1982

A Simple Macro Processor

John R. Rice
Purdue University, jrr@cs.purdue.edu

William A. Ward

Report Number:
81-400

A SIMPLE MACRO PROCESSOR

John R. Rice and William A. Ward

Computer Sciences
Purdue University
CSD-TR 400
May 5, 1982

ABSTRACT

The design objective for this macro processor is to be as powerful as possible and yet remain simple to use and implement. It was developed primarily to manipulate computer programs where the processor takes a symbol table plus a program template containing macros and produces a specific program. This approach is applied to the macro processor itself; the algorithm consists of a portable Fortran 66 version of the processor plus a program template of the processor. The macro processor template may be run through the portable macro processor to produce a version tailored to the local computing environment. In particular, it is easy to produce a Fortran 77 version of the macro processor.

1. THE MACRO PROCESSOR

A macro processor is a tool to substitute values from a symbol table into a text. Thus, if DATE has the value 'JULY 10, 1983' and PLACE has the value 'HONG KONG' then the text fragment DATELINE: $DATE, $PLACE. THIS OBSERVER... would be transformed into DATELINE: JULY 10, 1983, HONG KONG. THIS OBSERVER... Substitution is simple to understand and implement; complexity in a macro processor arises from facilities to control the substitution. See [Cole, 1976] for a survey of macro processors; some are almost complete programming languages. The macro processor presented here is designed to be as powerful as possible while remaining simple to use and implement. It is expressly designed to manipulate Fortran code although it is suitable for general text processing.

The two ingredients of macro processing are the symbol table and the input text. This processor has a very small initial symbol table (mostly consisting of processor option switches) so the input text contains the information to build the symbol table. The facilities are of four kinds: (1) substitution of text, (2) manipulation of the symbol table (3) control of the substitution, and (4) others (e.g. comments, processor options). The processor is keyed to two special characters: $, the substitution prefix and *, the directive prefix. The input has lines of text with processor commands embedded in them. Each line is first scanned for substitution and these are made. The line is then scanned for directives (the * must be the first non-blank character) and these are executed. If a substitution involves multiple lines then each line is processed as though it were input. This allows for indefinite nesting of substitutions which may include control directives.

The algorithm contains a complete user's guide for the macro processor so we limit further description here to compact tabular summary of the facilities, Table 1 and the processor options, Table 2.

The principal drawbacks to a portable macro processor in Fortran are (1) characters must be stored one per word and (2) the Fortran I/O packages are usually very inefficient. The input/output of the macro processor is isolated in the short routines IOERRM, IOLIST, IOPAGE, IOROLN, and IOWRLN. These may be replaced by more efficient, machine dependent routines without much difficulty.

Storing one character per word and using Fortran 66 makes the macro processor inefficient in space. These inefficiencies are not very significant for short texts or occasional use but become important with heavy use. For this reason a program template of the macro processor is included so the portable Fortran 66 version can produce a version which uses the character data type facilities of Fortran 77. Other tailoring, such as resetting standard unit numbers, can be made at the same time. The details of this procedure are given in the user's manual.

Table 1 below summarizes the facilities of the simple macro processor. Table 2 lists its options. The nature and use of the processor is, perhaps, best seen from the simple example application in the next section.

TABLE 1. SUMMARY OF MACRO PROCESSOR FACILITIES

1. TEXT SUBSTITUTION

Facility	Description and Examples
$(Name), $Name	Substitutes value of Name into text $(TYPE)VECTOR => REAL VECTOR or INTEGER VECTOR
$DEF(Name)	Returns .TRUE. or .FALSE. depending on whether Name is defined or not. Used for control in *IF facility.
$LIST(Name)	Substitute next item from list Name
*INCLUDE(Name)	Substitutes lines of text of Name. Similar to $(Name) on a line by itself, but behaves differently when substitution flag is off
LABEL	This is a special variable which is increcremented by 1 each time it is accessed. *SET(MAINLOOP = LABEL) *SET(EXIT = LABEL) DO $MAINLOOP I = 1, $ITEMS ... GO TO $EXIT ... $MAINLOOP CONTINUE produces DO 9004 I = 1,200 ... GO TO 9005 ... 9004 CONTINUE

