Paper 058-29

Complementing SAS® with Perl
Updating, creating, running, and checking SAS® code

Marcelo Coca-Perraillon, Adheris Inc., Woburn, MA
Matthew P. Lambert, Adheris Inc., Woburn, MA

ABSTRACT

This paper describes a way to complement the SAS macro system using Perl. SAS macros can save a considerable amount of typing since they can generate SAS code. Perl can save even more tedious, repetitive typing and clicking since it can generate macro statements. Suppose, for example, that one needs to create reports and statistical analyses for many experimental protocols. Although the original SAS program uses macros, one still needs to define several macro parameters and variables before running each report. The variables and parameters contain dates, titles, output paths, conditions for data steps, and so on. Further, there are three other constraints. First, the code must be flexible and easy to update. Second, a SAS code file for each report needs to be saved in order to perform further analyses and debugging. Finally, one has to import parts of the SAS output into another application. A short Perl script not only can accomplish all these tasks simultaneously, but it can also create the necessary code to run all the reports in batch mode and scan the log files for errors.

INTRODUCTION

In many practical situations, one has to perform the same analysis using different data sets or different variables. The macro system is convenient in these situations since macros generate SAS code. In general, running a program that uses macros involves changing the values of macro variables and parameters. This task is tedious when an analysis not only involves changing many parameters, but also requires updating the same parameters in many files. It gets worse when one has to import the SAS output into another application, such as Microsoft's Excel. Even if the changes are minor, the SAS code files need to be opened to make changes and then saved. Repeating this process many times is extremely tedious and one is prone to make mistakes.

This paper describes a Perl script that can be used to do most of the typing and all the clicking. Besides updating files, the script can write code to run SAS in batch mode. With a few more lines, it can be used to import SAS output into another program. After running the SAS code files, another script can scan the log files in search of error and warning messages. In order to explain in detail the basic features of the Perl scripts, this paper uses a very simple example in which there is only one SAS program, a "template," that needs to be updated. The objective is to create two SAS code files that are run in batch mode and a summary report of error and warning messages.

In order to change the values of macro variables and parameters, the script searches for them in a file. Once their values are located, they are used to write macro statements and create SAS code files. The values (dates, titles, etc.) are all centralized in a comma-separated file. Having them centralized not only makes the updating easier but it also helps prevent mistakes. For example, suppose that one needs to run the same analysis every month. Once a month, a date used as a condition in a data step needs to be changed, from, say, 01JAN2003 to 01FEB2003. If the date 01JAN2003 is in a data file, then one could use SAS (or Excel) to update it instead of typing 01FEB2003. While this seems trivial, letting SAS change values prevents typos and arithmetic mistakes.

The first section provides and overview of how the Perl script works and describes the input and output files. The details of the script are explained in the next section. The third section contains a brief explanation of how to use Perl to manipulate the values of the macro variables and parameters before they are used. It also describes how to import SAS output into Excel and other programs. The last section describes another Perl script that reads log files and produces a report of errors messages.

PERL SOLUTION OVERVIEW

The script uses two input files to create two SAS code files. One file is a comma-separated file that has the particular information needed to run a report. In this example, the file is called "definitions.csv" and has three variables: name, id, and startdate, respectively:

```
ReportA, 399, 01JAN2003
ReportB, 400, 01FEB2003
```

These variables identify a report and are used to define a libname path, titles, and conditions in a data step.

The other input file is a template of a SAS program called "TemplateSASCode.txt:"

```
/* Insert definitions below */
%titledate = %substr(&startdate,4,7);

data example.filtered&name;
   set unfiltered;
   if varid = &id and date > &startdate;
run;

%KernelEst(title= "Kernel for &name - &titledate", data = example.filtered&name);
```

The Perl script uses these files in the following way: it reads "definitions.csv" line by line. The script then breaks down each row in three parts, assigning each part to a Perl variable. The Perl variables are then used to write macro statements. For example, after reading the first row of the file "definitions.csv," the script creates the following macro statements:

```
libname example "D:\SUGI29\OutputReportA";
%let name = ReportA;
%let id = 399;
%let startdate = '01JAN2003'd;
```

