PharmaSUG 2021 - Paper QT-134

Using Regex to Parse Attribute-Value Pairs in a Macro Variable

Rowland Hale, Syneos Health

ABSTRACT

In some circumstances it may be deemed preferable to pass multiple values to a macro not via multiple parameters but via a list of attribute-value pairs in a single parameter. Parsing more complex parameter values such as these may seem daunting at first, but regular expressions help us make light of the task! This paper explains how to use the PRXPARSE, PRXMATCH and the lesser known PRXPOSN regular expression functions in SAS® to extract in robust fashion the values we need from such a list. The paper is aimed at those who wish to expand on a basic knowledge of regular expressions, and although the functions are applied and explained within a specific practical context, the knowledge gained will have much potential for wider use in your daily programming work.

INTRODUCTION

The author occasionally comes across seasoned and capable SAS programmers who haven't taken the plunge into the powerful world of regular expressions. Not so surprising, given that regular expressions look, well, downright scary, don't they? For sure, regex patterns *can* appear complex, but, on the other hand, when you see a series of three (or more!) "normal" (i.e. non regex) string handling functions, all mind-bogglingly nested on a single line of code (and, if in a macro context, each further buried inside a %SYSFUNC macro function) and realise that this can all be replaced with a relatively simple regular expression, you may start to wonder which is the easier to deal with.

Use case

The author was recently asked by his client to write a particular macro, and thanks to the potentially large number of input values required by the macro, the initial result included a correspondingly large number of parameters. So the decision was taken to consolidate certain parameters into a smaller number of what we called "options" parameters, each one taking a list of (mainly) "attribute=value" pair options, with each attribute replacing one of the "old" parameters. Although such an approach, with its less intuitive parameter value syntax, could be seen as impeding the user-friendliness of the macro, this counter argument was seen as marginal given that the macro relies anyway on default attribute values most of the time. In other words, it is *only* when the user wishes to override the default attribute values that he or she has to start thinking about the entire syntax of the attribute definitions as required by these options parameters. All that remained was to figure out how best to handle these parameters, and regular expressions were deemed to offer a sensible solution.

What is a regular expression?

A regular expression can be defined as a sequence of literal, wild card and other special characters which describe a text pattern used for searching, replacing and extracting text within or from a source string of characters. Thankfully SAS provides us with a nice set of functions and call routines for applying Perl regular expressions in our SAS programming, all named PRX*, and we will be using three of these functions, PRXPARSE, PRXMATCH and PRXPOSN, to handle our options parameters.

Audience, objective

The paper assumes an understanding of the pattern matching concept and a basic knowledge of Perl regular expressions, including wildcards and metacharacters. Whilst the context used to illustrate the technique is quite specific, the aim of the paper is to make the reader familiar enough with the concept to apply it in many other situations, not only in macro code but in a data step also.

THE FUNCTIONS PRXPARSE, PRXMATCH AND PRXPOSN

The function which we will ultimately use to return the required values from our options parameters is PRXPOSN. Note that this function cannot be used in isolation – in order to return what is needed, namely a specific portion of a match, PRXPOSN requires the following: a compiled regular expression (the task of PRXPARSE), and a successful match based on that regular expression (the task of PRXMATCH). Here's a description of the functions:

- PRXPARSE compiles a regular expression for subsequent, and likely repeated (as in our case), use in other PRX* functions or call routines. The function returns an ID which can be regarded as a pointer to the compiled regular expression.
- PRXMATCH applies a regular expression to a source string and returns the position in the source string of the match if a match is found, or zero if no match is found. The regular expression is passed to the function either as a literal regular expression or as an ID returned from a previous call to PRXPARSE.
- PRXPOSN extracts all or part of the match found by the PRXMATCH function. The extracted text is known as the capture buffer. Capture buffers are specific parts of a match as defined by enclosing the relevant part of a regular expression in parentheses. Multiple capture buffers can be defined and specific capture buffers are returned by passing their ordinal number in the regular expression as the second argument to PRXPOSN. Note that parentheses are used in regular expressions not just for the purposes of defining capture buffers specifically but also for the logical grouping of regular expression components, so care is needed to specify the second argument for PRXPOSN correctly, particularly where nested parentheses are present, as PRXPOSN regards all pairs of parentheses as defining capture buffers.

