Paper T01-2009

Getting By with a Little Help from My Regular Expressions

Scott Davis, COMSYS, Portage, MI

ABSTRACT

Much has been written about the power and flexibility of regular expressions. Regular expressions represent an entirely new drawer in the programmer's toolbox that provide yet another way of attacking a problem and developing a solution.

Imagine a rather common scenario where, during a testing phase, output from the current run is compared to output from the previous run. If updates to the program do not pertain to a particular table, then the output from both runs should be the same. What about the logs though? Excluding things like dates, what happens when changes in the program result in vastly differing logs. Compare utilities cannot adequately match up these logs with the additional information. Enter some regular expressions along with metadata to help solve the comparison problem.

The goal of this paper is to explore a particular application of regular expressions (the comparison of vastly differing logs). This paper assumes a basic understanding of regular expressions.

INTRODUCTION

One of the less inviting aspects of a programmer's job can be running through a large suite of tests for a complex program or macro. A large macro, for example, can have hundreds upon hundreds of tests needed to ensure proper functioning. Often times, the intent of a test and its subsequent success or failure is simple: did the macro produce the intended output? or, is the orientation correct according to specifications?

When writing a large macro programmers usually want to provide the user with a lot of relevant information in the log. Things like alerts for invalid parameter values, usage of default values for parameters, or custom error messages can be very useful. They can also make a log very crowded, and difficult to wade through for relevant information.

A large-scale project will require multiple iterations of a test suite as refinements or enhancements are made to the original macro over time. As the size and complexity of a macro grows, so do the test programs, logs and output. To measure whether a particular test passes or fails is dependent on the intended outcome of the test along with a thorough check of the log for anything that is not expected or conversely is expected, but does not appear, e.g. one of the notice-type messages mentioned above. Even when talking about just one pair of logs it can be overwhelming to depend solely on the eye for checking that everything is as expected. This expands exponentially when talking about multitudes of logs to compare.

Well, we are programmers so we'll use a program to assist with the comparison, right? Of course, but exactly how do we compare logs when they are inherently different yet should contain much of the same information? It's easy to strip away line numbers and dates, but how we categorize the remaining messages is what we will focus on in this paper.

REGULAR EXPRESSIONS

Regular expressions have been around for some time and exist in many different languages. One of the more popular flavors is Perl regular expressions. These are relatively new to SAS. Introduced in version 9, the Perl regular expressions augmented the regular expressions SAS had implemented in version 6.12 (Cody, 2006). Many programmers have opted to learn Perl regular expressions as they can easily port them to other applications. The regular expressions give you the ability to find patterns in text strings. The power of the regular expression lies in its ability to match many different kinds of patterns with relatively little code.

For example, let's say your macro produces a warning and/or an error message when invalid values are supplied for two variables, firstdate and lastdate. You could use the SCAN function with 'd' as a delimiter, but that would break as soon as you decided to look at a variable named dosedate. The text string we're interested in would look something like this:

WARNING: firstdate=<value>…

or

ERROR: lastdate=<value>…

The regular expression used to capture either one of these situations, up to the equal (=) sign, would look like this:

/[A-Z]+: \w+date=/

where:

'/' (forward slash) at the beginning and end of the expression serve as the boundaries for the expression,

'[A-Z]' will capture the uppercase characters specified in the range (in this case, all of them)

'+' (plus sign) indicates to repeat the "process" specified prior to the '+'

'\w' (backslash 'w') will match any alphanumeric character including an underscore. '[a-z]' could also be used since there is no underscore present in the search string.

SO . . . 'first' (5-character string) or 'last' (4-character string) will both match if they are immediately preceding the word 'date'

REGULAR EXPRESSION METADATA

To facilitate my log checking program I needed some additional fields on top of the actual regular expression. I created some fields that are informational in nature, and I added some flags to help with processing later on. I chose to create my metadata in Excel mainly for ease of adding records and updating fields. The Excel libname engine makes it very easy to bring in the metadata to create a SAS dataset. By enclosing the regular expression in parentheses you create a capture buffer that can be used to retrieve the enclosed text. Multiple sets of capture buffers are then referenced in the order they appear by a number, 1 for the first capture buffer, 2 for the second and so on.

variable	**description**
prx	the actual Perl regular expression
type	ERROR, WARNING or other
category	description of message for grouping
value	'null' or numeric capture buffer
orderid	grouping id
last	flag for last message of group
errflag	flag to denote 'unique' error message
repeatflag	flag for repeating value variable

Table 1. List of metadata variables

The *type*, *category* and *orderid* variables are used mainly for grouping the variables. They can be used for special processing if the need arises, but the flags are more relevant in processing the message once a match has been made. The use of the flags will be covered in more detail in the program section.

Depending on the need, it can be a little tricky to create the regular expression for maximum effect. It's great to make the regular expressions as reusable as possible, but it's important to avoid making them impossible to understand. After spending some time with regular expressions it's easy to see how complex they can become. A regular expression that is designed to 'do' too much can quickly became hard to debug and while accomplishing a new and complicated match it is possible to "achieve" the adverse effect of 'breaking' a match that had worked earlier.

