1982

PG - A Preprocessor Generator

John Brophy

Report Number:
84-472

PG - A Preprocessor Generator

John Brophy

Computer Science Department
Purdue University

CSD-TR 472
February 10, 1982

Libraries and packages of Fortran routines already exist in many areas of scientific computing, however their use demands a familiarity with the details of the routine. The user must allocate storage, determine the proper sequence of routines to call, etc. Therefore considerable time and effort can be spent dealing with problems of Fortran programming. Such libraries can be made much easier to use if a language is provided to describe the problem to be solved in a natural manner. PG is a program designed to generate preprocessors to convert programs written in such languages to Fortran.

PG has been used to write a preprocessor for the ELLPACK language. ELLPACK is a language for solving elliptic partial differential equations. Examples from the ELLPACK preprocessor are used in this report, but knowledge of ELLPACK is not needed to understand them. To implement a problem oriented language, such as ELLPACK, both a grammar for the language and a driver routine for the preprocessor must be supplied. The driver routine is usually fairly short and is needed to initialize the program, open files, etc. A more detailed description of what is needed is given below. The grammar describes the language to be implemented. PG converts the grammar into a set of Fortran subroutines which are called by the user supplied driver routine.

PG is designed around the macro processor, described in Appendix C. The preprocessor contains routines for storing text strings and a template processor. The use of these routines is described below.

1. Introduction to PG Grammars

A preprocessor must perform two basic tasks: parsing the input language, e.g. ELLPACK and generating an output program. Parsing is controlled by grammar **rules** and generating an output program is done by **actions**.

In a PG grammar a line beginning with an asterisk is a comment; blank lines are also considered comments. Either type may occur anywhere in the grammar. All lines are assumed to be at most 80 characters long. A statement may be continued

to a new line by ending the current line with an ampersand(&).

We start with a number of simple examples from the ELLPACK language and its preprocessor. A very simple PG grammar, consisting of just one rule is

```
PROG      ->   'OPTION.'   NAME   '='   NAME
END.
```

The name of the rule is PROG; rule name must be valid Fortran names. To the right of the arrow (->) are **subrules.** Subrules are either character strings to be matched or the name of another rule. PG is **not** recursive and it is illegal for a rule to use itself as a subrule directly or indirectly, as in rule A using rule B using rule A. There are several built-in rules: NAME is a built-in rule which matches an alphanumeric string without embedded blanks. The last statement in a PG program must be 'END.'. Valid input to the preprocessor generated by the above grammar is

```
OPTION.   WORKSPACE = 5000
```

It is assumed that blanks in the problem-oriented language, e.g. ELLPACK, are usually not significant, so blanks are skipped automatically **after** each subrule. The string OPTION. must occur in the first column of the input, but the rest of the line can begin in any column.

The text matched by NAME may not contain any embedded blanks. If it is desired to allow embedded blanks, then the following program can be used

```
PROG      ->   ALFNUM+
END.
```

ALFNUM is a predefined rule which matches any alphanumeric character, (see Table 1 for a complete list of built-in rules). The '+' is a PG operator and means match one or more occurrences of ALFNUM. Since blanks are skipped after **every** character matched by ALFNUM embedded blanks are ignored. The '+' may be used with either a rule name, as here, or with a quoted string. The complete set of PG postfix operators is given in Table 2.

In ELLPACK there are two forms for the option statement, e.g.

```
OPTION.   WORKSPACE = 5000
```

and

```
OPTION.   MEMORY
```

The PG grammar for the option segment is

```
PROG      ->   ALFNUM+   '='   ALFNUM+
          ->   ALFNUM+
END.
```

The preprocessor will first try to use the first rule. If it fails, i.e. one of its subrules fails to match the input, then it will try the next **alternative**, marked by -> with no rule name to its left, will be tried. A rule may have any number of alternatives.

