20 A 4). If these tokens are converted back into a character string before the output is given to a compiler, then this %EXPMACRO should produce the same result as the first macro. If the tokens are input directly to the syntactical scan of a compiler, then these two macros might not yield the same result. They would certainly output different token strings.

Macro-time string variables might be included which could increase the power of this second type of expression macro.

A pattern matching macro. It should be possible to insert a pattern matching macro into the preprocessor ahead of the lexical analysis which produces the tokens. If no pattern macros are defined, then the pattern matching could be suppressed. Pattern matching and token generation with symbol table lookup are said to cost about the same [11]. The pattern macro could be modeled after those in MORTRAN2 [14] which is fairly easy to understand and implement and probably provides sufficient power.

REFERENCES

[1] J. Gary and R. Helgason, "An Extension of FORTRAN Containing Finite Difference Operators", Software-Practice and Experience, 2, 1972, pp. 321-336.

[2] G. Locs and J. Gary, "A FORTRAN Extension for Data Display", to appear in IEEE Transactions on Computers.

[3] J. Martin, R. Zwakenberg, S. Solbeck, "LTSS Livermore Time-Sharing System", Computation Department, M-026, Lawrence Livermore Laboratory, Livermore, California.

[4] S. Mandil, "A General Purpose 'Problem-to-Program' Translater", Comp. Bull. 16, 1972, pp. 492-497.

[5] D. Kenmison, "FRED, A FORTRAN Editor", NCAR Scientific Library, National Center for Atmospheric Research, Boulder, Colorado, 80302, 1973.

[6] I. Macleod, "MP/1 - A FORTRAN Macroprocessor", Comp. Jour., 1970.

[7] H. Mills, "Topdown Programming in Large Systems", in "Debugging Techniques in Large Systems", Rustin(ed), Prentice Hall, 1971.

[8] J. Gary, "PDELAN: A Mesh Operator Variant of FORTRAN", Department of Computer Science, University of Colorado, Boulder, Colorado, 80302. 1974.

[9] P. Brown, "The ML/I Macro Processor", Comm. ACM, Vol. 10, pp. 618-623, 1967.

[10] W. Waite, "A Language-independent Macro Processor", Comm. ACM. Vol. 10, pp. 443-440, 1967.

[11] P. Brown, "A Survey of Macro Preprocessors", Annual Review in Automatic Programming, pp. 37-88, 1969.

[12] S. Pollack and T. Sterling, "A Guide to PL/I", Holt, New York, 1969.

[13] D. Knuth, "Structured programming with GOTO statement" Computing Surveys, 6, pp. 261-301 (1974)

[14] Workshop on FORTRAN preprocessors for numerical software, Jet Propulsion Laboratory, Pasadena, Calif., Nov. 1974.

[15] J. Cohen, "Experience with a conversational Parser Generating System", Software-Practice and Experience, Vol. 5, pp. 169-180 (1975).