PharmaSUG2011 – Paper TU08

Perl Regular Expressions: Out of the Oyster, Into Your Code

Eric Larson, Madison, Wisconsin

ABSTRACT

Perl Regular Expressions (PRX) are powerful, but frequently unused, functions that can be employed in-line with your code. These functions are typically used to parse data for patterns, but they can also replace multiple uses of other functions [e.g. FIND()]. The purpose of this paper is to demonstrate how Perl Regular Expressions are constructed and deployed within SAS® programs, including explanations of the "pattern-matching" code and general examples of "when" and "where" to use PRX functions.

INTRODUCTION

There are multiple ways to program the same outcome is SAS; Perl Regular Expressions (PRX) is one method that can be employed to make a programmer's life easier by taking out much of the "overhead" that is involved in the programming of the code. With PRX, strings of text that have a particular "look" can be parsed. For example, "Page NN of….", where "NN" may represent a value of "1" or "7484" can be represented with one expression, rather than coding several overhead variables for looping. PRX functions eliminate the need for the programmer to maintain overhead variables.

The remainder of this paper introduces the building blocks of Perl expressions, provides an example of how to build an expression from these basic blocks, and presents a simple demonstration of the difference between the use of the traditional SAS function FIND and the PRX functions.

METACHARACTERS

Metacharacters are the building blocks of Perl Expressions. These "codes" are used to create Perl Expressions that the PRX functions employ to parse strings of text.

THE COMMON METACHARACTERS:

\ Marks the next character as either a special character, a literal, a back reference, or an octal escape:

- "n" matches the character "n"
- "\n" matches a new line character
- "\\" matches "\"
- "\(" matches "("
- "\/" matches "/"

\d matches a digit character that is equivalent to [0-9].

\D matches a non-digit character that is equivalent to [^0-9].

| Specifies the 'or' condition when you compare alphanumeric strings.

+ Matches the preceding subexpression one or more times:

- "zo+" matches "zo" and "zoo"
- "zo+" does not match "z"
- + is equivalent to {1,}

? Matches the preceding subexpression zero or one time:

- "do(es)?" matches the "do" in "do" or "does"
- ? is equivalent to {0,1}

{n} n is a non-negative integer that matches exactly n times:

- "o{2}" matches the two o's in "food"
- "o{2}" does not match the "o" in "Bob"

{n,} n is a non-negative integer that matches n or more times:

- "o{2,}" matches all the o's in "foooood"
- "o{2,}" does not match the "o" in "Bob"
- "o{1,}" is equivalent to "o+"
- "o{0,}" is equivalent to "o*"

{n,m} m and n are non-negative integers, where n<=m. They match at least n and at most m times:

- "o{1,3}" matches the first three o's in "fooooood"
- "o{0,1}" is equivalent to "o?"

Note: You cannot put a space between the comma and the numbers.

(pattern) matches a pattern and creates a capture buffer for the match. To retrieve the position and length of the match that is captured, use CALL PRXPOSN. To retrieve the value of the capture buffer, use the PRXPOSN function. To match parentheses characters, use "\(" or "\)".

\b matches a word boundary (the position between a word and a space):

- "er\b" matches the "er" in "never"
- "er\b" does not match the "er" in "verb"

\w matches any word character including the underscore and is equivalent to [A-Za-z0-9_].

\W matches any non-word character and is equivalent to [^A-Za-z0-9_].

OTHERS USEFUL METACHARACTERS:

^ Matches the position at the beginning of the input string.

$ Matches the position at the end of the input string.

* Matches the preceding subexpression zero or more times:

- zo* matches "z" and "zoo"
- * is equivalent to {0}

period (.) matches any single character except newline. To match any character including newline, use a pattern such as "[.\n]".

x|y matches either x or y:

- "z|food" matches "z" or "food"
- "(z|f)ood" matches "zood" or "food"

[xyz] specifies a character set that matches any one of the enclosed characters:

- "[abc]" matches the "a" in "plain"

[^xyz] specifies a negative character set that matches any character that is not enclosed:

- "[^abc]" matches the "p" in "plain"

[a-z] specifies a range of characters that matches any character in the range:

- "[a-z]" matches any lowercase alphabetic character in the range "a" through "z"

[^a-z] specifies a range of characters that does not match any character in the range:

- "[^a-z]" matches any character that is not in the range "a" through "z"

\B matches a non-word boundary:

- "er\B" matches the "er" in "verb"
- "er\B" does not match the "er" in "never"

\s matches any white space character including space, tab, form feed, and so on, and is equivalent to [\f\n\r\t\v].