The grammar and its rules define the input language and

the actions associated with these define the output. The actions are executed when the associated alternative is matched. Most actions are to set or reset variables in the template processor or to write Fortran statements. For example, the action statement

```
$(TPVNAME) = 'ASTRING'
```

sets the template variable TPVNAME to ASTRING. Template variable names must begin with a letter and contain only letters and digits; there is no limit on their length. A reference to a template variable is indicated by $ preceding the template variable name. The PG grammar used by ELLPACK to set a template variable via the OPTION segment is

```
PROG      ->   'OPTION.'   ALFNUM+   '='   ALFNUM+
                     $($2) = $4
          ->   'OPTION.'  ALFNUM+
                     $($2) = '.TRUE.'
END.
```

The dollar notation (adopted from YACC) is as follows: each element in a rule or alternative is numbered in order of its occurrence and **what it matches** is referenced by its number preceded by $. In the first rule above, $1 is OPTION., $2 is whatever is matched by the first ALFNUM+, $3 is =, and $4 is whatever is matched by the second ALFNUM+.

Thus, if the input is

```
OPTION.   ->   WORKSPACE = 5000
```

then $2 is WORKSPACE and $4 is 5000. When this line is read by the ELLPACK preprocessor generated using this grammar the template variable WORKSPACE is set to the string 5000. If the input is

```
OPTION.   MEMORY
```

then the template variable MEMORY is set to .TRUE.. Note that this approach allows one to associate language keywords, (e.g. WORKSPACE and MEMORY), with template variables.

While all template variables contain text strings, special provision is made for two special classes. A **logical template variable** is on that contains either the text string .TRUE. or .FALSE.. An **integer template variable** is one that contains a character string representing an integer. A logical template variable can be set using the action

```
$L(TPVNAME) = <LOGICAL EXPRESSION>
```

and an integer template variable can be set using

```
$I(TPVNAME) = <INTEGER EXPRESSION>
```

In addition, both $L(TPVNAME) and $I(TPVNAME) can be used in expressions as if they were Fortran function calls of the corresponding type. The use of these features is illustrated by the following grammar.

```
PROG      ->   'OPTION.'   ALFNUM+   '='   ALFNUM+
                     $($2) = $4
                     $I(TOTAL) = $I(TOTAL) + $I($2)
                     $L(TOOBIG) = $I($2) .GT. 10000
```

Using the same input as before will result in WORKSPACE being

set to 5000, adding 5000 to the variable TOTAL and setting TOOBIG to .FALSE..

There is a **string concatenation** operator //. The grammar

```
PROG      ->   'OPTION.'   ALFNUM+   '='   ALFNUM+
                    $(DOUBLE) = $4 // $4
                    $L($2 // SET) = .TRUE.
     END.
```

with the same input as above, will set DOUBLE to 50005000 and set WORKSPACESET to .TRUE..

There is an IF-ELSE-ENDIF construct in PG, illustrated by the following grammar.

```
PROG      ->   'OPTION.'   ALFNUM+   '='   ALFNUM+
                    IF ( $I($2) .LT. $I($4) )
                         $($2) = $4
                    ENDIF
     END.
```

With the same input as above, this sets WORKSPACE to the maximum of the currently assigned value and any previously assigned value. Note that $I($2) is an integer value of WORKSPACE while $($2) is the character string '5000'. Any number of action statements may occur between the IF and the ELSE and between the ELSE and the ENDIF.

There is also a set of PG **utility routines** to aid in programming the action statements. The logical function

```
LQMTCH ( <STRING1>, <STRING2> )
```

returns .TRUE. if <STRING1> the same as <STRING2>. The integer function

```
IQIVAL ( <STRING> )
```

returns the numerical value of <STRING>. The complete set of these PG functions is given in Table 4.

