1982

A Simple Macro Processor-User's Guide

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

William A. Ward

Report Number:
81-403

A Simple Macro Processor - User's Guide

by

John R. Rice and William A. Ward

May 20, 1982

CSD-TR 403

Abstract

This is the user's guide for the simple macro processor. This tool is designed to manipulate Fortran program text for applications like, 1. The implementation of very high level languages, 2. Modifying code for particular target environments, 3. Generating code for particular precisions or variable types. It has the basic facilities required for a general purpose macro processor and can be used in most macro processor applications.

This program was developed at Purdue University as part of the TOOLPACK project. This work was supported by the National Science Foundation under grant MCS - 7926310. Please report any bugs or suggestions to the authors at the address: Computer Science Department, Purdue University, West Lafayette, IN 47907.

Contents

May 28, 1982

1. General Description

This processor accepts as input templates or patterns for producing text in some user specified format. Although it has certain features which are particularly helpful when the text happens to be lines of FORTRAN, these features are not crucial to its operation. This processor is part batch editor, part macro processor, and part interpreter. It is a batch editor in the sense that it capable of manipulating text. It is a macro processor in that it is able to define and redefine macros and substitute their values when references to them occur in the input text. Finally, it is an interpreter because its commands are decoded and executed as they are encountered.

Specifically, the input consists of commands embedded in lines of text. These commands may take the form of either substitutions or directives. Two special characters are used to distinguish commands from ordinary text: the substitution prefix character (SPC) and the directive prefix character (DPC). For the remainder of this report the SPC is assumed to be $ and the DPC is assumed to be *. The processor operates as follows. Each input line is first scanned for occurrences of the SPC and if substitutions are required, then they are performed. Then the line is examined to determine if it is a directive line or a text line. If the first nonblank character in the line is the DPC, then the directive is decoded and executed. Otherwise, the line is assumed to be a text line and is copied to the output file. Note that substitutions may take place in both directive and text lines.

One should view the use of this processor as consisting of two phases. In the first, names and their associated values are entered into the symbol table. In the second, text containing directives and references to macros defined in the first phase is processed. In the simplest case, the symbol table might consist of the names STUDENT and ID along with their respective values 'John Jones' and 22215. The text input might be $STUDENT, ID = $ID and the output would be John Jones, ID = 22215. The processor thus has three levels of commands:

Directives which build or manipulate the symbol table:

*APPEND	Append a value to a variable.
*APPEND, *ENDAPP	Append lines of text to a variable.
*DELETE	Delete a variable from the table.
*RESET	Reset a list pointer.
*SET	Assign a value to a variable.
*SET, *ENDSET	Assign lines of text to a variable or perform multiple assignments.

Macro invocations:

$(NAME) or $NAME	Substitute the value of NAME.
$DEF(NAME)	Substitute '.TRUE.' if NAME is defined and '.FALSE.' otherwise.
*INCLUDE(NAME)	Include lines of text.
$LABEL or $(LABEL)	Substitute a new label.
$LIST(ITEMS)	Substitute the next item from ITEMS.
$$	Replace '$$' by '$'. (Embeds a single SPC in the text.)

Control directives

*COMMENT, *ENDCOM	Ignore the enclosed comment lines.
*DO, *ENDDO	Specify loops.
*END	End of input.
*IF, *ELSE, *ENDIF	Conditional processing of lines.

The features which distinguish this processor from other, perhaps more sophisticated, macro processors should be mentioned. First, it has a small and syntactically consistent set of commands which are easily learned. Second, it is capable of recursive macro substitution. Finally, it is written in PFORT-portable FORTRAN. Though this last feature somewhat reduces the processor's performance, it is an important advantage because it reduces the human time required to make the processor operational on a given system.

2. A Simple Example

The template processor is invoked by a subroutine call of the form

```
CALL TPDRV (iunit, lunit, ounit),
```

where iunit is the unit number of the input file, lunit is the unit number of the listing file, and ounit is the unit number of the output file. In the following example, four files are used: The first input file, assigned to unit 4, contains definitions of macros referenced in the second input file, assigned to unit 5. the listing file is assigned to unit 6 and the output file is assigned to unit 7.

