Paper 178-30

Mapping an Exclusive Regular Expression Strategy

Mark Tabladillo PhD, markTab Consulting, Atlanta, GA
Associate Faculty, University of Phoenix

ABSTRACT

In version 9, the SAS System introduces Perl regular expressions (sometimes known by the acronym PRX, the first three letters of these new functions or call routines). However, previous versions of SAS already had regular expressions (known by their acronym RX, the first two letters of these functions or call routines). This presentation will map specific functional and performance differences in these two exclusive regular expression strategies, and offer recommendations on when to use each strategy.

INTRODUCTION

Let’s define some terms pertinent to this topic. According to the SAS 9.1 Documentation (SAS Institute, 2003):

- **Pattern matching** enables you to search for and extract multiple matching patterns from a character string in one step, as well as to make several substitutions in a string in one step
- **Regular expressions** are a pattern language which provides fast tools for parsing large amounts of text.
- **Metacharacters** are special combinations of alphanumeric and/or symbolic characters which have specific meaning in defining a regular expression.
- **Character classes** are single or combinations of alphanumeric and/or symbolic characters which represent themselves.

Though the SAS documentation says pattern matching can be done in one step, useful examples of regular expressions often involve several steps. However, what could be accomplished in a few code lines with either SAS regular expressions (RX) or Perl regular expressions (PRX) would take many more lines without them.

Regular expressions do not add any functionality beyond what an advanced programmer could write using other SAS functions combined together, however they do provide essentially a short-cut syntax to accomplish several things in a shorter amount of space. Functionally, what regular expressions do sounds like it could be a SAS procedure. However, the functionality is not implemented with a PROC statement, but instead a combination of CALL routines and functions which can be used by the DATA step, or SAS/Macro language, or SAS/AF.

Metacharacters are special character combinations used to generate patterns. One well-known pattern which works in both the SAS and Perl regular expression environments is the asterisk. The asterisk is a general wildcard character, meaning any zero or more characters. You may have used this metacharacter in the Unix or Windows® environment, and it even works with popular search engines like Google® or EBay®. However, the repetitive and high-volume or complex string searching activity common to SAS applications benefits most from regular expressions.

What can make either SAS (RX) or Perl (PRX) regular expressions hard to understand is that they are a combination of:

- Alphanumeric and/or symbolic characters representing themselves (character classes)
- Special combinations of alphanumeric and/or symbolic characters (metacharacters) representing zero or more combinations of alphanumeric and/or symbolic characters
- Specially flagged combinations of alphanumeric and/or symbolic characters which would normally be interpreted as metacharacters, but instead represent themselves (character classes)

This paper is presented in three sections. The first section will examine similar functionality between SAS regular expressions (RX) and Perl regular expressions (PRX). The second section will summarize capabilities unique to Perl regular expressions (PRX). .The last section will offer recommendations on which exclusive set of functions to use, and show two examples which apply the recommended strategy. After the conclusion, there is an annotated bibliography of using regular expressions in the SAS System.

SIMILARITIES BETWEEN SAS (RX) AND PERL REGULAR EXPRESSIONS (PRX)

The similar regular expression functionality is summarized in the following table, which will be discussed in the following text.

SAS (RX)	Perl (PRX)	Description
RXPARSE Function	PRXPARSE Function	Compiles a regular expression (RX or PRX) that can be used for pattern matching of a character value
RXMATCH Function	PRXMATCH Function	Searches for a pattern match and returns the position at which the pattern is found
CALL RXSUBSTR Routine	CALL PRXSUBSTR Routine	Returns the position and length of a substring that matches a pattern (RX includes score)
CALL RXCHANGE Routine	CALL PRXCHANGE Routine PRXCHANGE Function	Performs a pattern-matching replacement
CALL RXFREE Routine	CALL PRXFREE Routine	Frees unneeded memory allocated for a regular expression (either RX or PRX)

Table 1. Similar SAS (RX) and Perl (PRX) Functionality

SIMILARITY ONE: PARSE FUNCTION

The PARSE function of creates a regular expression in memory and references this location using a unique numeric variable, called the **regular expression ID**. The programming challenge will be to determine a single regular expression which will search for the expected pattern. However, sometimes discovering that single regular expression may be elusive in a complex case, and the final results sometimes come by trial and error. Instead of a single regular expression, sometimes multiple parsing statements make more sense; for example, it may be desirable to collect statistics on the numbers of matches to a specific pattern.