Table 1 summarises the arguments and return values of the three regex functions used:

Function	Arg 1	Arg 2	Arg 3	Return value
PRXPARSE	Perl regex	*n/a*	*n/a*	Regex ID (pointer to compiled regex)
PRXMATCH	Perl regex or regex ID	Source string (constant, variable or expression)	*n/a*	If pattern matched, position of match, otherwise 0
PRXPOSN	Regex ID	Capture buffer number (if 0 returns entire match)	Source string	Contents of capture buffer (subject to previous match)

Table 1. Arguments and return values of the PRXPARSE, PRXMATCH and PRXPOSN functions

Note that PRXPOSN works not only in conjunction with PRXMATCH to return a capture butter, but also with PRXSUBSTR, PRXCHANGE and PRXNEXT. These latter three functions are not considered in this paper – please see SAS online documentation for further information.

HANDLING A SIMPLE SET OF ATTRIBUTE-VALUE PAIRS

A so-called options parameter of the type described in the introduction may, as an example, take the following form:

```
MARKER_OPTIONS = var=<variable> symbol=<symbol> size=<size>
```

The first task is to define a regular expression to match each attribute-value pair which may be specified in the parameter. We'll use the "var=<variable>" pair as an example to work with:

```
%let _rxid1 = %sysfunc(prxparse(/var\s*=\s*([_a-z]\w{0,31})/i));
```

Note that we wish to extract just the value from the attribute-value pair. Placing the "value" portion of the regular expression within parentheses defines this as a capture buffer (highlighted in yellow). Table 2 explains each component of the regular expression:

Regex component	Explanation
/	Start of regular expression
var	The literal *var*
\s*=\s*	0 or more spaces, then an equals sign, then 0 or more spaces (permitting spaces like this within a match allows the user more flexibility)
(	Start of capture buffer
[_a-z]	First (or only) character of a valid SAS variable name (underscore or letter)
\w{0,31}	Between 0 and 31 subsequent characters of a valid SAS variable name ("\w" is the regex wildcard for word character (alphanumeric plus underscore)
)	End of capture buffer
/	End of regular expression
i	Match to be case insensitive

Table 2. Explanations of regex components

Now let's see if there's a match in our MARKER_OPTIONS parameter (here shown as a normal macro variable), and if there is, extract the capture buffer we defined into the variable MARKER_OPTION_VAR:

```
/* The parameter */
%let marker_options = var=marker symbol=squarefilled size=7;

/* Compile the regex to extract the variable name */
%let _rxid1 = %sysfunc(prxparse(/var\s*=\s*([_a-z]\w{0,31})/i));

/* Is there a match? */
%if %sysfunc(prxmatch(&_rxid1, &marker_options)) > 0 %then %do;

   /* Extract the variable name (capture buffer 1) */
   %let marker_option_var = %sysfunc(prxposn(&_rxid1, 1, &marker_options));
%end;

%else %do;
   /* Code to handle a non-match if required */
%end;
```

As expected, MARKER_OPTION_VAR will now contain the value portion of the required attribute-value pair, namely the word "marker" (and again appropriate code should be written to handle the possibility of no match). Regarding the remaining attribute-value pairs (*symbol* and *size* in our example), simply repeat the above with an appropriate regular expression for each.

Note that for good order a compiled regular expression should always be released from memory as soon as it is no longer needed. This is done as follows:

```
%syscall prxfree(_rxid1);
```

HANDLING A MORE COMPLEX SET OF ATTRIBUTE-VALUE PAIRS

Let's now take a look at handling more complex (here, nested) attribute-value pairs:

```
%let marker_options = subject=subjidn subjectmarkerattrs=(size=7 color=blue)
                      ongoing=on_flag ongoingmarkerattrs=(size=7 color=red);
```