Main program:

```
      CALL  TPDRV  (4, 6, 7)
      CALL  TPDRV  (5, 6, 7)
      STOP
      END
```

First input file (on unit 4).

```
*SET
       LASTNAME = 'DOE'
       FIRSTNAME = 'JOHN'
       MONTH = 08
       DAY = 24
       YEAR = 81
       SEMESTER = 'FALL'
*ENDSET
*SET ( NCOURSES = 3 )
*SET ( COURSES = 'BIO 255$$/' )
*APPEND ( COURSES, 'GEO 110$$/' )
*APPEND ( COURSES, 'PSY 201' )
*END
```

TPDRV applied to the first input file merely creates a sym-
bol table for later use.

Second input file (on unit 5).

```
*SET ( NAME = '$$LASTNAME, $$FIRSTNAME' )
*SET ( DATE = '$$MONTH/$$DAY/$$YEAR' )
NAME:          $NAME
DATE:          $DATE
SEMESTER:      $SEMESTER
LIST OF COURSES:
*DO ( I = 1, NCOURSES )
         $I.   $LIST(COURSES)
*ENDDO
*END
```

Template processor output (on unit 7).

```
NAME:          DOE, JOHN
DATE:          08/24/81
SEMESTER:      FALL
LIST OF COURSES:
         1.   BIO 255
         2.   GEO 110
         3.   PSY 201
```

TPDRV also produces a listing of these two files on unit 6.

Note that an item separator($/) is embedded in the text

by specifying $$/; otherwise, the processor would expect a name to follow the SPC. Also note that the same output results if the two input files are concatenated and read in as one.

3. Substitution Processing Facilities

3.1 Simple Substitutions

A reference to the macro NAME may be specified by $NAME or by $(NAME). The latter form of substitution is required when the substitution takes place within a block of text and it is necessary to separate the characters of name from those in the surrounding text. NAME may be an alphanumeric string of any length but its first character must be alphabetic.

Substitutions may be recursive. For example, if we have

Variable	Value
A	A(B)(E)
B	BC$(D)
D	D
E	E

then $A results in substituting ABCDE. The depth to which substitutions may be nested depends only on the amount of available workspace. Recursive substitution, along with memory management for the processor and the implementation of do loops and lists, all make use of the same pool of pointers. Therefore, a depth of recursion which is impossible when the symbol table is relatively full might easily be achieved when some space is returned (see *DELETE).

If a single occurrence of $ (the SPC) is needed in a string, then $$ should be specified. This is similar to the technique used by some FORTRAN compilers to embed quotes in quoted strings.

3.2 The $DEF Function

The $DEF function is used to determine if a name is defined in the template processor symbol table. $DEF(NAME) returns the string '.TRUE.' if NAME is defined and returns '.FALSE.' otherwise. This function is used with the *IF directive discussed below.

3.3 The $LABEL Function

The special variable LABEL is used to generate Fortran labels (or other sequences of increasing integers). Each time LABEL is referenced, its value is incremented by 1. Thus, when $LABEL or $(LABEL) is encountered in the text, the current value of LABEL is substituted, and LABEL is incremented. LABEL must be initialized using a *SET statement and may be assigned a new value at any time. Legal values for LABEL are integers between 1 and 99999 inclusive. If it is set to any other string, an error message is be generated the next time LABEL is referenced.

3.4 The $LIST Function

Lists of items may be created and substituted. The list separator is $/ and thus

```
$SET(XYZ = ' A $$/ B+C $$/ D*E $$/ ')
$SET(THREELINES)
     THIS IS LINE 1
   $$/THIS IS LINE 2
   $$/THIS IS LINE 3
   $$/
*ENDSET
```

creates two lists: XYZ = (A, B+C, D*E) and THREELINES = the 3 lines given above. The statement

```
$APPEND (XYZ, ' F-G $$/ HIJ $$/ ')
```

lengthens the list XYZ by adding the two items F-G and HIJ.