Metacharacters for SAS regular expressions (RX) and Perl regular expressions (PRX) are usually different. However, while the metacharacter language may be different, the resulting regular expression can be coded to return an identical result.

The SAS documentation (SAS Institute, 2003) presents an example where the goal is to find a pattern that matches (XXX) XXX-XXXX or XXX-XXX-XXXX for phone numbers in the United States. The first three digits are the area code, and by standardized rules, the area code cannot start with a zero or a one. The fourth through sixth digits are the prefix, and again by standard rules, the prefix also cannot start with a zero or one. The suffix may have any digit, including zero or one, in any of the four places. See the documentation for the code. Let's illustrate how to do the same thing with a SAS regular expression (RX).

```
data _null_;
   if _N_ = 1 then
      do;
         paren = "'('$'2-9'$d$d')'['  ']$'2-9'$d$d'-'$d$d$d$d";
         dash = "$'2-9'$d$d'-'$'2-9'$d$d'-'$d$d$d$d";
         regexp = paren || "|" || dash;
         retain re;
         re = rxparse(regexp);
         if missing(re) then
            do;
               putlog "ERROR: Invalid regexp " regexp;
               stop;
            end;
      end;

   length first last home business $ 16;
   input first last home business;

   if ^rxmatch(re, home) then
      putlog "NOTE: Invalid home phone number for " first last home;

   if ^rxmatch(re, business) then
      putlog "NOTE: Invalid business phone number for " first last business;

   datalines;
Jerome Johnson (919)319-1677 (919)846-2198
Romeo Montague 800-899-2164 360-973-6201
Imani Rashid (508)852-2146 (508)366-9821
Palinor Kent . 919-782-3199
```

```
Ruby Archuleta . .
Takei Ito 7042982145 .
Tom Joad 209/963/2764 2099-66-8474
;

run;
```

Figure 1. Phone Number Example using SAS (RX)

The SAS regular expression (RX) took the same number of code lines as the Perl regular expression (PRX). The two methods can be compared by placing them (individually) inside a SAS macro, and comparing the run times. The run time difference is not distinguishable using only one run, so a SAS macro will allow the code to run multiple times. Because a macro is used, the DATALINES will not be allowed, so they were instead placed inside a text file and read with an INFILE statement. The NONOTES option will be used to suppress messages to the log. The example below is for SAS regular expressions, and the Perl regular expression code would be similar.

```
%macro sasRX;
%let startDateTime = %sysfunc(datetime());
options nonotes;

%do counter = 1 %to 500;
data _null_;
   if _N_ = 1 then
      do;
         paren = "'('$'2-9'$d$d')'['  ']$'2-9'$d$d'-'$d$d$d$d";
         dash = "$'2-9'$d$d'-'$'2-9'$d$d'-'$d$d$d$d";
         regexp = paren || "|" || dash;
         retain re;
         re = rxparse(regexp);
         if missing(re) then
            do;
               putlog "ERROR: Invalid regexp " regexp;
               stop;
            end;
      end;

   length first last home business $ 16;
   infile "L:\My SAS Files\9.1\regularExpression.txt";
   input first last home business;

   if ^rxmatch(re, home) then
      putlog "NOTE: Invalid home phone number for " first last home;

   if ^rxmatch(re, business) then
      putlog "NOTE: Invalid business phone number for " first last business;

   run;
%end;

options notes;
%let endDateTime = %sysfunc(datetime());
%let runTime = %sysevalf(&endDateTime - &startDateTime);
%put START:   %sysfunc(putn(&startDateTime,datetime21.2));
%put END:     %sysfunc(putn(&endDateTime,datetime21.2));
%put RUNTIME: %sysfunc(putn(&runTime,time11.2));

%mend sasRX;

%sasRX;
```

Figure 2. Calculating SAS regular expression (RX) Performance using a SAS Macro

The winner, in this case, is the SAS regular expression (RX) code, coming in at 3.69 seconds compared to 3.80 seconds for the Perl code. This one example **cannot** be used to extrapolate a general statement about comparative regular expression performance. Processing times are a function of processor speed and available memory, and are particular to the defined regular expression. However, the technique can be used in any specific environments to comparatively assess performance between the two methods. It is best to start with one iteration and increase until you see a difference.