The following grammar will set WORKSPACE to the maximum of its previous value and the current value. Note that if the option is not WORKSPACE then no action is taken.

```
PROG      ->   ALFNUM+   '='   ALFNUM+
                    IF (LQMTCH($2, 'WORKSPACE'))
                         $($2) = MAX0($I($2), IQIVAL($4))
                    ENDIF
     END.
```

We could **not** use the action

```
$($2) = MAX0( $I($2), $I($4) )
```

above, because $I applies only to template variables and $4 stands for the string 5000 which is not the name of a template variable. We could, however, have used

```
$(TEMP) = $4
$($2) = MAX0( $I($2), $I(TEMP) )
```

In addition to the above PG utilities which act like functions there are PG **procedures** which act like subroutine calls. The grammar

```
PROG      ->   'OPTION.'   ALFNUM+   '='   ALFNUM+
                    WRITE('      ' // $2 // $3 //$4)
          ->   'OPTION.'   ANY+
                    ERROR('ILLEGAL OPTION - ' // $2)
          END.
```

Given the input

```
OPTION.   LEVEL = 3
```

the statement

```
LEVEL=3
```

will be written to the **program** file. If it is given the input

```
OPTION.   MEMORY
```

then it will write the error message

```
ILLEGAL OPTION - MEMORY
```

to the **listing** file and set a flag indicating that an error has occurred. The complete list of PG procedures is given in Table 3.

The following grammar which shows how to allow several options separated by dollar signs, to occur on the same line introduces several new features.

```
PROG      ->   'OPTION.'  OPTEXP ++ '$'
OPTEXP    ->   OPTNAM  '='  ALFNUM+
                    $($1) = $3
                    CLEAR
OPTNAM    ->   ALFNUM+
          END.
```

The rule PROG is defined in terms of the subrules OPTEXP and OPTNAM. Given the input

```
OPTION.   LEVEL = 3    $    WORKSPACE = 5000
```

this rule and its actions sets the template variable LEVEL to 3 and sets WORKSPACE to 5000. The infix operator ++ as in

```
<SUBRULE1> ++ <SUBRULE2>
```

matches a list of <SUBRULE1> type objects separated by objects of matching <SUBRULE2>. Here we want a list of options separated by dollar signs. The text matched by ALFNUM+ on the right side of the OPTNAM rule is passed back to the OPTEXP rule.

Text would be passed back to the PROG rule except for the CLEAR procedure. If text were always passed back to the PROG rule the system would retain the entire input. This would be very wasteful of space since there is no need to keep the input text once it has been parsed and the relevant actions performed. The CLEAR statement tells PG that all the text matched so far can be thrown away.

2. PG Rule Syntax

The basic PG grammar rules are quoted strings and a set of built-in rules. (A quoted string may be thought of as a special kind of built-in rule).

Table 1

The built-in rules of the PG system. Blanks are automatically skipped after each rule, except for rules marked (NS).

Rule Name	Matches
'TEST'	quoted text, used '' to quote a quote
ALFNUM	a letter or a digit
ANY	anything but an end-of-line
BLANKS	any number of blanks
CHAR	anything but an end-of-line (NS)
DIGIT	a digit
EOL	end-of-line (NS)
FEXPR	Fortran expression
FNAME	Fortran-type name
LETTER	a letter
LINE	everything up to an end-of-line
NAME	name (alphanumeric string without embedded blanks)

A basic rule may have a postfix operator applied to it. The postfix operators are given in Table 2.

Table 2

The postfix operators in PG.

Operator	Effect
RULE *	match zero or more occurrences of RULE
RULE +	match one or more occurrences of RULE
RULE ?	match zero or one occurrences of RULE
RULE -	match rule, but otherwise **ignore** matched text, do not pass the text back up
RULE ** N	match exactly N (>0) occurrences of RULE
RULE ** 0	match rule, but then back up the input so that the text is not considered matched
RULE ?* N	match up to N (>0) occurrences of RULE
RULE ?-	match zero or one occurrence of RULE, but ignore matched text

The **list operator**, RULE1 ++ RULE2, matches a list of RULE1 type objects separated by RULE2 type objects. For example, FEXPR ++ ',' matches a list of Fortran expressions separated by commas.

