Paper 299-2012

Regular Expressions in SAS® Enterprise Guide®

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 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 describe specific functional and performance differences in these two exclusive regular expression strategies, and offer recommendations on when to use each strategy. The technologies will be compared using SAS Enterprise Guide® 4.3.

INTRODUCTION

Regular expressions are the foundation of character pattern matching. Increasingly, textual data is becoming important in both descriptive and predictive analytics as processing power increases and disk cost decreases. SAS® software historically offered robust RX regular expressions (Cassell, 2004), though now deprecated in SAS® 9.3. Fortunately, SAS Institute had already introduced Perl regular expressions (PRX). The best practice for already-existing RX regular expressions is to migrate them to Perl regular expressions in SAS® 9.3 and Enterprise Guide® 4.3.

While SAS Institute offers other excellent advanced products for text mining, Perl regular expressions provide complex text analysis within the Enterprise Guide® license. Intermediate to advanced Enterprise Guide® users should become aware of what this software can provide for character analytics. This regular expression technology can accept any characters you could put into quotations within a SAS data step (the specifics are determined by the configuration of your SAS® installation and native operating system), thus expanding possible use beyond just English.

Let's define some terms pertinent to this topic (SAS Institute, 2012c, 2012e):

- 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 Institute documentation says pattern matching can be done in one step, useful examples of regular expressions often involve several steps. A few code lines with Perl regular expressions replace potentially many lines of code. Single-step pattern matching allows for easier use in PROC SQL and the %SYSFUNC macro command. However, multiple step pattern matching is always available in SAS Enterprise Guide® through the improved Program Editor (SAS Institute, 2012h).

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.

Metacharacters are special character combinations used to generate patterns (SAS Institute, 2012e). 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 Bing®. However, the repetitive and high-volume or complex string searching activity common to SAS® applications benefits most from regular expressions.

What can make 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 four sections. The first section provides a guide for migrating from SAS (RX) regular expressions to Perl (PRX) regular expressions. The second section on Perl regular expression functions and routines indicates best practices for where and when to use these functions in SAS Enterprise Guide®. The third section on advanced Perl regular expression capabilities provides scenarios for multiple-step regular expression commands. The final section provides two useful regular expression examples.

HOW TO MIGRATE FROM SAS (RX) TO PERL (PRX) REGULAR EXPRESSIONS

The old SAS (RX) regular expressions have been deprecated in SAS® version 9.3. If your SAS® version supports Perl regular expressions, you should convert the code to the newer format. The following table provides a matching guide, and you can find details of the Perl regular expressions in the SAS Institute documentation (SAS Institute, 2012b).

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. Mapping from the deprecated SAS (RX) to the Perl (PRX) Regular Expression Language

BEST PRACTICES FOR PERL REGULAR EXPRESSION FUNCTIONS AND ROUTINES

The following table summarizes the major regular expression commands, what they do, and where they are available.

Command	Description	Accepts Perl Regular Expression	Accepts Regular Expression ID	Has a Call Routine Variant
PRXCHANGE	Performs a pattern-matching replacement.	YES	YES	YES
PRXDEBUG	Enables Perl regular expressions in a DATA step to send debugging output to the SAS log.	no	no	YES
PRXFREE	Frees memory that was allocated for a Perl regular expression.	no	no	YES
PRXMATCH	Searches for a pattern match and returns the position at which the pattern is found.	YES	YES	no

PRXNEXT	Returns the position and length of a substring that matches a pattern, and iterates over multiple matches within one string.	no	no	YES
PRXPAREN	Returns the last bracket match for which there is a match in a pattern.	no	YES	no
PRXPARSE	Compiles a Perl regular expression (PRX) that can be used for pattern matching of a character value.	YES	no	no
PRXPOSN	Returns a character string that contains the value for a capture buffer.	no	YES	YES
PRXSUBSTR	Returns the position and length of a substring that matches a pattern.	no	no	YES

Table 2. Chart of Perl Regular Expression Functions and CALL Routine Variants (SAS Institute, 2012b)

This table provides a helpful reference for establishing best practices for applying regular expressions within SAS Enterprise Guide®. Let's consider each column in order. The first column has the function names, and the second column contains a one-line description from the SAS Documentation (SAS Institute, 2012b).

The third column labeled "Accepts Perl Regular Expression" indicates when the command could be used in a single line function. Most useful are the PRXCHANGE and PRXMATCH functions (the PRXPARSE is intended for subsequent regular expression functions or routines). Within Enterprise Guide®, that ability means easy use for these two functions in:

- Filter and Sort
- Query Builder
- The %SYSFUNC Macro command
- Data Step (Program Editor) (SAS Institute, 2012g)

Using a function in a macro command is important for SAS Enterprise Guide® 4.3, which introduces conditional macro variable processing (SAS Institute, 2012h).