The SAS Documentation (SAS Institute, 2003) provides a complete enumeration of SAS regular expression (RX) metacharacters. However, the documentation enumerates only unsupported Perl regular expression metacharacters, summarizes common metacharacter combinations, and refers the reader to externally available sources for a full enumeration. Overall, there are more metacharacter combinations available for Perl regular expressions (PRX), but the SAS (RX) metacharacters include unique features such as $n, which matches a valid SAS name.

SIMILARITY TWO: MATCHING

As illustrated in previous examples, the matching function uses the regular expression to determine a specific numeric position in a string. In those examples, a mismatch was used to be the condition for printing a message to the log. While both the SAS and Perl forms return a position, the Perl form additionally allows the regular expression to be defined within the function. The optional /o flag allows the expression to only be defined one time, since defining a regular expression takes time and memory. There is face-value appeal in being able to define a regular expression and get the match value in one line, but besides being able to do it in one step, the Perl (PRX) capability is equivalent to the SAS (RX) capability.

SIMILARITY THREE: SUBSTRING

The substring routine allows for inputting a regular expression and string, and outputting a position and length. The functionality could not be put into a SAS function, which has only one output. Routines can have a variable number of inputs or outputs. The SAS routine, RXSUBSTR, also allows outputting a score, defined under the RXPARSE function as defaulting the numeric number of times a regular expression matches a string. However, there are metacharacter options available during regular expression creation which allow different ways to calculate a score (see the RXPARSE function documentation, SAS Institute, 2003).

SIMILARITY FOUR: CHANGE

The change routine allows for inputting a regular expression, a maximum number of times to replace, and an old string, and outputs a new string. Both the SAS (RX) and Perl (PRX) routines allow changing a string in place. The Perl routine, PRXCHANGE, also allows outputting a result length, a truncation value (a zero or one binary flag) and the total number of changes (see the PRXCHANGE routine documentation, SAS Institute 2003). Of specific interest may be the truncation value, which is a flag set to zero if the entire replacement is not longer than the string length, and set to one indicating truncation (when the replacement length is longer than the string).

SIMILARITY FIVE: FREE

The free routine will free the memory used to define the regular expression, and the functionality is the same for both SAS (RX) and Perl (PRX). In the author's experience, freeing memory makes a difference when calling regular expressions inside SAS/AF methods, and in general is recommended to be used with all production software, even though the earlier example from the SAS documentation did not use a FREE routine.

UNIQUE PERL REGULAR EXPRESSION (PRX) CAPABILITIES

Perl regular expressions (PRX) add the capabilities in the following table, which will be discussed in the following text.

Perl (PRX)	**Description**
CALL PRXPOSN Routine	Returns the start position and length for a capture buffer
PRXPOSN Function	Returns the value for a capture buffer
PRXPAREN Function	Returns the last bracket match for which there is a match in a pattern
CALL PRXNEXT Routine	Returns the position and length of a substring that matches a pattern and iterates over multiple matches within one string
CALL PRXDEBUG Routine	Enables Perl regular expressions in a DATA step to send debug output to the SAS log

Table 2. Unique Perl (PRX) Capability

Three of these unique features (CALL PRXPOSN routine, PRXPOSN function, and PRXPAREN function) are based on creating capture buffers, defined as part of a match, enclosed in parentheses, explicitly specified in a Perl regular expression (SAS Institute, 2003). Capture buffers are ordered, and may either be identical in definition or different. Conceptually, the capture buffers are a one-dimensional numbered array of results specified in a regular expression.

The regular expression has to create a ordered series of capture buffers in the first place. The examples in the SAS documentation show useful examples, such as parsing the hours, minutes, and seconds from a time string, and determining a first, middle, and last name (even when the middle name is missing). Another use would be parsing an area code, prefix, and suffix from a phone number, which was input in a variety of possible ways. Capture buffers are a good example of why it is not good to claim that regular expressions do things "all in one step". These features require a PRXPARSE function first, followed by one or more commands which access the created capture buffer.

UNIQUE FEATURE ONE: CALL PRXPOSN ROUTINE

The CALL PRXPOSN routine finds the start position and length of a numbered capture buffer within a string. The obvious follow-up would be using a SUBSTR command to take the position and length and do something with the results. A more advanced follow up would be using the resulting capture buffer to do further regular expression definitions, as would be the requirement in a complex problem (such as the address breakdown example mentioned earlier).