It's clear here that a regular expression which looks for a match based on the *size* or *color* attribute alone won't be sufficient, as these attributes can appear as sub-attributes of different (main) attributes. We have the choice of defining a more complex regular expression to extract say the value of the *size* sub-attribute for subject markers, or defining two simpler regular expressions, one to extract the entire set of sub-

attributes for the main attributes and ones to extract the values of the sub-attributes from the result. For the sake of simplicity, the second approach may be considered the most prudent. Here we compile a regular expression to extract the *subjectmarkerattrs* attribute's set of sub-attribute-value pairs:

```
%let _rxid1 = %sysfunc(prxparse(/subjectmarkerattrs\s*=\s*\(([^\(\)]+)\)/i));
```

Table 3 explains each component of the regular expression:

Regex component	Explanation
/	Start of regular expression
subjectmarkerattrs	The literal *subjectmarkerattrs*
\s*=\s*	0 or more spaces, then an equals sign, then 0 or more spaces
\(	A literal open-parenthesis character (note the escape character "\")
(	Start of capture buffer
[^\(\)]+	Any characters (one or more) which are *not* (as defined by the "^" carat) open or close-parenthesis characters – the aim here is to capture the entire set of sub-attribute-value pairs *between* the brackets
)	End of capture buffer
\)	A literal close-parenthesis character
/	End of regular expression
i	Match to be case insensitive

Table 3. Explanations of regex components

Having extracted the required set of sub-attributes, we can apply code similar to that in the first example to extract the values for specific sub-attributes:

```
/* The parameter */
%let marker_options = subject=subjidn subjectmarkerattrs=(size=7 color=blue)
                      ongoing=on_flag ongoingmarkerattrs=(size=7 color=red);

/* Compile the regex to extract the set of subjectmarkerattrs sub-attribute-
   value pairs */
%let _rxid1 = %sysfunc(prxparse(/subjectmarkerattrs\s*=\s*\(([^\(\)]+)\)/i));

/* Is there a match? */
%if %sysfunc(prxmatch(&_rxid1, &marker_options)) > 0 %then %do;

   /* Extract the sub-attribute-value pairs (capture buffer 1) */
   %let subjectmarkerattrs = %sysfunc(prxposn(&_rxid1, 1, &marker_options));
   %put >>> &=subjectmarkerattrs;  /* See Output 1 below */

   /* Compile the regex to extract the size (integer from 1 to 9) */
   %let _rxid2 = %sysfunc(prxparse(/size\s*=\s*([1-9])/i));

   /* Is there a match? */
   %if %sysfunc(prxmatch(&_rxid2, &subjectmarkerattrs)) > 0 %then %do;

     /* Extract the size (capture buffer 1) */
     %let subjectmarkersize = %sysfunc(prxposn(&_rxid2, 1, &subjectmarkerattrs));
     %put >>> &=subjectmarkersize;  /* See Output 1 below */
   %end;

   %else %do;
      /* Code to handle a non-match if required */
   %end;
```

```
%end;

%else %do;
   /* Code to handle a non-match if required */
%end;

/* Release the compiled regexes */
%syscall prxfree(_rxid1);
%syscall prxfree(_rxid2);
```

Output 1 shows the log output resulting from the two %PUT statements in the above code, namely the values of the variables SUBJECTMARKERATTRS and SUBJECTMARKERSIZE following their assignment with the corresponding capture buffers:

```
>>> SUBJECTMARKERATTRS=size=7 color=blue
>>> SUBJECTMARKERSIZE=7
```

Output 1. Values of SUBJECTMARKERATTRS and SUBJECTMARKERSIZE as written to the log

Note that the very nature of pattern matching does not stipulate an order in which the searched-for pattern has to occur in the source. Our "parameter":

```
%let marker_options = subject=subjidn subjectmarkerattrs=(size=7 color=blue)
                      ongoing=on_flag ongoingmarkerattrs=(size=7 color=red);
```

may just have well been defined by the user as follows:

```
%let marker_options = ongoing=on_flag ongoingmarkerattrs=(color=red size=7)
                      subject=subjidn subjectmarkerattrs=(color=blue size=7);
```

and this would have been of no consequence for the regular expressions as we defined them. Furthermore, it can be seen that the use of regular expressions removes the need for multiple calls to functions such as INDEX, FIND, SUBSTR, SCAN and NVALID, potentially together with multiple IF conditions, to extract the required information. And if a regex pattern is defined with precisely no more and no less flexibility than is needed, built-in validation of user-defined values becomes inherent in the regular expression and an invalid value ideally results in a non match which is easily tested for and handled.