These statements are inserted into the template file “TemplateSASCode.txt,” below the commented line `/* Insert definitions below */`. The modified template is saved as "ReportA.sas:"

```
 libname example "D:\SUGI29\OutputReportA";
%let name = ReportA;
%let id = 399;
%let startdate = '01JAN2003'd;
%titledate = %substr(&startdate,4,7);

data example.filtered&name;
   set unfiltered;
   if varid = &id and date > &startdate;
run;

%KernelEst(title= "Kernel for &name - &titledate", data = example.filtered&name);
```

The same steps are repeated in order to create a file called "ReportB.sas:"

```
 libname example "D:\SUGI29\OutputReportB";
%let name = ReportB;
%let id = 400;
%let startdate = '01FEB2003'd;
%titledate = %substr(&startdate,4,7);

data example.filtered&name;
   set unfiltered;
   if varid = &id and date > &startdate;
run;

%KernelEst(title= "Kernel for &name - &titledate", data = example.filtered&name);
```

The SAS files do not do anything too interesting and the definition of the macro %KernelEst has been omitted. Nonetheless, it shows how the input file is used to write SAS statements. Note, for example, that the date in the first row of the input file is 01JAN2003. However, the script writes '01JAN2003'd to define the macro variable &startdate. Also, the SAS function %substr extracts the month and year of &startdate in order to define another macro variable called &titledate. The same (and more) can be achieved using Perl, as it is explained in the third section of this paper.

Instead of manipulating the date, one could include more variables in “definitions.csv.” For instance, a fourth column could be '01JAN2003'd. However, it is simpler to include the date only once and keep it in a format that is easier to update.

After creating the SAS code files, the script writes a "BAT" file called "RunReports.bat," which contains the following command lines:

```
D:\SAS\V8\SAS.EXE -SYSIN D:\SUGI29\ReportA.sas -LOG D:\SUGI29\ReportALog.log
D:\SAS\V8\SAS.EXE -SYSIN D:\SUGI29\ReportB.sas -LOG D:\SUGI29\RerportBLog.log
```

Files with the extension BAT are executable in Windows, and can be run by double clicking on them. To run SAS in batch mode on other platforms, the command lines need to be modified (see the SAS documentation for details). In addition, the command lines above assume that "sas.exe" is in the directory "D:\SAS\V8\." Regardless of the specific commands, the idea is to use Perl to write them.

In summary, the script uses two input files to create two SAS code files and one BAT file. The number of SAS code files depends on the number of rows in the file "definitions.csv." If "definitions.csv" had one hundred rows, then the script would create one hundred SAS code files that can be run by double clicking on "RunReports.bat," which would also have one hundred lines.

THE SCRIPT

Below is the complete script with the lines to facilitate the explanation that follows. The description of the script is divided into three parts. The first part explains how to read the input file “definitions.csv” line by line. The second part explains how to use the template “TemplateSASCode.txt.” Finally, the last part describes the code that writes the BAT file “RunReports.bat.”

```
1   use warnings;
2   $csvinput = "definitions.csv";
3    $SASTemplate =  "TemplateSASCode.txt";
4
5   open(BATFILE, "> RunReports.bat") || die("Can't open RunRerpots.bat $!");
6   open(CSVINPUT, "< $csvinput") || die("Can't open $csvinput $!");
7   while(<CSVINPUT>) {
8      chomp;
9      @var = split(/,/, $_);
10     $Name = $var[0];
11     $ID = $var[1];
12     $StartDate = $var[2];
13
14     $SASFileName = "$Name".".sas";
15     open(SASCODE, "> $SASFileName") || die("Can't open $SASFileName $!");
16
17     open(SASTEMPLATE, "< $SASTemplate") || die("Can't open $SASTemplate $!");
18     while(<SASTEMPLATE>) {
19       print SASCODE;
20       if (/\/\* Insert code below \*\//) {
21  print SASCODE <<eof;
22  libname example \"D:\\SUGI29\\Output$Name\";
23  %let Name = $Name;
24  %let ID = $ID;
25  %let StartDate = \'$StartDate\'d;
26  eof
27       }
28   }
29  close(SASTEMPLATE);
30  close(SASCODE;
31
32  print BATFILE "D:\\SAS\\V8\\SAS.EXE -SYSSIN $SASFileName -LOG $Name"."Log.log
      \n";
33  }
34  close(CSVINPUT);
```

READING THE INPUT FILE

Line 1 is a pragma that instructs the Perl compiler to provide additional warning messages. Although it is not necessary, it facilitates debugging Perl code.