UNIQUE FEATURE TWO: PRXPOSN FUNCTION

The PRXPOSN function uses the positional number of the capture buffer to return the value of the result. Again, the capture buffers are collectively like a one-dimensional array, so a single number will return the ordered piece specified in the regular expression. In most cases, the PRXPOSN function would likely be more useful than the CALL PRXPOSN routine.

UNIQUE FEATURE THREE: PRXPAREN FUNCTION

The PRXPAREN function assumes that the capture buffers are created in a ordered hierarchy, where last is both highest and greatest (the last shall be first). The PRXPAREN function will return the non-missing capture buffer which comes in at the highest place. This function is therefore useful when the capture buffer array is **ordinal**, defined as being in a specified position within a numbered series.

The SAS Documentation (SAS Institute, 2003) provides an example where three terms are searched for, in this order: "magazine", "book" and "newspaper". Input strings which have the phrase "newspaper" would return "newspaper" from the PRXPAREN function. Others may only have the highest term of "book" and would return that phrase. A missing return would mean that none of the three keywords were found.

The following example assumes that the Perl regular expression (PRX) is '/(magazine)|(book)|(newspaper)/':

Capture Buffer	Code
1	(magazine)
2	(book)
3	(newspaper)

Table 3. Numbered Ordinal Capture Buffers by Search Code

Input String	Numeric Result of PRXPAREN
"magazine"	1
"book"	2
"newspaper"	3
"magazine book"	2
"book magazine"	2
"magazine boo newspaper"	3
"wall street journal"	. (missing)
"wall street magazine journal"	1
"magazine book news"	2

Table 4. Results of PRXPAREN

A more advanced application would be creating a format first where the terms were assigned to ordinal integers starting with one, and then using the results of PRXPAREN to match back to the original search term. Again, this would be a powerful application which would deny the simplistic belief that regular expressions do things in one step.

UNIQUE FEATURE FOUR: CALL PRXNEXT ROUTINE

The CALL PRXNEXT routine allows either an entire string or substring to be searched with a Perl regular expression (PRX), and will return the position and length of the result. Thus, this routine functionally does something similar to PRXMATCH.

However, this routine is substantially different from PRXMATCH because it allows the string to be iteratively searched for more matches. The "NEXT" term in the routine signifies that it will not only search using the regular expression, but then be ready to find the next occurrence. The SAS Documentation (SAS Institute, 2003) provides an example where the goal is to look for the terms "bat" or "cat" or "rat" within a string, and the CALL PRXNEXT routine is embedded inside a DO WHILE loop (SAS Institute, 2003). However, the author discovered that there is an error in the PRXNEXT routine through version 9.1.3, and in cases where the end of the string includes a match, the start variable will be set higher than the stop variable. Though this condition is not true for the example below, a recommended workaround line has been added to the example (see bold text) for applications through SAS version 9.1.3, and should be used in every case until the bug is fixed:

```
data _null_;
   ExpressionID = prxparse('/[crb]at/');
   text = 'The woods have a bat, cat, and a rat!';
   start = 1;
   stop = length(text);

      /* Use PRXNEXT to find the first instance of the pattern, */
      /* then use DO WHILE to find all further instances.       */
      /* PRXNEXT changes the start parameter so that searching  */
      /* begins again after the last match.                     */
   call prxnext(ExpressionID, start, stop, text, position, length);
      do while (position > 0);
         found = substr(text, position, length);
         put found= position= length=;
         if start > stop then position = 0;
         else
            call prxnext(ExpressionID, start, stop, text, position, length);
      end;
run;

The following lines are written to the SAS log:

   found=bat position=18 length=3
   found=cat position=23 length=3
   found=rat position=34 length=3
```

Figure 3. CALL PRXNEXT Example (SAS Institute, 2003)

UNIQUE FEATURE FIVE: CALL PRXDEBUG ROUTINE

The CALL PRXDEBUG routine works like a SAS option, and allows debugging routine to be sent to the log. This option only works with the DATA step. I did try it in SAS/AF and it does not work there.

The SAS Documentation (SAS Institute, 2003) provides an example of how to use this routine, which is centrally useful when the code is valid but the results are not expected (invalid results will send an error message to the log).

The recommended way to debug a regular expression is to build one piece at a time, perhaps even making several dummy regular expressions, and using PUT statements (which work both in the DATA step and in SAS/AF) to send resulting information to the log.

CHOOSING AN EXCLUSIVE REGULAR EXPRESSION STRATEGY

Advanced programmers should be able to do it all, so you can skip this section, and go to the next section on the application of regular expression strategy.