\S matches any character that is not a white space character and is equivalent to [^\f\n\r\t\v].

\t matches a tab character and is equivalent to "\x09".

\num matches num, where num is a positive integer. This is a reference back to captured matches:

- "(.)\1" matches two consecutive identical characters.

PERL EXPRESSIONS

To create the expressions needed for PRX functions, the programmer will “string” together metacharacters to match the pattern of text that you are looking for.

To start, all expressions are contained within slashes (/) at the beginning and the end of the expression (e.g. “/Page \d+ of/”).

Consider a character variable that may or may not have dates in it, but if there is a date, it is in the format of CCYY/MM/DD.

1.	Start with the “/” and add the “CCYY” code “\d{4} to get	“/\d{4}”
2.	Add the code for the “/” to get	“/\d{4}\/”
3.	Add the code for the “MM” to get	“/\d{4}\/\d{2}”
4.	Add the code for next “/” to get	“/\d{4}\/\d{2}\/”
5.	Code for the “DD” to get	“/\d{4}\/\d{2}\/\d{2}”
6.	Add the ending “/” to get	“/\d{4}\/\d{2}\/\d{2}/”

This expression will find ANY text patterns that have four digits followed by a slash then two digits followed by a slash and the two final digits.

How does the code change if the pattern of the dates also uses the dash(-) where the slash(/) is being used? The section of the expression that looks for the slashes using the “or” metacharacter ‘|’ needs to be modified, but doing this is tricky, Just changing the code to “/\d{4}\/|-\d{2}\/|-\d{2}/” will change the meaning of the expression; it will look for 4 digits and a slash OR a dash followed by 2 digits and a slash OR a dash followed by 2 digits.

- To solve this a parenthesis “()” needs to be added around the “\/|-“ value to get “/\d{4}(\/|-)\d{2}(\/|-)\d{2}/”. This will find the patterns “CCYY/MM/DD” as well as “CCYY-MM-DD”. However, since the parentheses were added around the “\/|-“ values, it is recommended to add them around all of the distinct sections of the expression in order to better track sections of the code. Doing this gives the following code: “/(\d{4})(\/|-)(\d{2})(\/|-)(\d{2})/”.

Now, what if the data has the month not as a number but as text (i.e. “DEC” instead of “12”)?

- This means that the month portion of the expression needs to be modified from “(\d{2})” to “(\d{2}|\D{3}) to get “/(\d{4})(\/|-)(\d{2}|\D{3})(\/|-)(\d{2})/”. This expression will now find “CCYY/MM/DD”, “CCYY-MM-DD”, “CCYY/MON/DD”, and “CCYY-MON-DD”.

This code is a good foundation for the Perl expression needed to find dates, but consider that some of the dates have switched the year and day portions. The previous expression is not valid for this order and needs to be changed.

1. The “year” part of the current expression “(\d{4})” looks for exactly 4 digits, but for this example it needs to look for 2 to 4 digits. To do this, the code needs to be modified to “(\d{2,4})” - which means “look for a minimal number of 2 digits, but only up to 4”. The code is now changed to “/(\d{2,4})(\/|-)(\d{2}|\D{3})(\/|-)(\d{2})/”.
2. The same needs to be done for the current “day” portion of the expression, changing “(\d{2})” to “(\d{2,4})”. After these changes the expression is now “/(\d{2,4})(\/|-)(\d{2}|\D{3})(\/|-)(\d{2,4})/”.

But, this new expression causes an issue; it will find any pattern that starts with 2-4 digits and ends with 2-4 digits, thus finding patterns that look like “1234-DEC-5678”. To avoid these patterns, the expression needs modification.

1. Look at the original expression “/(\d{4})(\/|-)(\d{2}|\D{3})(\/|-)(\d{2})/” From this, the expression needs to find this date pattern OR a pattern that has the “day” in place of the “year” and the “year” in place of the “day”. The expression will change to “/(\d{2})(\/|-)(\d{2}|\D{3})(\/|-)(\d{4})/”.
2. But we want to look for both possibilities so the resulting expression is “/((\d{4})(\/|-)(\d{2}|\D{3})(\/|-)(\d{2}))|((\d{2})(\/|-)(\d{2}|\D{3})(\/|-)(\d{4}))/”.

There is still a small problem with this new expression. When a string looks like “1234-DEC-5678”, the data returned will be “1234-DEC-78”, which is wrong. Space boundaries need to be added.