HANDLING LISTS

Say you have an attribute-value pair where the value is a list of one or more variables, like this:

```
%let other_markers = vars=start end colors = blue red;
```

It's clearly important to ensure that `colors` is treated as an attribute and not as the third item in the list of variable names. Here's a regular expression which handles the situation (the capture buffer we require is once again highlighted in yellow):

```
/vars\s*=\s*([_a-z]\w{0,31}(\s+[_a-z]\w{0,31})*)((\s+[a-z]+\s*=)|$)/
```

Table 4 explains the salient components of the regex:

Regex component	Explanation
`[_a-z]\w{0,31}`	A valid SAS variable name – this captures "`start`"
`(\s+[_a-z]\w{0,31})*`	0 or more valid SAS variable names, each preceded by 1 or more spaces – this captures "`end`"
`((\s+[a-z]+\s*=)|$)`	1 or more spaces, then 1 or more letters, then 0 or more spaces, then an equals sign OR simply end of line ($) in case of no further attribute being present – this captures "`colors =`". The presence of the equals sign results in the match of "`colors =`" occurring outside of the capture buffer we will be requesting in the call to PRXPOSN (**1** again).

Table 4. Explanations of regex components

The regular expression has admittedly become more complex, however to assist readability the character pattern for a valid SAS variable name, if repeated several times in a regular expression or in the program as a whole, can always be put into a macro variable:

```
%let _vrx = [_a-z]\w{0,31};
%let _rxid1 = %sysfunc(prxparse(/vars\s*=\s*(&_vrx(\s+&_vrx)*)((\s+[a-z]+\s*=)|$)/i));

%if %sysfunc(prxmatch(&_rxid1, &other_markers)) > 0 %then %do;
    %let vars = %sysfunc(prxposn(&_rxid1, 1, &other_markers));
%end;

%syscall prxfree(_rxid1);
```

THE WORD BOUNDARY CHARACTER

As a quick but important aside, the possibility of inadvertently matching a searched-for word which happens to form part of a larger word in the source text must always be considered. Here's a simple example which illustrates the point:

```
%let marker_options = nooutline /* and possibly some other options */;
```

The following regular expression will return a positive match because "outline" is present in the source, albeit as *part* of the word "nooutline", and as a result the OUTLINE_FLAG variable will be set, incorrectly, to Y:

```
%if %sysfunc(prxmatch(/outline/i, &marker_options)) > 0 %then %do;
   %let outline_flag = Y;
%end;
%else %do;
   %let outline_flag = N;
%end;
```

Using the word boundary character `\b` in the regular expression guards against falling into such a trap:

```
%if %sysfunc(prxmatch(/\boutline\b/i, &marker_options)) > 0 %then %do;
/* ... */
```

The author recommends always bearing the \b character in mind when building a regular expression and making prudent use of it wherever appropriate.

CONCLUSION

We saw how PRXPARSE is used to compile a regular expression containing a capture buffer, how PRXMATCH then uses that regular expression to find a match, and how PRXPOSN is used to extract the defined capture buffer following a successful match. Using these three functions together offers a powerful technique for extracting specific pieces of information from complex strings without the need for multiple calls to diverse string handling functions followed by validation of the values extracted. Scalability

can also be a benefit facilitated by the use of regex – in a standard macro context adding new options to existing parameters may well be easier than adding new parameters, if only from a documentation point of view.

The topic of regular expressions is non trivial and we have touched on one small but practical use of them. If you are not familiar with regular expressions, or you only have a basic knowledge, take the plunge now, you will not regret it!

RECOMMENDED READING

- *Pattern Matching Using Perl Regular Expressions (PRX)*
  (SAS® 9.4 and SAS® Viya® 3.4 Programming Documentation)
- *SAS® 9 Perl Regular Expressions Tip Sheet:*
  https://support.sas.com/rnd/base/datastep/perl_regexp/regexp-tip-sheet.pdf
- *An excellent site for testing and understanding your regular expressions online:*
  https://regex101.com

CONTACT INFORMATION

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

Rowland Hale
Syneos Health
rowland.hale@syneoshealth.com
www.syneoshealth.com