Line 2 and 3 define two Perl variables that have the names of the input files. The symbol $ is used to declare a scalar variable. Array variables are named with an initial `@`.

Line 5 creates the file "RunReports.bat" in writing mode. The symbol > specifies how a file is accessed. Among other mode indicators, > is used for writing, < for reading, and >> for appending. If the file cannot be opened, the function `die` issues an error message. The BAT file needs to be created before reading “definitions.csv,” for reasons that will become apparent later. For the time being, it is better to focus on the next line.

Line 6 opens the file defined by the variable `$csvinput` and assigns it the filehandle `CSVINPUT`. By default, Perl reads a file line by line.

Line 7 starts a `while` loop that processes each line. The condition for the loop is given by the “diamond” or “angle operator” <>, which returns the next line from the associated filehandle. So the condition is valid while there is another line in the file `CSVINPUT`. By default, each line (row) of `CSVINPUT` is saved in the scalar variable `$_` , which is used in Line 9.

Line 8 removes the trailing new-line marker at the end of the line being read through the function `chomp`. Since there is no argument, `chomp` removes the marker from the default scalar `$_`.

In line 9, the function `split` breaks down the line `$_` into three parts, which are separated by commas. Again, `$_` could be omitted. The separator is specified inside the slashes and could be any character. The divided line is assigned to the array `@var`. As in C, the first element of an array is indexed with a zero.

Line 10 to 12 assign each element of the array to the scalar variables `$Name`, `$ID`, and `$StartDate`, so it is easier to identify them. When the `while` loop reads the first line, for example, the values of `$Name`, `$ID`, and `$StartDate` are ReportA, 399, 01JAN2003, respectively.

Line 14 uses the scalar variable `$Name` to define another variable that has the name of the output SAS code file. When reading the first line of CSVINPUT, `$SASFileName` is equal to “ReportA.sas.” The dot separating `$Name` and `.sas` is a concatenation operator.

Line 15 uses `$SASFileName` to open the SAS code file in writing mode.

USING THE TEMPLATE

Line 17 opens the template for reading (the template's name was defined in Line 3). Note that the script is still processing one line at a time. Therefore, the template is opened once for every row of `CSVINPUT`.

Line 18 starts processing the template line by line through a `while` loop.

Once the script reads the first line of `SASTEMPLATE`, it immediately prints it to the file `SASCODE` (Line 19). In effect, the script is “copying” each line of the template to the file `$SASFileName` (“ReportA.sas” or “ReportB.sas”).

Line 20 tests whether the line being read is `/* Insert code below */`. This is where the script uses the basics of “regular expressions.” Regular expressions are how patterns are specified. Perl has a wide range of regular expressions, and this capability is what makes it a popular language in information technology and bioinformatics. However, the syntax of regular expressions is not easy to learn, and they look more like hieroglyphics than a computer language.

One of the main operators in pattern matching is the matching operator `m//` (the `m` is often omitted). The matching operator is used to define a pattern. The function `split` in Line 9 uses it to find a comma. The matching operator is also used in Line 20 to simply test whether a line in `SASTEMPLATE` matches the string `/* Insert code below */`. Searching for this SAS comment is straightforward since it is not really a pattern. However, Line 20 looks strange only because the character `/` and `*` have a function in regular expressions and need the escape character `\` before each of them. With an escape character, Perl understands that it needs to find a slash and an asterisk. In other words, `\/` matches a slash, while `\*` matches an asterisk.

Once Perl finds the line, it writes the macro statements in line 22 to 25 to the `SASCODE` file. The macro statements are written using the variables defined in lines 10 to 12.