3. PG Actions

In a PG action a **basic string variable** is either $<INTEGER>, or a quoted string. If the quoted string contains only letters and digits the quotes may be omitted. A quote may be included in a quoted string by using two quotes. A **string variable** is either a basic string variable, the value of a template variable or a combination of these concatenated using the concatenation operator //. A template variable is referenced by $(<SV>), where <SV> is a string variable. Recall the definition that $1 refers to the text matched by the first subrule, $2 to the text matched by the second subrule, etc. Examples of string variables are

```
'X+3.6*Y'
'ISN''T'
TEXT
$(TEMPLATE)
$3
ABC // $(VARIABLE) // $2 // $(DEF)
$( ABC // $(XYZ) ) // R12
```

Text matched with an ignore operator attached ('-' or '-?') cannot be accessed with $<N>.

A PG **action statement** may assign a value to a template variable. The action statement

```
$( <SV1> )  =  <SV2>
```

assigns the string <SV2> to the template variable <SV1>. To append the string <SV2> to <SV1> the action statement

```
$( <SV1> )  <-  <SV2>
```

is used.

If the value to be assigned to a template variable is either logical ('.TRUE.' or '.FALSE.') or an integer (an optional '+' or '-', followed string of digits) then the special assignment statements

```
$L( <SV> )  =  LOGICAL-EXPRESSION
$I( <SV> )  =  INTEGER-EXPRESSION
```

may be used. The logical and integer expressions follow the usual Fortran rules augmented by the logical and integer valued functions

```
$L( <SV> )
$I( <SV> )
```

For example,

```
$I(COUNT)  =  $I(COUNT) + 1
```

takes the digits in the string named by COUNT, converts them to an integer, adds one to the value; the new value is then converted back to a digit string and assigned to COUNT.

PG action statements may also involve ordinary Fortran statements with template variables. These statements are local to the set of actions associated with a rule and do not, in

general, affect the action statements of other rules (but see the GLOBAL statement below). Fortran variables in these statements have their types determined by the usual Fortran rules. The following example shows how one can access Fortran facilities in a PG action statement using Fortran statements.

```
IWORK  = $I(WORKSPACE)/2 + 10
SCALE  =  ALOG10( FLOAT(MAX0(IWORK,100)) ) + 1
$I(WORKSPACE)  =  10**SCALE + $I(WORKSPACE)
```

The built-in action procedures are given in Table 3. They each have one of the forms

```
<NAME>
<NAME> ( <SV> )
<NAME> ( <SV1>, <SV2> )
```

Table 3

The PG Procedures

ERROR(<SV>)	Prints <SV> as an error message on the listing file and sets an error flag.
WRITE(<SV>)	Prints <SV> on the normal output file.
FWRITE(<SV>)	Prints <SV> on the output file, using the Fortran continuation convention if the line is longer than 72 characters.
DONE	Sets a flag indicating normal completion.
CLEAR	Resets PG so that all text matched so far is deleted. THis must be done, usually after each input program segment is finished, to keep the amount of text stored by PG within set limits.
IGNORE	The text matched by the current rule is cleared. When control returns to the rule for which the current rule is a subrule, it will appear that the subrule succeeded but the corresponding string $<N> is null.
REPLACE(<SV>)	The text matched by the current rule is replaced by <SV>. <SV> may contain text matched in the current rule. This command allows the text to be rearranged. For the rest of the action in with this command appears only $1 is defined and it refers to <SV>. When control returns to the calling rule the corresponding $<N> will have the value <SV>.