2. SYMBOL TABLE CONSTRUCTION AND MANIPULATION

```
*SET(Name1 = Name2)
*SET(Name1 = 'literal')
*SET(Name1 = integer constant)
*SET
   ...
*ENDET

*SET(Name1)
   ...
*ENDSET
```

Assigns values to Name in the symbol table. Example to set several values.

```
*SET
    MONTH = 'APRIL'
    DAY   = 20
    YEAR  = CURRENTYEAR
*ENDSET
```

Example to set value to several lines

```
*SET(READTIME)
    IF(TIMER) CALL SECOND(TIME1)
    KTIME = KTIME+1
    TIME(KTIME) = TIME2-TIME1
    TIME1 = TIME2
 *ENDSET
```

*DELETE(Name)

Remove variable Name from symbol table

*APPEND(Name1, Name2)
*APPEND(Name1, 'literal')
*APPEND(Name1)
...
*ENDAPP

Append or concatenate text to Name1. Append is much more efficient than *SET when used for the same task. Multiple lines may be appended as follows:

```
*APPEND(PROCESSACCOUNT)
        PRINT $LABELB, ACCOUNT, BALANCE
$LABELB FORMAT('BALANCE OF ACCOUNT' I8 'IS' F12.2
       A          'ON $DAY $MONTH $YEAR')
*ENDAPP
```

adds three lines of code to process an account

3. CONTROL

*IF(Logical)Line
*IF(Logical)
Linetrue
*ELSE
Linefalse
*ENDIF

The text in Line, Linetrue and Linefalse is processed if the value of Logical is appropriate. Logical can be a logical constant (.TRUE., .FALSE.) or a logical variable (including $DEF(Name)). *IFs may be nested to any depth.

```
*IF(NOLIMIT) *SET(LIMIT = 1000)
*IF($DEF(LIMIT))
   *ELSE
      *SET(LIMIT = 1000)
*END IF
*IF(DEBUG) WRITE($(OUTPUT),66) X,Y,Z
*IF(SUPERUSER) *SET(PRIORITY = HIGHESTPRIORITY)
```

*DO(Name = i1,i2,i3)
...
ENDDO

DO-Loop much as in Fortran. Name assumes integer values so $(Name) becomes 12, say, in the text. The range specifications must be integer literals or variables with integer values.

```
DO (K = 1, NLIST, 3)
   $(K), $LIST(ITEMS) $LIST(ITEMS)-$LIST(ITEMS)
*ENDDO
```

produces (for NLIST = 9 and appropriate values in ITEMS)

```
   1, BIOLOGY  200-299
   4, MATHEMA  100-299
   7, PHYSICS  110-320
```

4. OTHER

*COMMENT
...
*ENDCOM

Comment lines. No substitutions are made or directives processed in comments.

*END	Terminate processing (end-of-file)
*RESET(Name)	Reset pointer for list Name to beginning of list
*OPTIONS(Name1 = Name2) *OPTIONS(Name1 = 'literal')	Set macro processor option Name1. Name2 or literal must be an appropriate value. Name1 values are given with defaults in parentheses. The possible options are listed in the next table.

TABLE 2. MACRO PROCESSOR OPTIONS

Name	Default	Definition
CDIR	*	Directive prefix character
CEOL	-	End-of-line marker is $-
CEOR	/	List item separator is $/
CONC	+	Continuation prefix character
CSUB	$	Substitution prefix character
ICPLI	72	Characters per line of input
ICPLO	72	Characters per line of output
IUNITI	5	Input unit number
IUNITL	6	Listing output unit number
IUNITO	7	Output unit number
LBREAK	.TRUE.	Switch to break output at nice character
LCOL1	.TRUE.	Only check column 1 for CDIR
LFORT	.TRUE.	Write lines with Fortran continuation
LISTI	.TRUE.	List input
LISTO	.TRUE.	List output
LSUB	.TRUE.	Process substitutions after this point
LITRIP	.TRUE.	Use one-trip DO-loops