The $LIST function is used to substitute items from a list one at a time. Thus the input text

```
X = $LIST(XYZ)
Y = $LIST(XYZ)
Z = $LIST(XYZ) - ($LIST(XYZ))
```

results in the output text

```
X = A
Y = B+C
Z = D*E - (F-G)
```

The next reference $LIST(XYZ), wherever it might be, produces HIJ. This function treats a variable as a list of items separated by the marker $/. Each list has a pointer which is incremented each time the list is referenced. The pointer may be reset to the start of the list. Thus the input

```
$RESET(XYZ)
```

```
K = $LIST(XYZ)
```

results in

```
K=A
```

If the RESET were not present, one would obtain K=HIJ instead. If the pointer reaches the end of the list, then the next reference to the list results in an error message and no substitution is made.

3.5 Special Escape Characters

The use of special escape characters has already been illustrated; the complete list is as follows:

$ Substitution prefix character. It signals that the next name is a variable whose value is to be substituted. The character is doubled to $$ to obtain a $ inside a quoted string.

* Directive prefix character. It signals the beginning of a macro processor directive provided it is the first non-blank character on a line. Otherwise * is an ordinary character.

$- End-of-line marker. This special character is placed at the end of every input line. Its effect can br overridden by the continuation marker.

$+ Continuation marker. This special character at the end of a line of input overrides the end-of-line marker and continues the line with the text from the next line.

$/ Item separator. It separates items in a list.

The input text

```
*SET ( AVERYLONGNAME = 'A VERY $+
 LONG STRING' )
```

assigns the same value to AVERYLONGNAME as the line

```
*SET ( AVERYLONGNAME = 'A VERY LONG STRING' )
```

Variables may contain end-of-line markers, if LINES has the value

```
'       T = A $-       A = B $-       B = T $-'
```

then *INCLUDE(LINES) results in

```
T = A
A = B
B = T
```

It has already been noted that $$ is used to embed an occurrence of the SPC $ in a string for later processing. Note that

```
$SET(LINES = ' T=A $$- A=B $$- B=T $$-')
```

or

```
$SET(LINES)
     T=A
     A=B
     B=T
*ENDSET
```

can be used to assign LINES the value used above. If the double $$ is not used in the first case, then the macro processor attempts to make an immediate substitution for $-; this, of course, fails as there is no variable with - as a name.

Finally note that lists may contain lines and lines may contain lists but the end-of-line marker $- does not indicate an end-of-item marker $/ or vice-versa.

4. Macro Processing Directives

This section list (in alphabetical order) the directives of the macro processor which provide the facilities to create the symbol table and to control the substitution phase.

4.1 *APPEND and *ENDAPP

The first form of this construct is

```
*APPEND ( name1 , name2 or literal ).
```

and the second form of this statement is

```
*APPEND ( name1 )
     lines of text
*ENDAPP
```

This statement is similar in syntax and function to the SET

command except that instead of assigning a value to the name, it appends (or, if you prefer, concatenates) the indicated value to the string already associated with the name. To emphasize this difference a, comma (,) instead of an equal (=) is used to separate the name from the right side. One should also note that it is much more efficient to append one variable to another by using *APPEND (A, B) than by using *SET (A = 'AB').

4.2 *COMMENT and *ENDCOM

Input files may be documented by bracketing comment lines with the directives *COMMENT and *ENDCOM. All lines between these two directives are ignored.

4.3 *DELETE

The statement

*DELETE (name)

deletes the variable 'name' from the symbol table.