PROGRAM

Once the metadata is in place, a regular expression needs to be compiled. The SAS function used to do that is PRXPARSE. The compilation only needs to occur once in order for the regular expressions to be available for the program. One approach is to use the automatic variable, _N_, to the compiling once at the beginning of the data step. It is possible to include multiple regular expressions to parse by reading them into a temporary array. Using the temporary array also simplifies the code and alleviates the need to use a retain statement. Here's what the initialization looks like:

```
data matches(drop=_:);
      if (0) then do;
        set regex_metadata (drop=prx);
      end;

      *Perform regex compiling once;
      if (_n_=1) then do;

         *Create temporary array to retain values of regex id values;
         array _prxID {numRegex} 8 _temporary_;
         *Loop through metadata to parse each regex;
         do _currTest=1 to _numRegex;
            set regex_metadata (keep=prx rename=(prx=_prx))
                               point=_currTest nobs=_nPRX;
            _prxID {_currTest} = prxparse(_prx);
         end;
      end;

      set logMessages;
run;
```

When a match is found there is further processing done depending on certain factors in the metadata. One of the things that happens based on the data is related to groups of messages. There may be a message spans several lines in the log and also repeats for multiple parameters. For example, a default value message may look something like this:

NOTIFICATION: The macro parameter XXXXX does not have a specified value,

NOTIFICATION: as a result, the default value of YYYYY will be used.

Regardless of the number of parameters, the above message may be repeated multiple times if the default parameter values are chosen. This can pose a problem for the log comparison because there isn't anything to tie these two lines together for each unique pair that appears in the log. Here's where the variables *value*, *last,* and *repeatflag* are used. The table below represents a relevant section of the metadata:

prx	value	last	repeat flag
/NOTIFICATION: The macro parameter ([^]+) does not have a specified value/	$1	0	1
/NOTIFICATION: as a result, the default value of ([^]+) will be used/	Null	1	1

Table 2. Metadata values used for repeating values

The program will take the name of the parameter captured as *value* in line 1 and repeat that value for line 2. This will have the desired effect of uniquely identifying groups of like messages. You will now have an easy way of merging your current run and previous run datasets with the variable *value* being part of a distinct (or unique) key.

One of the more important flags in the metadata is the *errflag*. Usually we see some 'standard' SAS error messages regardless of whether we have constructed a detailed error/abort message. SAS likes to tell us things like, '"Ooh, I

can't believe you did that, I am going to print the error on a lot of pages". Of course, in reality it's a much more polite message like 'Errors printed on pages, 1, 2, n.' These messages need to not be grouped with other error messages that our macro generates since they will likely appear at different times in our test suite and should remain independent from our messages. These standard messages will have *errflag* set to 0. There is a routine within the program to look at whether a group of error messages have already been identified in the metadata and should be compared or if they actually represent a new set of error messages. Error messages that are not these generic, SAS supplied messages get an *errflag* value of 1 when they are added to the metadata. The routine simply checks to see if there is a positive integer when a group of error messages is added together. If the sum is zero then it is clear that the error messages that are appearing in the log are new and need to be added to the metadata. The existing standard messages with the errflags set to 0 are simply deleted and removed from the comparison for those new messages.

After the regular expressions have been parsed and applied to the input data the program becomes very simple. The dataset with the metadata applied will look like this:

prx	logMessage	type	category	value	order id	last	err flag	repeat flag
/NOTIFICATION: The macro parameter ([^]+) does not have a specified value/	NOTIFICATION: The macro parameter REPTFMT does not have a specified value	NOTIFICATION	defaulted parameter	REPTFMT	1.1	0	.	1
/NOTIFICATION: as a result, the default value of ([^]+) will be used/	NOTIFICATION: as a result, the default value of PDF will be used	NOTIFICATION	defaulted parameter	REPTFMT	1.2	1	.	1
/NOTIFICATION: The macro parameter ([^]+) does not have a specified value/	NOTIFICATION: The macro parameter REPTLAYOUT does not have a specified value	NOTIFICATION	defaulted parameter	REPTLAYOUT	1.1	0	.	1
/NOTIFICATION: as a result, the default value of ([^]+) will be used/	NOTIFICATION: as a result, the default value of LANDSCAPE will be used	NOTIFICATION	defaulted parameter	REPTLAYOUT	1.2	1	.	1
/ERROR: The parameter ([^]+), is required. The macro will abort/	ERROR: The parameter INPUTDATA, is required. The macro will abort	ERROR	required parameter	null	2.1	.	1	.
/ERROR: Errors printed on page(\w)? ([^]+)/	ERROR: Errors printed on pages 1, 4	ERROR	errors printed on page(s) n	null	3.1	.	0	.

The log that was previously hard to categorize can now be joined with another log using a simple merge and a by statement. The merge can be done to output log messages found in the current test run and not in the previous run and vice versa.

CONCLUSION

Learning to use Perl regular expressions gives programmers more options to write lean, powerful code. As the scale of a project increases, the use of Perl regular expressions can become invaluable when attempting to find patterns in the text output (be it a log or actual report output). The addition of flag and grouping variables to metadata allows for even greater functionality. Using some creativity, the possibilities are boundless for ways to accomplish the job.

The task at hand can seem rather daunting, but given some time there is a solution out there. It is helpful to share coding problems with colleagues because someone just may have a solution that hadn't come up before.

REFERENCES

Ron Cody, “An Introduction to Perl Regular Expressions in SAS 9”. http://www2.sas.com/proceedings/sugi31/110-31.pdf

ACKNOWLEDGMENTS

Special thanks to Jack Fuller for suggesting the use of and mentoring me with Perl regular expressions and to Rosie Grzebyk for her valued assistance in reviewing this paper.

RECOMMENDED READING

SAS 9.2 Language Reference: Dictionary – Functions and Call Routines –

Using Perl Regular Expressions in the DATA Step

All PRXnnn and CALL PRXnnn functions

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Scott Davis
COMSYS
5220 Lovers Lane, Suite 200
Portage, MI 49002
Work Phone: 269-553-5122
E-mail: sdavis@comsys.com