Line 21 uses the "here-document" structure so the `print` function prints lines exactly in the same way they are written. Since everything below Line 21 is printed as it appears, indenting Lines 22 to 22 would result in the lines being indented in the SAS code files. Although not indenting the code is not very aesthetic, it is still better than using the print function in the usual manner. For instance, the alternative way of using the print function in lines 22 and 23 is:

```
print "libname example \"D:\\SUGI29\\OutputReportA\"; \n"
print "%let name = ReportA; \n"
```

(The extra backslashes are escapes while `\n` writes a new line.)

Line 29 and 30 close the files `SASTEMPLATE` and `SACODE`, but the script needs to write the BAT file before closing `CSVINPUT`.

WRITING THE BAT FILE

The file "RunReports.bat" was created in Line 5 and it is ready to be used. Note that it needs to be opened before starting to read the `CSVFILE` because there is only one BAT file, not one per line of `CSVFILE`. Also, it needs to be written before starting reading the next line of `CSVFILE`, since the BAT uses the value of the variable `$SASFileName`.

Line 32 writes the commands to run each report in batch mode and specifies the name of the SAS log file. For the first row of `CSVINPUT`, the log file is called "ReportALog.log."

Finally, `CSVINPUT` is closed in Line 34, after Perl has processed the entire file line by line.

Perl is a flexible language and one can do the same in different ways. In fact, one can often take shortcuts even in the syntax. For example, Line 6 could have been: `open CSVINPUT "< $csvinput" or die "Can't open $csvinput $!";` In turn, Line 9 could be much shorter: `@var = split /,/;` Flexibility can be handy sometimes, but it tends to be confusing when learning Perl. Lines of code that do the same can look very different.

Unless one needs to process thousands of long files, it is not necessary to worry about efficiency. But if efficiency is a consideration, the script could be improved. For example, the script opens the template every time it reads a line of "definitions.csv." Instead, it would be more efficient to open the template only once, saving the text before and after `/* Insert code below */` in two variables. These variables can then be used to write the SAS code files, instead of copying the template line by line.

ADDITIONS

The script described above can be adapted in several ways. First, the variables read from the file "definitions.csv" can be processed to alter their format or to extract some information from them. Second, the script can be extremely useful if one has to import SAS output into other programs, such as Excel or a C program. Finally, after running SAS in batch mode, it is easy, and always a good idea, to check the log files.

MORE PATTERN MATCHING WITH PERL

The SAS template uses the macro function %substr to extract the month and the year in a date. The same can be done using Perl by writing:

```
$titledate = substr($StartDate,2,7);  # variable $titledate is JAN2003
```

The month in the variable `$StartDate` can be extracted with:

```
$StartDate =~ /\D+/;   # variable $& is JAN
```

The operator =~ is used to find a pattern in the variable `$StartDate`. The pattern is specified by the regular expression `\D+` using the match operator `//`. The so-called meta-character `\D` matches any non-digit character, while the plus sign is a quantifier indicating that that one or more non-digit characters must be matched. Note that the operator =~ is not assigning the match to the variable `$StartDate`; it is searching a pattern in it. The match is assigned to the special variable $&.

Perl defines other variables after a successful match. For example, `` $` `` has the string that is to the left of the match, while `$'` has the string that is to the right. One could use these variables to change the format of the date or just print the date in a different way:

```
$OtherDate = "$`-$'";              # reformat the date to 01-2003
print "$`  $&  $' \n";             # prints: 01  JAN  2003
```

Sometimes it is necessary to save only one part of a string that fits a pattern. For example, the following code matches a string that starts with the word `SYMBOLGEN` (the ^ indicates that the match is at the beginning) and has a number anywhere afterwards:

```
$Message = "SYMBOLGEN: Macro variable AGE resolves to 21";
$Message =~ /^SYMBOLGEN.+(\d+)/;    # variable $1 equals 21
```

The variable $& has the entire string matched, but the number 21 is saved in the variable $1. The parentheses enclosing `\d+` are used to "capture" the number. If there were a second pair parentheses, the match would be saved in the scalar $2.