Practically, there are only periodic needs or uses for regular expressions, and with the anticipation that many people will use them only sporadically, the major recommendation is to use the type of regular expression which has the functionality desired. Certainly, the Perl regular expressions (PRX) have the major advantage of using capture buffers and iterating with CALL PRXNEXT. However, the SAS regular expression (RX) won the phone number performance contest, and there were several features not present for Perl regular expressions (PRX).

Actually, an advanced programmer would not have skipped this section (as advised) because they really do want to know it all, and one task an advanced programmer might do is write a regular expression several ways in order to fine-tune performance, including simulating regular expression results without using them at all. This issue may be important when the regular expression is complex, when there are many regular expressions, or when there are many observations (rows) in the dataset (table).

For people who have not used either form, start with Perl regular expressions (PRX), which not only have more functions and call routines, but also are used in other languages. The SAS Documentation (SAS Institute, 2003) only contains a summary of the common metacharacters, and links to an external website with even more Perl regular expression formats. Not all Perl regular expression metacharacters are supported in SAS, but since the licensing and porting have been worked out, it is possible for the SAS System to expand the functionality of Perl regular expressions (PRX) if and when they change.

Functionally, both SAS regular expressions (RX) and Perl regular expressions (PRX) do similar things, and the major hurdle in learning either one is deciphering what combinations of metacharacters will retrieve the pattern of interest.

APPLICATION OF REGULAR EXPRESSION STRATEGY

Following the above advice, the strategy was then applied to two SAS/AF applications which process survey data. These applications have been discussed in previous papers (Tabladillo, 2003a, 2003b, 2004). None of the SAS (RX) code was changed or replaced, but instead the Perl (PRX) code was written for two sections where no regular expression code was used in the past.

ILLUSTRATION ONE: WINDOWS PRINTER NAMES

The SAS System version 9.1 uses the SYSPRINT system option to either get or set the printer name using the Universal Naming Convention (UNC), which is in the following format:
\\computer_name\printer_shared_name

Because the UNC name uses the backslash, applying Perl regular expression (PRX) functions means that the backslash will have to be flagged. Here is the code before regular expressions:

```
      DCL
         char
            sasPrinter
            ;

      * Check for an active printer;
      sasPrinter = trim(upcase(optgetC('sysprint')));
      logMessage = 'Detected Printer is <'||sasPrinter||'>';
```

Figure 4. Code for Determining UNC Printer Name

The logMessage variable has a setCam method (an event) which will always send the variable name to the log when it is set. The issue is that the sasPrinter variable could hold the printer name in a variety of legal UNC formats:

- \\computer_name\printer_shared_name
- (\\computer_name\printer_shared_name)
- ("\\computer_name\printer_shared_name")

Thus, the printer name could optionally be enclosed in parentheses, or parentheses and double quotation marks. There is currently a total of 12 valid printers enumerated for the application, making a total of 36 possible combinations. Instead of using regular expressions, another approach would be to have an SCL list generate the combinations based on the original bare list of 12 options.

SAS regular expressions (RX) could be used to determine where the printer name, and a way to tackle the problem would be to make three different regular expressions for each of the varieties, and process based on the single truth (and do error process code if none of the three varieties were true).

In this case, speed is not an issue because the regular expression code is not applied to iterative observations in a dataset, but instead applied only once when the application is initialized. Additionally, Perl regular expressions (PRX) offer the feature of capture buffers, which is what was used to get the printer name. The code now looks like the following:

```
      DCL
         char
            sasPrinter
            sasPrinterName
            ,
         num
            prx
            ;

      * Check for an active printer;
      sasPrinter = trim(upcase(optgetC('sysprint')));

      prx = prxparse('/(\\\\[-\\\w]+|[-\w]+)/');
      if prxmatch(prx,sasPrinter) then
         sasPrinterName = prxposn(prx,1,sasPrinter);
      logMessage = 'Parsed Printer Name:  <'||sasPrinterName||'>';
```

Figure 5. Code for Determining UNC Printer Name using a Perl Regular Expression (PRX)

The next table shows how to understand the regular expression string, which is a total of 25 characters long:

Column(s)	Description
1-25	`'/(\\\\[-\\\w]+|[-\w]+)/'`
1	Opening single quotation mark, to start the string – this requirement is from SAS and could have been a double quotation mark – if you start with a single quotation mark then you have to double up all single quotation marks inside the string. The SAS documentation shows many examples of building a string, and simply passing the name inside the regular expression function.
2	Denotes start of a Perl regular expression
3	The open parentheses means the start of the first capture buffer
4-7	A double backslash is used to show the start of the UNC name, but since the backslash is a special Perl character, each of the double backslashes has to be flagged by another one
8-14	The square brackets means any one of the enclosed characters, and the valid characters include hyphen (position 9), backslash (positions 10-11), and any word character including the underscore (positions 12-13)
15	The plus sign means match the preceding expression one or more times
16	The vertical bar is a concatenation symbol (only one, not the SAS usual of two)
17-21	The square brackets mean any one of the enclosed characters, and the valid characters include hyphen (position 18), and any word character including the underscore (positions 19-20). Note that backslash is NOT included in the valid list since this last portion will be equivalent to the UNC printer shared name. Also, note that the end does NOT include parentheses or quotation marks or braces or brackets.
22	The plus sign means match the preceding expression one or more times
23	The closing parentheses show the end of the first capture buffer
24	The slash shows the end of the Perl regular expression
25	The closing single quotation mark – if you start single, you have to end single

Table 5. Printer Character Regular Expression

Having specified the printer name, it could therefore be enclosed by any combination of braces, brackets, or quotation marks, and still the regular expression would simply pick out the printer name. The PRXPOSN function then simply will get the string from the first capture buffer. Finally, the PRXFREE routine is always used to prevent memory allocation problems.

ILLUSTRATION TWO: WINDOWS DIRECTORY NAMES

The second example has to do with Windows subdirectory names, and like the first example the backslash is a consideration. It was desired to get simply the subdirectory from the longer string which started with the drive name and ended with a specific filename:

X:\\Sub_Directory_1\Sub_Directory_2\...\Sub_Directory_N\Filename.Extension

In other words, “Sub_Directory_N” (without the backslashes) would be considered the “PreviousSubdirectory” and the target of interest. This function was previously done with the SUBSTR command, but as in the first example, the Perl regular expression (PRX) code was chosen because the capture buffer could easily clean off the undesired starting and finishing characters, leaving the bare subdirectory of interest. The following code now determines that subdirectory string:

```
       DCL
          num
             prx
             ,
          char
             previousSubdirectory
             ;

       if not(systemError) then do;
          prx = prxparse('/([A-Za-z]:[.-\\\w]+)\\([.-\w]+)\\([.-\w]+)/');
          if prxmatch(prx,inputDirectory) then do;
             previousSubdirectory = prxposn(prx,2,inputDirectory);
          end;
          else
             systemMessage = 'ERROR:  PREVIOUS SUBDIRECTORY NOT FOUND';
          call prxfree(prx);
       end;
```
Figure 6. Determining Previous Subdirectory using a Perl regular expression (PRX)

The next table shows how to understand the regular expression string, which is a total of 46 characters long:

Column(s)	Description
1-46	`*2+^D0]d0}‘=^10___z‘.,__+^10_z‘.,__+^10_z‘.,2*`
1	Opening single quotation mark, to start the string – this requirement is from SAS and could have been a double quotation mark – if you start with a single quotation mark then you have to double up all single quotation marks inside the string. The SAS documentation shows many examples of building a string, and simply passing the name inside the regular expression function.
2	Denotes start of a Perl regular expression
3	The open parentheses means the start of the first capture buffer
4-11	The square brackets mean any one of the enclosed characters, and the valid characters include capital A through Z (positions 5 to 7) and lower case a through z (positions 8 to 10). This single character will represent the drive name.
12	The single colon stands for itself, and indicates the end of the drive name and start of the directories and file name.
13-20	The square brackets mean any one of the enclosed characters, and the valid characters include period (position 14), hyphen (position 15), backslash (positions 16 and 17), and any word character including the underscore (positions 18-19)
21	The plus sign means match the preceding expression one or more times. It is assumed that there is at least one subdirectory preceding the subdirectory of interest, which is an application rule enforced in earlier code. The star (or asterisk) would be used if a zero or more time rule is desired.
22	The closing parentheses show the end of the first capture buffer – note that for this example, it was not required to capture this beginning information in a specific buffer.
23-24	A flagged backslash, indicating the delimiter which separates subdirectories
25	The open parentheses means the start of the second capture buffer
26-31	The square brackets mean any one of the enclosed characters, and the valid characters include period (position 27), hyphen (position 28), and any word character including the underscore (positions 29-30)
32	The plus sign means match the preceding expression one or more times.
33	The closing parentheses show the end of the second capture buffer
34-35	The flagged backslash, indicating the delimiter which separates the last subdirectory from the filename
36	The open parentheses means the start of the third capture buffer
37-42	The square brackets mean any one of the enclosed characters, and the valid characters include period (position 38), hyphen (position 39), and any word character including the underscore (positions 40-41)
43	The plus sign means match the preceding expression one or more times.
44	The closing parentheses means the end of the third capture buffer
45	The slash shows the end of the Perl regular expression
46	The closing single quotation mark – if you start single, you have to end single