The fourth column labeled "Accepts Regular Expression ID" indicates when the command requires a previously-defined numeric value linking to the memory storage location of a compiled Perl regular expression. The PRXPARSE function makes this special numeric value. The compilation includes more than just the regular expression string pattern, and may include one or more capture buffers. This complexity therefore requires previously defining and saving a new numeric variable. Here are some common applications:

- Data step processing -- This classic form of SAS programming is cursor-based, and considers one row (or observation) at a time. The method is generally slower than PROC SQL since the SAS compiler is limited in its ability to optimize joins. However, SAS programmers have a long love relationship with the Data Step, and therefore SAS provides many examples (SAS Institute, 2012g). The recommended approach is to make a numeric variable, and then drop that variable (or do not keep it) for output datasets.
- Macro Processing -- The recommended method is to create a numeric regular expression variable within a local (or even nested) SAS Macro variable. That variable would not need to be carried forward beyond its use in helping other Perl regular expression functions.
- PROC SQL -- A numeric regular expression ID variable could be included in the output dataset. The SAS functions will only work when SAS processes the SQL, but not with pass-through SQL (hosts such as SQL Server have their own functions and commands). A subsequent SQL statement could then drop the numeric regular expression ID value (which typically has no subsequent utility).

The following code provides an example.

```
proc sql;
   create table work.MarkTabTest as
   select S.*,
      prxparse("/\w*chips/") as re,
      ifc(
         prxmatch(calculated re, S.product),
         prxposn(calculated re, 0, S.product),
         " "
      ) as PRX_Return_String
   from sashelp.snacks as S;
   alter table work.MarkTabTest
      drop re;
quit;
```

The fifth column labeled “Has a Call Routine Variant” describes when a CALL routine may be available. All these CALL routines require first establishing a numeric regular expression ID. These routines are made for the data step or for the macro command %SYSCALL, and are not intended for PROC SQL. When you look up those functions in the SAS documentation, you will see applications to the data step (SAS Institute, 2012g). In SAS Enterprise Guide® 4.3, the program window has an improved Program Editor with autocomplete and integrated syntax help, both of which aid in creating data steps (SAS Institute, 2012h). Using the CALL routines with macros is trickier because %SYSCALL will sometimes will unquote the arguments (SAS Institute, 2012f). The recommended best practice is to use the CALL routines in a data step; only advanced users should use CALL routines in a macro, and even then, with caution.

Both the fourth and fifth columns indicate commands intended to be part of a multiple-step regular expression. In my experience, those types of combinations offer the most power and flexibility for character pattern matching. While Enterprise Guide® users might prefer the single-line PRXCHANGE and PRXMATCH commands (due to ease of use), discovering how to make and use multiple-command regular expressions allows for a wider scope of functionality. The examples in this presentation, therefore, focus on the multiple-step commands, and hopefully provide the motivation to study how to use all the available regular expression functions and routines as appropriate.

ADVANCED PERL REGULAR EXPRESSION CAPABILITIES

Perl regular expressions (PRX) add the capabilities in the following table, which will be discussed in the following text. These functions do not have equivalents in the old SAS (RX) regular expressions, another reason for converting regular expression code from RX to PRX.

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 3. Perl (PRX) Capabilities

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, 2012b). 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.

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).

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.

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, 2012d) 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 4. 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 5. 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.

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, 2012a) 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, 2012a).

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 and should be used in cases where this bug occurs:

```
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:

```
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
```

Output 1. Output from PRXNEXT Routine

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® (an advanced SAS product independent of Enterprise Guide®) and it does not work there. The SAS Documentation (SAS Institute, 2012b) 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 to send resulting information to the log.

EXAMPLES OF REGULAR EXPRESSIONS

Perl regular expressions were used in two applications which process survey data. These code snippets could be used in the Program Editor window for a data step. The original applications have been discussed in previous papers (Tabladillo, 2003a, 2003b, 2004). Perl regular expression 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 9.3 uses the SYSPRINT system option to either get or set the printer name using the Universal Naming Convention (UNC), which could be in the following format:

- \\computer_name\printer_shared_name

The goal of this code is to only allow known printers to be used by the application. Originally, the checking did not use regular expressions. 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||'>';
```

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 are currently a total of 12 valid printers enumerated for the application, making a total of 36 possible combinations. The old code has a series of 36 if-then statements to test for a valid printer.

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 offer the feature of capture buffers, which is what was used to get the printer name. Because the UNC name uses the backslash, applying Perl regular expression functions means that the backslash has to be flagged. 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||'>';
```

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	The forward slash denotes the start of this 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 forward slash denotes the end of the Perl regular expression
25	The closing single quotation mark – if you start single, you have to end single

Table 6. 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. It was desired to get simply the subdirectory from the longer string which started with the drive name and ended with a specific filename, such as one of the following formats:

- X:\\Sub_Directory_1\Sub_Directory_2\...\Sub_Directory_N\Filename.Extension
- \\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 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('/([:.-\\\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;
```

The next table shows how to understand the regular expression string, which is a total of 38 characters long. Please study the details if you are considering using this code, since you might want to make some adjustments or improvements:

Column(s)	Description
1-38	`'/([:.-\\\w]*)\\([.-\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	The forward slash denotes the start of this Perl regular expression
3	The open parentheses means the start of the first capture buffer
4-12	The square brackets mean any one of the enclosed characters, and the valid characters include colon (position 5), period (position 6), hyphen (position 7), backslash (positions 8-9), and the \w

	metacharacter, meaning any word character including the underscore (positions 10-11)
13	The asterisk means zero or more characters.
14	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.
15-16	A flagged backslash, indicating the delimiter which separates subdirectories
17	The open parentheses means the start of the second capture buffer
18-23	The square brackets mean any one of the enclosed characters, and the valid characters include period (position 19), hyphen (position 20), and the \w metacharacter, meaning any word character including the underscore (positions 21-22).
24	The plus sign means match the preceding expression one or more times.
25	The closing parentheses show the end of the second capture buffer
26-27	The flagged backslash, indicating the delimiter which separates the last subdirectory from the filename
28	The open parentheses means the start of the third capture buffer
29-34	The square brackets mean any one of the enclosed characters, and the valid characters include period (position 30), hyphen (position 31), and the \w metacharacter, meaning any word character including the underscore (positions 32-33).
35	The plus sign means match the preceding expression one or more times.
36	The closing parentheses means the end of the third capture buffer
37	The forward slash denotes the end of the Perl regular expression
38	The closing single quotation mark – if you start single, you have to end single

Table 7. 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 finally included to gracefully manage memory.

CONCLUSION

This paper provides an overview of regular expressions within SAS Enterprise Guide®, particularly version 4.3. The first section provides a guide for migrating from SAS (RX) regular expressions to Perl (PRX) regular expressions. The second section on Perl regular expression functions and routines indicates best practices for where and when to use these functions in SAS Enterprise Guide®. The third section on advanced Perl regular expression capabilities provides scenarios for multiple-step regular expression commands. The final section provides two useful regular expression examples. Regular expressions can be used to build basic text mining and processing capability within SAS Enterprise Guide®.

REFERENCES

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

SAS Institute Inc. (2012a), *SAS® 9.3 Functions and CALL Routines: Reference: CALL PRXNEXT Routine*, Cary, NC: SAS Institute, Inc. Retrieved March 1, 2012 from http://support.sas.com/documentation/cdl/en/lefunctionsref/63354/HTML/default/viewer.htm#n1obc9u7z3225mn1npwnassehff0.htm

SAS Institute Inc. (2012b), *SAS® 9.3 Functions and CALL Routines: Reference: Functions and CALL Routines by Category*, Cary, NC: SAS Institute, Inc. Retrieved March 1, 2012 from http://support.sas.com/documentation/cdl/en/lefunctionsref/63354/HTML/default/viewer.htm#p0w6napahk6x0an0z2dzozh2ouzm.htm

SAS Institute Inc. (2012c), *SAS® 9.3 Functions and CALL Routines: Reference: Pattern Matching Using Perl Regular Expressions (PRX)*, Cary, NC: SAS Institute, Inc. Retrieved March 1, 2012 from http://support.sas.com/documentation/cdl/en/lefunctionsref/63354/HTML/default/viewer.htm#n13as9vjfj7aokn1syvfyrpaj7z5.htm

SAS Institute Inc. (2012d), *SAS® 9.3 Functions and CALL Routines: Reference: PRXPAREN Function*, Cary, NC: SAS Institute, Inc. Retrieved March 1, 2012 from http://support.sas.com/documentation/cdl/en/lefunctionsref/63354/HTML/default/viewer.htm#p1kh2qqybgx44gn12who44jdxifs.htm

SAS Institute Inc. (2012e), *SAS® 9.3 Functions and CALL Routines: Reference: Tables of Perl Regular Expression (PRX) Metacharacters*, Cary, NC: SAS Institute, Inc. Retrieved March 1, 2012 from http://support.sas.com/documentation/cdl/en/lefunctionsref/63354/HTML/default/viewer.htm#p0s9ilagexmjl8n1u7e1t1jfnzlk.htm

SAS Institute Inc. (2012f), *SAS® 9.3 Functions and CALL Routines: Reference: Using Functions and CALL Routines*, Cary, NC: SAS Institute, Inc. Retrieved March 1, 2012 from http://support.sas.com/documentation/cdl/en/lefunctionsref/63354/HTML/default/viewer.htm#p07c8e0ktns1c7n1vtowjev5g724.htm

SAS Institute Inc. (2012g), *SAS® 9.3 Functions and CALL Routines: Reference: Using Perl Regular Expressions in the DATA Step*, Cary, NC: SAS Institute, Inc. Retrieved March 1, 2012 from http://support.sas.com/documentation/cdl/en/lefunctionsref/63354/HTML/default/viewer.htm#p1vz3ljudbd756n19502acxazevk.htm

SAS Institute Inc. (2012h), *What's New in SAS Enterprise Guide® 4.3*, Cary, NC: SAS Institute, Inc. Retrieved March 1, 2012 from http://support.sas.com/documentation/cdl/en/whatsnew/62580/HTML/default/viewer.htm#egwhatsnew43.htm

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
Enterprise: MarkTab Consulting
Web: http://www.marktab.com/