Another operator that comes in handy is `[]`, called “class operator.” For instance, `[abc]` matches a, b, or c. As another example `/[SYMBOLGEN|AGE]/` would match any string with either `SYMBOLGEN` or `AGE` in it.

Sometimes it is easier to find a string by specifying what is missing in that string. For example, `[^SYMBOLGEN]` matches any string that does not contain `SYMBOLGEN`. Note that `^` has a different function (it negates) inside a class operator.

EXPORTING SAS OUTPUT

Importing output into Excel can be accomplished by writing a VBA macro in Excel, which is often very simple. However, it would be time consuming to type the calls into the Excel macro. For instance, if the output of “ReportA.sas” and “ReportB,sas” were two HTML files called “ReportA.html” and “ReportB,html,” one could just use the Perl script to write the calls. The script could write them directly or insert them into a VBA template. In the example script, this can be done after Line 32, before closing the `CSVINPUT` file since the variables defined in `CSVINPUT` are still needed to write the name of the HTML files.

SCANING THE LOG FILES

A problem with running SAS in batch mode is that one cannot see the log files while SAS is executing. Furthermore, it is time consuming to read each log file. Suppose, for example, that we ran “ReportA.sas” with a typo in a data step. Instead of “set unfiltered;” the program had “set unfilteredw;.” After running SAS in batch mode, the file “ReportALog.log” looks like:

```
SYMBOLGEN:  Macro variable NAME resolves to ReportA
6
7 data example.filtered&name;
8    set unfilteredw;
ERROR: File WORK.UNFILTEREDW.DATA does not exist.
9   if varid = &id and date > &StartDate;
SYMBOLGEN:  Macro variable ID resolves to 399
SYMBOLGEN:  Macro variable STARTDATE resolves to '01JAN2003'd
10 run;

NOTE: The SAS System stopped processing this step because of errors.
WARNING: The data set EXAMPLE.FILTEREDREPORTA may be incomplete.  When this step was stopped
there were 0 observations and 2 variables.
WARNING: Data set EXAMPLE.FILTEREDREPORTA was not replaced because this step was stopped.
NOTE: DATA statement used:
      real time           0.01 seconds
      cpu time            0.01 seconds
```

(The line numbers can be matched to the “ReportA.sas” file in the enhanced editor in SAS.)

The following is a stand-alone Perl script that scans all the files with the extension “.log” in a directory. It searches for both error and warning messages. By using regular expressions, the script can be easily adapted to find any other message.

The script processes each log line by line. First, it tests whether a line starts with a number. If it does, it saves the number in a variable. If the line does not start with a number, the script checks if it contains the words “WARNING” or “ERROR.” If it does, then it copies the line to a summary report. The report also includes the last number saved. If more than two errors are found, the script stops scanning the current file and moves on to the next one.

In real applications, the limit should be more than two errors, but it is a good idea to have a limit. SAS continues processing a program even if there is an error, generating many more error messages that may not provide much additional information.

For this example, the script scans the log file of “ReportA.sas” (showed above) and that of “ReportB.sas,” which does not have any error message. The final output is a file called “ErrorReport.txt:”

```
----- ReportAlog.log -----
ERROR found after line 8: "ERROR: File WORK.UNFILTEREDW.DATA does not exist."

WARNING found after line 10: "WARNING: The data set EXAMPLE.FILTEREDREPORTA may be
incomplete.  When this step was stopped there were 0 observations and 2 variables."

**Note: More than 2 errors found.

----- ReportBlog.log -----
No ERROR or WARNING message found.
```