FAIL — Causes the current rule to fail and control to return to the calling rule without trying any more alternatives of the current rule. This can be used, for example, to write a rule which matches anything but a '$' or an end-of-line marker:

```
NOTDOL    ->   '$'
                    FAIL
          ->   ANY
```

Conditional execution of action statements is possible using IF-ELSE-ENDIF. It has the form

```
IF ( <LOGICAL-EXPRESSION> )
     any number of action statements
ELSE
     any number of action statements
ENDIF
```

The ELSE clause may be omitted. IF-ELSE-ENDIF statements may be nested.

PG also has a DO-loop construct. It has the form

```
DO  <I>  =  <FEXPR1>, <FEXPR2>, <FEXPR3>
     any number of action statements
ENDDO
```

<I> is a Fortran integer variable and the <FEXPRn> are Fortran integer expressions. As in Fortran <FEXPR3> may be omitted; it defaults to one. DO loops may be nested and may include IF statements.

There are several Fortran routines provided for use in PG action statements. In all cases their arguments are string variables. They are summarized in table 4.

Table 4

Utility Routines for Use in Actions

Routine Name	Purpose
IQNO1 (<SV>)	Integer function returning -1, 0, or 1 if <SV> is a Fortran expression equal to -1, 0, 1, respectively; otherwise, it returns 2.
IQIVAL (<SV>)	Integer function returning the value of the string <SV> (as an integer).
LQMTCH (<SV1>, <SV2>)	Logical function whose value is .TRUE. if <SV1> is identical to <SV2>.
QQADLS (<SV1>, <SV2>)	Subroutine which adds the Fortran name <SV2> to a comma separated list of names in <SV1>, if <SV2> is not already on the list. This is intended for maintaining a lists of variables to

be used in a Fortran declarations.

4. Special Segments

A PG grammar must include several special segments. The first of these is the (optional) GLOBAL segment in which Fortran declaration statements may be included. If it is used it must be the first segment in a PG grammar. Each PG grammar rule corresponds to a Fortran subroutine. The declarations in the GLOBAL segment are included in all of these subroutines. Variables which are referenced in actions associated with different rules must be in a COMMON block. It is also possible to declare the types of variables and arrays. If arrays are declared they should be in a COMMON block, or else, since they will be declared for each rule, an excessive amount of memory may be needed.

The GLOBAL segment is also used to set the sizes of various arrays used by PG to other than their default values. The list of these arrays, and the template variables controlling their sizes, is given in Table 5.

Table 5

Template variables that control the size of PG internal arrays.

Name	Default	Size of
IQSPMX	1000	CQSTCK (the primary stack)
IQMXIN	2000	CQBFIN (the input buffer)
ICBDIM	1500	input buffer
ICSDIM	1000	character storage array
IHADIM	401	hash table (must be prime)
ISTDIM	6000	integer pointer pool (must be divisible by 3)

An example of a GLOBAL segment is

```
GLOBAL.
     COMMON /ABC/  X, Y, Z
     INTEGER Y
     REAL Z(10)
     $(IQSPMX) = 2000
```

The second special segment is MAIN. It is a required segment. It contains actions which are executed in the generated main routine. Usually it includes statements to initialize template variables and a call to a (separately compiled) Fortran routine to open files, etc.

To execute a grammar rule from the actions under MAIN or from a Fortran subprogram use

```
CALL <RULE>
```

The ELLPACK grammar (and the PG sample grammars in this document) use the rule PROG as the base rule, which causes all of the other rules to be executed. Hence exactly one CALL PROG statement is needed to cause an entire program to be parsed.

In the separately compiled Fortran routine, to open the input file set IQINPF in the common block CQCINT (see section 5) to the Fortran unit number and OPEN the file. Similarly, the listing file unit number is given by IQLSTF and the file for the generated code by IQOUTF. The template processor is invoked by the call

```
CALL TPDRV ( IN, ILIST, IOUT )
```

where IN, ILIST, IOUT are the Fortran unit numbers of the input, listing and output files, respectively.

The last segment must be END. It has not values and no actions.