4.4 *DO and *ENDDO

This control statement has a form similar to the Fortran DO-loop:

*DO (iname = i1, i2, i3)

The variable iname is the loop index and i1, i2 and i3 are the initial, final and incremental values, respectively. The behavior of this construct is very similar to its FORTRAN 66 counterpart, that is

a) The index may not be modified within the range of the do loop.

b) The loop is always performed at least once .

c) i1 is the starting value for iname, i2 is the termination value, and i3, which is optional, is the step size for index. i1, i2, and i3 must be integers or names of template variables containing integer values.

d) The do range is closed by the statement *ENDDO.

e) The depth to which do loops may be nested is restricted

only be the amount of available workspace.

To illustrate its use suppose one has a list COEFS with values -12.3, 16.2, -4.9, 8.2,... . Then the three statements

```
*DO (I = 1, 5, 2)
      A($I,1) = B($I) + ($LIST(COEFS))
*ENDDO
```

result in

```
      A(1,1) = B(1) + (-12.3)
      A(3,1) = B(3) + ( 16.2)
      A(5,1) = B(5) + ( -4.9)
```

4.5 *END

This statement terminates the execution of the macro processor.

4.6 *IF, *ELSE, and *ENDIF

This control statement has the form

```
*IF ( lname or lconstant ) directive or text line
    or
*IF ( lname or lconstant )
      lines in the true range
*ELSE
      lines in the false range
*ENDIF
```

The expression inside the parentheses following the IF must be the name of a variable with a logical value, or one of the logical constants .TRUE. or .FALSE. For the one-line form, if the value is true, then the line following the closing parenthesis is processed, otherwise it is ignored. For the multi-line form, if the value is true, then the lines in the true range are processed, otherwise, those in the false range are processed. *ELSE's are optional and *IF's may be nested to any depth as illustrated by the following example.

```
*SET ( L1 = .TRUE. )
*SET ( L2 = .TRUE. )
*SET ( L3 = .FALSE. )
*IF  (L1)
       A = B
       *IF (L2)
              B = C
              *ELSE
              B = D
       *ENDIF
       *IF ( L3 )       C = D
       *IF ( $DEF(L3)) D = E
*ENDIF
```

results in

```
      A = B
      B = C
      D = E
```

Note that even though L3 is .FALSE., $DEF(L3) is .TRUE. because L3 has been assigned a value.

4.7 *INCLUDE

*INCLUDE(Name) is alternate form of substitution; it must appear on a line by itself. Its purpose is to allow one to substitute lines of text without performing the substitutions within these lines. There is a switch (see *OPTION below) to turn substitution off and on. If this switch is on, then *INCLUDE(Name) and $(Name) are identical. The use of this facility is illustrated by the text

```
*SET(LINSYSCALL)
   *IF(TIMER)
      *INCLUDE(TIME1)
   *ENDIF
   CALL $NAME($MATRIX,$SOLUTION,$RHS,$NUMBEQNS,WORK,IER)
   *IF(TIMER)
      *INCLUDE(TIME2)
   *ENDIF
*ENDSET
```

When this text is processed with the substitution switch off and TIMER = .TRUE., the value of LINSYSCALL has lines of code before and after the subroutine call which time the execution of the subroutine. For example, we might have

```
$SET(TIME2)
   CALL SECOND(TIME2)
      TIME(KTIME) = TIME2-TIME1
```

```
      PRINT $TIMELABEL, TIME(KTIME), $NAME
      KTIME = KTIME+1
*ENDSET
```

Later in the processing there will probably be a line *INCLUDE(LINSYSCALL) with the substitution switch turned on. At that the value values for $NAME and $TIMELABEL are substituted in the code to print out the timing information and the name of the subroutine being timed.

4.8 *OPTION

This command has the form

*OPTION (option name = name or literal)

and is used to set internal flags and values which affect the operation of the template processor. The following table lists the options with their defaults.