The complete script is:

```
1  use warnings;
2
3  # Read current directory, saves .log files in array @LogFiles;
4  opendir(DIRTOSCAN, '.') || die ("Cannot open current directory $!");
5  @LogFiles = grep(/\.log/, readdir(DIRTOSCAN));
6  closedir(DIRTOSCAN);
7
8  # Create summary file;
9  open(ERRORREPORT, "> ErrorReport.txt") || die("Can't open ErrorReport.txt $!");
10
11  # Scan each file in array @files;
12  foreach $File (@LogFiles) {
13     print ERRORREPORT "----- $File ----- \n";
14
15    #Read log file line by line;
16    open(LOGTOSCAN, "< $File") || die("Can't open $File $!");
17    $ErrorNumber = 0;
18
19    LINELOOP: while(<LOGTOSCAN>) {
20      chomp;
21      $Line = $_;
22      if (/^(\d+) .*/) {
23         $LineNumber = $1;
24      }
25      elsif (/(ERROR|WARNING)/) {
26         $ErrorNumber++;
27         # End scanning file if it has more than 2 errors;
28         if ($ErrorNumber == 3) {
29            print ERRORREPORT "**Note: More than 2 errors found.\n\n";
30            last LINELOOP;
31         }
32         print ERRORREPORT "$1 found after line $LineNumber: \"$Line\" \n\n";
33     }
34   }
35   print ERRORREPORT "No ERROR or WARNING message found. \n\n" if $ErrorNumber == 0;
36   close(LOGTOSCAN);
37 }
38 close(ERRORREPORT);
```

Lines 4 to 6 read the current directory and save all the files with the extension .log in the array `@LogFiles`. The function `opendir` is similar to the function open but `readdir` (obviously) reads a directory instead of a file. The argument `'.'` opens the current directory. The function `grep` returns all the elements for which the condition is true. Note that the condition is a regular expression enclosed by the match operator `//`.

Line 6 closes the directory since all the logs are saved in `@LogFiles`.

Line 9 opens the file "ErrorReport.txt" in writing mode.

Line 12 processes all the elements of the array `@LogFiles` thought the loop `foreach`. It passes each filename to the variable `$File`.

Line 19 is more familiar, with the exception of the `LINELOOP:`, which assigns a label to the loop. A name is assigned to the loop because this loop will be halted later.

Line 22 searches for a number at the beginning of the line. Since a line number can have more than one digit, the + is added to match one or more digits. The number must be followed by a space and any character can follow the space. The wildcard `.` (dot) matches any character. However, if a line consists of only a number, then there is no “any” character after the number. The asterisk after the dot needs to be added so the regular expression matches none or any number of characters. Note that + means at least one, while the asterisk means none or any.

Line 23 assigns the number contained in the line matched (which Perl saved in `$1)` to the variable `$LineNumber`.

Line 25 checks if the line has either an ERROR or a WARNING message. If it does, the error counter `$ErrorNumber` is increased by one in Line 26.

Line 28 checks if `$ErrorNumber` is equal to three. If it is, then the message `**NOTE: More than 2 errors found` is printed to ERROREPORT.

Line 30 stops the `while` loop started in Line 19. Since more than two errors were found, the script stops reading the log and moves on to the next log file.

Line 25 uses the variable $1 to print a message indicating what type of problem was found and where it was found (it was found below the last value of `$LineNumber`). The line being read, saved in the variable `$Line`, is also included in the report.

Finally, if the there were no error messages, as in “ReportBLog.log,” line 35 prints "No ERROR or WARNING message found."

This script can be easily adapted to find other messages. One just needs to modify the regular expression in Line 22.

CONCLUSION

This paper has shown how to use Perl programs to complement the SAS macro system. In essence, the script creates SAS code files and writes macro statements, which in turn create SAS code. A useful feature of the script is that it creates SAS code files that can be used later. Furthermore, since the script works with a template, one can perform a new analysis for every protocol by just changing the template and running the script again. When one is working with hundreds of protocols, Perl can save days of tedious work.

REFERENCES

Schwartz, R.L., and Phoenix, T. (2001), *Learning Perl*, Third Edition, O'Reily & Associates.
Wall, L., Christiansen, T., and Orwant, J. (2000), *Programming Perl*, Third Edition, O’Reily & Associates.

ACKNOWLEDGMENTS

We would like to thank Marta Arias for comments and Perl advice, and Michael A. Mace for several useful comments and suggestions.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact:

Marcelo Coca-Perraillon	Mathew P. Lambert
Adheris Inc.	Adheris Inc.
400 West Cummings Park	400 West Cummings Park
Suite 3050	Suite 3050
Woburn, MA 01801	Woburn, MA 01801
mcoca@adheris.com	mplambert@adheris.com