A complete example of a simple PG grammar is:

```
GLOBAL.
     COMMON /ABC/ X, Y, Z
     INTEGER Y
     REAL Z(10)
     $(IQSPMX) = 2000
MAIN.
     $I(COUNT) = 0
     CALL PPINIT
     CALL PROG
PROG      ->   'OPTION.'  ALFNUM+  '='  ALFNUM+
                    $($2) = $4
                    $I(COUNT) = $I(COUNT) + 1
          ->   'OPTION.'  ALFNUM+
                    $L($2) = .TRUE.
                    $I(COUNT) = $I(COUNT) + 1
END.
```

5. PG Internals

In this section we give a description of the data structures used by PG generated programs.

Most PG routines manipulate a character stack stored in CQSTCK. Character strings are pushed onto the stack using the routine QQPUSH. Strings are removed from the stack by resetting the pointer IQSP. if the rule fails. The action CLEAR resets the pointer to 1.

```
/QQCSTK/  IQSP      pointer to next free location in CQSTCK
          IQSPMX    dimensioned size of CQSTCK
```

/QQCCST/	CQSTCK	matched characters

All PG routines get their input through the routine QQGTCH. It maintains a circular buffer of the characters read. The data structures for this buffer are

/QQCINP/	IQBFLG	for use by input routine
	IQNXIN	pointer to next character in CQBFIN to be returned
	IQINB1	pointer to first valid character in CQSTCK
	IQINB2	pointer to last valid character in CQSTCK
	IQMXIN	dimensioned size of CQBFIN
/QQCBIN/	CQBFIN	input characters

The pointer IQINB1 is moved only the the CLEAR action. The characters between IQINB1 and IQNXIN have been tentatively matched; they can be put back into the input queue by adjusting IQNXIN. PG built-in and generated rules save the pointer IQNXIN on entry. If the rule fails then IQNXIN is reset to its initial value.

When IQNXIN reaches IQINB2 the routine QQINPT is called to input the next line. It calls QQREAD to get the next line, prints the line, skips lines beginning with a comment character and adds the end-of-line marker if the previous line did not end with the continuation character. QQREAD calls UTIORD to get the line and strips trailing blanks. All input for PG programs (including input for the template processor) is done though UTIORD. The data structures used by these routines are

/CQCINT/	IQLEVL	output listing level
	IQINPF	input file unit number
	IQOUTF	output file unit number
	IQLSTF	listing file unit number
	IQCEOL	number of characters used by end-of-line marker
/CQCCHR/	CQCEOL	end-of-line marker
/QQCCIN/	CQCOMT	comment character
	CQCNTN	continuation character
/QQCLOG/	LQERR	.TRUE. if the current rule has failed
	LQEOL	.TRUE. if current character is an end-of-line
	LQFATL	.TRUE. if a fatal error is flagged by the ERROR() action statement

6. Installing PG

The PG distribution tape contains the following files:

1. CSD-TR 405
2. PGLIB1.F
3. PGLIB2.F
4. PGDRIV.F

5 PG.F
6 SYMBOLS
7 PG.G

The files PGLIB1.F and PGLIB2.F together are a library of Fortran routines needed by PG and by the programs generated by PG. They should be compiled and made into a single object module library called PGLIB.

The file PGDRIV.F contains the OPEN statements needed by PG. It may need to be changed on some systems. The file PG.F contains the main PG routines and was itself generated by PG. PGDRIV.F and PG.F should be compiled and link-edited with PGLIB to produce the executable file PG.

The file SYMBOLS is used by PG when it executes. (An OPEN statement in PGDRIV.F refers to it). The file PG.G is the grammar that was used to produce PG.F. Executing PG with PG.G as input should produce PG.F as output.

References:

J. Brophy, PG - A preprocessor generator. Computer Science Dept., Purdue University, CSD-TR405, 1982.

J. Rice and W. Ward, A simple macro processor, Computer Science Dept., Purdue University, CSD-TR400, 1982.