Name	Default	
CDIR	*	The directive prefix character *.
CEOL	-	The second character of the end-of-line marker.
CEOR	/	The second character of the end-of-item marker.
CONC	+	The second character of the continuation marker.
CSUB	$	The substitution prefix character.
ICPLI	72	The number of characters per line of input.
ICPLO	72	The number of characters per line of output.
IUNITI	5	The input unit number.
IUNITL	6	The listing unit number.
IUNITO	7	The output unit number.
LBREAK	.TRUE.	Try to break lnes longer than ICPLO at a convenient break character.
LCOL1	.TRUE.	Check only column 1 in the input file for occurrences of the DPC.
LFORT	.TRUE.	Write out long lines by providing Fortran continuation cards.
LIST1	.TRUE.	List input lines as they are read in.
LISTO	.TRUE.	List output lines as they are written out.
LSUB	.TRUE.	Start processing substitutions.
L1TRIP	.TRUE.	Use 1-trip do-loops.

Note that option names which begin with c, i, and l expect a value which consists of a character, an integer, or a logical value, respectively.

4.9 *RESET

*RESET(ITEMS) resets the list pointer of 'ITEMS' to the beginning of the list.

4.10 *SET and *ENDSET

The first form of this statement is

```
*SET ( name1 = name2 or literal ).
```

The value to be assigned on the right of the = may be either a variable name, in which case the value of name2 is assigned to name1, or a literal, in which case the literal itself is assigned to name1. A literal may be a quoted string, an integer constant, or one of the logical constants .TRUE. or .FALSE.

The second form of this statement is

```
*SET ( name1 )
      lines of text
*ENDSET
```

This causes the variable name1 to take on as its value the lines of text up to the next matching *ENDSET. End of line markers are automatically supplied. These lines of text are processed only for substitution (if the substitution switch is on) and are not examined for directives. In particular, if these lines contain another *SET - *ENDSET pair, the inner *SET will not be processed. For example,

```
*SET( A )
    *SET( B )
        X = Y + Z
    *ENDSET
*ENDSET
```

causes A to take on the value

```
'*SET ( B )$-        X = Y + Z$-     *ENDSET$-'
```

The variable B, however, will not be given a value until $A is encountered later.

The third form of this statement is the multiple assignment and is illustrated by

```
*SET
    name1 = value1
    name2 = value2
    name3 =
            line a
            line b
*
    name4 = value4
*ENDSET
```

The distinguishing feature of this form is that there is no name following *SET. The result of this statement is identical to

```
*SET ( name1 = value1 )
*SET ( name2 = value2 )
*SET ( name3 )
            line a
            line b
*ENDSET
*SET ( name4 = value4 )
```

Note that the * on a line by itself ends a group of text lines in a multiple assignment *SET.

5. Further Examples

5.1 An Example Which Modifies A LINPACK Program

The following example illustrates how the macro processor may be used to generate a program segment. Some background information is needed to understand this example. First, the resulting program calls LINPACK routines to factor and possibly solve a system of linear equations. Recall that the LINPACK routine SGECO factors a matrix and produces an estimate of its condition number, SGEFA only factors the matrix, and SGESL takes the factored matrix and solves a linear system given a right side. The double precision versions of these routines are DGECO, DGEFA, and DGESL, respectively. Finally, this example assumes that the template variables CONDNO, N, SINGLE, and SOLVE have already been set elsewhere to appropriate values.

Input text:

```
*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
```

Output text assuming SINGLE = .TRUE., CONDNO = .FALSE., SOLVE = .TRUE., and N = 10,

```
      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
```

Output text assuming SINGLE = .FALSE., CONDNO = .TRUE., SOLVE = .FALSE., and N = 5,

```
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
```

5.2 Other examples with the algorithm distribution.

The ACM algorithm is distributed with five files of test cases. The first and most important is an example that exhaustively tests all the facilities of the macro processor. The output from processing this text as input should be pairs of lines identical except for macro substitutions. The first line of each pair is produced by the processing of the text and the second line is the correct result of this processing. If the pairs are not identical, then there is something wrong with the macro processor. If the pairs are identical, then one has high confidence that the macro processor is working correctly. Alas, this exhaustive test is not infallible; we have found one bug (so far) that was not revealed by this test.