2. APPLICATIONS.

This macro processor is powerful enough to be applicable to a wide range of typical macro processor applications. These range from processing simple form letters to complex "instrumentations" of programs and texts. The processor is tuned to Fortran in several ways (e.g. it has a special variable LABEL for creating Fortran labels) and is targeted to Fortran code manipulation. Typical applications include

(1) Implementations of very high level languages via Fortran preprocessors. These preprocessors have two components: Language parsing and code generation. The language parser saves values in a symbol table which define what is to be done, these are then merged with the template of a Fortran program to generate the specific Fortran code. The macro processor can implement this second component. Some substantial languages have been implemented using this

macro processor.

(2) Tailoring programs to specific environments. A Fortran program can be put into a template with many "parameters" to be inserted for a specific version. These parameters may range from something simple like the I/O unit numbers or the dimensions of certain arrays to complex things like whole subroutines for specific environments or changing program type e.g. from REAL to DOUBLE PRECISION. The following example illustrates this type of application.

Consider the LINPACK routines to factor and solve a system of linear equations. We want to be able to create a specific program with the following options:

1. The code may be single or double precision,

2. The matrix condition number may be estimated,

3. A right side may be read and the linear system solved.

A template for this program follows:

```
*IF (SINGLE)
      *SET ( DECL = 'REAL' )
      *SET ( PREFIX = 'S' )
*ELSE
      *SET ( DECL = 'DOUBLE PRECISION' )
      *SET ( PREFIX = 'D' )
*ENDIF
      $DECL  A($N,$N)
*IF (CONDNO)
      $DECL  RCOND, WORK($N)
*ENDIF
*IF (SOLVE)
      $DECL  IPVT($N)
      READ(5,*) A
*IF (CONDNO)
      CALL  $(PREFIX)GECO (A, $N, $N, IPVT, RCOND, WORK)
      WRITE(6,*) RCOND
*ELSE
      CALL  $(PREFIX)GEFA (A, $N, $N, IPVT, INFO)
*ENDIF
*IF (SOLVE)
      READ(5,*) B
      CALL  $(PREFIX)GESL (A, $N, $N, IPVT, B, 0)
      WRITE(6,*) B
*ENDIF
      STOP
      END
```

We see that the code is parameterized by the variables

DECL = Fortran declaration keyword

PREFIX	=	LINPACK subroutines name prefix character
N	=	Order of the linear system
CONDNO	=	Switch for condition number
SOLVE	=	Switch for solving liner system
SINGLE	=	Switch for single or double precision

If the program template is preceded by the macro instructions

```
*SET
    SINGLE = .TRUE.
    CONDO  = .FALSE.
    SOLVE  = .TRUE.
    N      = 10
 *ENDSET
```

Then the macro processor produces the program

```
REAL  A(10,10)
REAL  B(10)
INTEGER  IPVT(10)
READ(5,*) A
CALL  SGEFA (A, 10, 10, IPVT, INFO)
READ(5,*) B
CALL  SGESL (A, 10, 10, IPVT, B, 0)
WRITE(6,*) B
STOP
END
```

If the macro instructions are changed to SINGLE = .FALSE., CONDO = .TRUE., SOLVE = .FALSE. and N=5 then the macro processor produces the program

```
DOUBLE PRECISION  A(5,5)
DOUBLE PRECISION  RCOND, WORK(5)
INTEGER  IPVT(5)
READ(5,*) A
CALL  DGECO (A, 5, 5, IPVT, RCOND, WORK)
WRITE(6,*) RCOND
STOP
END
```

3. DISTRIBUTED MATERIAL.

The algorithm consists of the following files:

(1) Portable, Fortran 66 version of the macro processor

(2) Text of this paper

(3) User's guide for the macro processor

(4) Macro processor template

(5) Test cases

A. Exhaustive test of all facilities

B. Form letter to authors to report problems

C. The LINPACK example given above

D. The simple examples from the user's guide

E. A complex example: the ELLPACK system template

4. REFERENCES

A.J. Cole, Macro Processors, Cambridge University Press, Camridge, England, 1976.