Table 6. Windows Subdirectory Regular Expression

The second capture buffer contains the subdirectory text without backslashes. Extra text is stored in the first and third capture buffers. The PRXFREE routine is always included.

CONCLUSION

This presentation compares and contrasted the similarities and differences between the exclusive SAS (RX) and Perl.(PRX) regular expressions. In general, the recommendation is to develop new software or ad-hoc code with the Perl regular expressions (PRX) because of the expanded functionality and generic application in other languages. Advanced programmers will learn both forms and all options for maximum performance.

ANNOTATED BIBLIOGRAPHY

This section provides commentary on selected regular expression documents available on the Internet, and are presented alphabetically by author. Comments and links are current at the time of this document's publication. To keep current, go to the SAS Institute Support website, and enter "regular expression" (no metacharacters) in the search field.

CASSELL (2004)

Cassell offers a tutorial on using Perl regular expressions (PRX) specifically in the SAS environment. The paper is self-titled an "Introduction" giving the very true impression that much more could be done with Perl regular expressions (PRX) than what the paper illustrates. See http://www2.sas.com/proceedings/sugi29/129-29.pdf

SAS INSTITUTE (2004A)

The webpage titled "PRX Function Reference" is not connected to the SAS Online Documentation, and provides an illustrative example of Perl (PRX) regular expressions, along with a all-in-one-page summary of what the routines and functions are. References to "PRXPOSN" are for the CALL PRXPOSN routine, since the PRXPOSN function is not listed. The webpage ends with an enumeration of non-supported metacharacters. See http://support.sas.com/rnd/base/topics/datastep/perl_regexp/regexp2.html

SAS INSTITUTE (2004B)

The webpage titled "Using Regular Expressions" is not connected to the SAS Online Documentation, and provides a step-by-step Perl regular expression (PRX) tutorial through three specific examples: data validation, search and replace, and extracting a substring. Seeing more examples provides ideas of how to combine the routines and functions for practical projects. See http://support.sas.com/rnd/base/topics/datastep/perl_regexp/regexp.motivation.html

SHOEMAKER (2003)

Shoemaker makes an extended and helpful distinction between **meta-characters** (hyphenated), **quantifiers**, **groupers**, and **anchors**. The SAS Documentation (SAS Institute, 2003) puts all these distinctions into the single term **metacharacters** (non-hyphenated), and is the standard followed in this paper.

REFERENCES

Cassell, David L. (2004), "The Perks of PRX…", *SUGI Proceedings*, 2003.

SAS Institute Inc. (2003), *SAS OnlineDoc® 9.1*, Cary, NC: SAS Institute, Inc.

SAS Institute Inc. (2004A), *PRX Function Reference*, Cary, NC: SAS Institute, Inc.

SAS Institute Inc. (2004B), *Using Regular Expressions*, Cary, NC: SAS Institute, Inc.

Shoemaker, Jack N. (2003), "How Regular Expressions Really Work", *Northeast SAS Users' Group (NESUG) Proceedings*, 2003.

Tabladillo, M. (2003a), "Application Refactoring with Design Patterns", *Proceedings of the Twenty-Eighth Annual SAS Users Group International Conference*, Cary, NC: SAS Institute, Inc.

Tabladillo, M. (2003b), "The One-Time Methodology: Encapsulating Application Data", *Proceedings of the Twenty-Eighth Annual SAS Users Group International Conference*, Cary, NC: SAS Institute, Inc.

Tabladillo, M. (2004), "How to Implement the One-Time Methodology", *Proceedings of the Twenty- Ninth Annual SAS Users Group International Conference*, Cary, NC: SAS Institute, Inc.

CONTACT INFORMATION

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

Mark Tabladillo
markTab Consulting
Web: http://www.marktab.com/

TRADEMARK CITATION