The second test is a simple form letter to the authors that may be used to report any problems with the algorithm. The third test is the LINPACK example given above and the fourth test is the set of all the simple illustrations given in this user's guide.

The final example is the temple used for code generations in the ELLPACK system. This is a complex application of the macro processor which involves dozens of variables, whose values are set by the parser for ELLPACK. The most complex part of the template is the statement *INCLUDE(SRCALLS) which brings most of the executable code for the resulting Fortran program. This variable is built up by the parser and has considerable depth of recursive substitution; it usually consists of a hundred or so lines of Fortran.

6. Preparation of a Tuned Version of the Macro Processor

The simple macro processor may be tuned by setting the following template variables to appropriate values and then applying the basic processor to the template for the macro processor.

CDC if .TRUE., a Purdue CDC compatible version is produced.

CSTAR1 if .TRUE., Fortran 77 declarations of the form character*1 are used instead of integer declarations for Hollerith variables.

DEBUG if .TRUE., MNF trace statements will be inserted. this should only be used if CDC = .true.

ICBDIM the dimension of the array CBUFFR. This dimension limits the size of text that the APPEND, INCLUDE and SET statements can process.

ICSDIM the dimension of the array CSTORE. The dimension limits the total amount of information in the symbol table.

IHADIM the dimension of the array IHASH. This dimension limits the number of names used. This should be a prime number.

ISTDIM the dimension of the array ISTORE. This dimension limits the number of pointers to the main storage area CSTORE. Should be less than ICSDIM.

NOPACK if .TRUE., all references to the array CSTORE will be direct (in-line) instead of being forced through subroutines.

SHORTB if .TRUE. and CDC = .TRUE., short file buffers will be used.

STATS if .TRUE., MNF timing statements will be inserted. This should only be used if CDC = .TRUE.

TESTCH if .TRUE., character testing used to check for alphabetic and numeric is performed using in-line IF statements instead of being isolated in separate subroutines.

UNIX produce a UNIX compatible version.

7. Error Messages

Each error message is preceded by the name of the subroutine in which it occcured.

IOREAD - BUFFER SPACE EXCEEDED
Cause: An input line or the multi-line text of an APPEND, INCLUDE, or SET was too long. Solution: Make the text shorter or recompile the processor with CBUFFR

dimensioned larger.

MMGET1 - STRING TOO LONG FOR CVALUE(*)
Cause: The buffer passed to MMGET1 was too short. If you are not calling subroutines of the processor yourself, then this argument is probably the internal array CBUFFR. Solution: Recompile the processor with a larger dimension for CBUFFR.

MMHASH - HASH TABLE ARRAY IHASH(*) is full
Cause: The hash table IHASH has been filled because the maximum number of names has been exceeded. Solution: Recompile the processor with a larger dimension for IHASH.

MMNEWI - STORAGE ARRAY ISTORE(*) IS FULL
Cause: All of the pointers used in dynamic memory management have been allocated. This may be due to memory fragmentation or simply to a large number of macro definitions. Solution: Recompile the processor with a larger dimension for ISTORE.

MMPOPC - STRING TOO LONG FOR BUFFER
Cause: The buffer passed to MMPOPC was too short. If you are not calling subroutines of the processor yourself, then this argument is probably the internal array CBUFFR. Solution: Recompile the processor with a larger dimension for CBUFFR.

MMPUT1 - STORAGE ARRAY CSTORE(*) FULL
Cause: The character storage array CSTORE has been filled. Solution: Recompile the processor with a larger dimension for CSTORE.

MMRETI - ATTEMPT TO RETURN INVALID POINTER
Cause: Should not occur under normal circumstances. Solution: Report the problem to the TOOLPACK group at Purdue University.

MPITEM - VARIABLE NOT DEFINED
Cause: A reference was made to an undefined macro. Solution: Check to see if the variable name is spelled properly and if it is actually defined.

MPLABL - ILLEGAL LABEL VALUE
Cause: The variable LABEL was assigned a non-integral value or a value that was longer than 5 digits or non-positive. Solution: Correct LABEL assignment.

MPMAC - variable not defined
Cause: A reference was made to an undefined macro. Solution: Check to see if the variable name is spelled properly and if it is actually defined.

MPPOPN - ILLEGAL VARIABLE NAME
Cause: An illegal variable name was encountered. Solution: Check to make sure the name begins with an alphabetic character and that it contains no special characters.

MPPOPN - MISSING RIGHT PARENTHESIS
Cause: Unbalanced parentheses
Solution: balance parentheses.

TPAPPE - APPEND HAS NO MATCHING ENDAPP
Cause: A multi-line APPEND has no closing ENDAPP. This is only detected when an END statement is encountered, so at this point the APPEND statement has gobbled up the end of your template.

TPCOMM - COMMENT HAS NO MATCHING ENDCOM
Cause: COMMENT has no closing ENDCOM. This is only detected when an END statement is encountered, so at this point the COMMENT statement has gobbled up the end of your template.

TPDO - DO HAS NO MATCHING ENDDO
Cause: A DO statement is not closed by an ENDDO. This is only detected when an END statement is encountered, so at this point the DO statement has gobbled up the end of your template.

TPELSE - IF HAS NO MATCHING ENDIF
Cause: An IF statement is not closed by an ENDIF. This is only detected when an END statement is encountered, so at this point the IF statement has gobbled up the end of your template.

TPENDO - MISPLACED ENDDO
Cause: An ENDDO not preceded by a matching DO has been encountered.

TPENDF - MISPLACED ENDIF
Cause: An ENDIF not preceded by a matching IF has been encountered.

TPEVAL - MISPLACED ENDAPP
Cause: An ENDAPP not preceded by a matching APPEND has been encountered.

TPEVAL - MISPLACED ENDCOM
Cause: An ENDCOM not preceded by a matching COMMENT has been encountered.

TPEVAL - MISPLACED ENDSET
Cause: An ENDSET not preceded by a matching SET has been encountered.

TPEVAL - ILLEGAL OR MISSPELLED DIRECTIVE
Cause: An unknown directive has been encountered.

TPIF - IF HAS NO MATCHING ENDIF
Cause: An IF has no closing ENDIF. This is only detected when an END statement is encountered, so at this point the IF statement has gobbled up the end of your template.

TPINCL - VARIABLE NOT DEFINED
Cause: A reference was made to an undefined macro.
Solution: Check to see if the variable name is spelled properly and if it is actually defined.

TPOPT - ILLEGAL OR MISSPELLED OPTION
Cause: An unknown option has been encountered.

TPOPT - OPTION REQUIRES SINGLE CHARACTER
Cause: An incorrect value has been supplied to an option which requires a single character.

TPOPT - OPTION REQUIRES AN INTEGER
Cause: An incorrect value has been supplied to an option which requires a integer value.

TPOPT - OPTION REQUIRES A LOGICAL VALUE
Cause: An incorrect value has been supplied to an option which requires a logical value of .TRUE. or .FALSE.

TPSET - SET HAS NO MATCHING ENDSET
Cause: A multi-line SET has no closing ENDSET. This is only detected when an END statement is encountered, so at this point the SET statement has gobbled up the end of your template.

TPSYNT - LEFT PARENTHESIS EXPECTED
Cause: A left parenthesis was expected but not found.

TPSYNT - RIGHT PARENTHESIS EXPECTED
Cause: A right parenthesis was expected but not found.

TPSYNT - COMMA EXPECTED
Cause: A comma was expected but not found.

TPSYNT - EQUALS SIGN EXPECTED
Cause: An equals sign was expected but not found.

TPSYNT - VARIABLE EXPECTED
Cause: A special character or constant was encountered where a variable name was expected.

TPSYNT - CONSTANT OR VARIABLE EXPECTED
Cause: A constant or variable name was expected but not

found.

TPSYNT - ILLEGAL CHARACTERS AT END OF LINE
Cause: Extra characters were found at the end of the input line.