- The code for space boundaries is ‘\b’, and adding it to the expression yields:
  “/((\b)(\d{4})(\/|-)(\d{2}|\D{3})(\/|-)(\d{2})(\b))|((\b)(\d{2})(\/|-)(\d{2}|\D{3})(\/|-)(\d{4})(\b))/”
  This solves the issues with patterns in the form of “1234-DEC-5678” being recognized as a valid pattern.

What if the dates do not have any slashes or dashes, so that the date looks like 01DEC2001? The current expression will not recognize this pattern, and needs to be modified further to capture this date format as well.

- Add the {n,m} metacharacter set as {0,1} to the “(\/|-)” code where they appear, making them “(\/|-){0,1}”, which yields the expression:
  “/((\b)(\d{4})(\/|-){0,1}(\d{2}|\D{3})(\/|-){0,1}(\d{2})(\b))|((\b)(\d{2})(\/|-){0,1}(\d{2}|\D{3})(\/|-){0,1}(\d{4})(\b))/”

This expression will match any date in the form of:

- CCYY/MM/DD
- CCYY-MM-DD
- CCYY/MON/DD
- CCYY-MON-DD
- DD/MM/CCYY
- DD-MM-CCYY
- DD/MON/CCYY
- DD-MON-CCYY
- CCYYMMDD
- CCYYMONDD
- DDMMCCYY
- DDMONCCYY

This is just an example for dates, but with other combinations, text strings can be parsed for a multitude of patterns, including the parsing of binary files for tags that may be needed as flags in SAS programs.

USING THE PERL EXPRESSION

THE PRX FUNCTIONS

```
PRXPARSE      - Compiles a Perl regular expression (PRX) that can be used for pattern
                matching of a character value.
PRXMATCH      - Searches for a pattern match and returns the position at which the
                pattern is found.
PRXPOSN       - Returns the value for a capture buffer.
PRXCHANGE     - Performs a pattern-matching replacement.
PRXPAREN      - Returns the last bracket match for which there is a match in a pattern.
```

THE PRX CALL ROUTINES

```
CALL PRXNEXT  - Returns the position and length of a substring that matches a pattern
                and iterates over multiple matches within one string.
CALL PRXSUBSTR- Returns the position and length of a substring that matches a pattern.
CALL PRXDEBUG - Enables Perl regular expressions in a DATA step to send debug output to
                the SAS log.
CALL PRXFREE  - Frees unneeded memory that was allocated for a Perl regular expression
```

These are some of the PRX functions and call routines that can be used. This paper will focus on the functions PRXPARSE, PRXMATCH, and PRXPOSN.

The following functions allow the created “date” expression to prepare, search, and extract the pattern in a text string.

```
PRXPARSE (perl-regular-expression)
PRXMATCH (regular-expression-id | perl-regular-expression, source)
PRXPOSN  (regular-expression-id, capture-buffer, source)
```

The PRXPARSE() function is used to compile the Perl expression.

```
data PRXTST;
  retain PE 0;  * holds the return code of the PRXPARSE function ;
  PE=PRXPARSE("/((\b)(\d{4})(\/|-){0,1}(\d{2}|\D{3})(\/|-
){0,1}(\d{2})(\b))|((\b)(\d{2})(\/|-){0,1}(\d{2}|\D{3})(\/|-){0,1}(\d{4})(\b))/");
  run;
```

If the value of PE = 0, SAS cannot compile the expression, otherwise PE will be a number that is >= 1

Once the PE value is >= 1, the expression can be used to check the text string for the pattern of data, using the PRXMATCH function. The function returns a value of zero if the pattern is not found or the starting position of the pattern.

```
data PRXTST;
  retain PE 0;  * holds the return code of the PRXPARSE function ;
  PE=PRXPARSE("/((\b)(\d{4})(\/|-){0,1}(\d{2}|\D{3})(\/|-
){0,1}(\d{2})(\b))|((\b)(\d{2})(\/|-){0,1}(\d{2}|\D{3})(\/|-){0,1}(\d{4})(\b))/");
  TestString = "The date of collection is: 2011-Dec-03";
  PatternMatch = PRXMATCH(PE, TestString);
  run;
```

To extract the value that the expression matched, use the PRXPOSN function, which will return the pattern matched into a character variable.

```
data PRXTST;
  retain PE 0;  * holds the return code of the PRXPARSE function ;
  PE=PRXPARSE("/((\b)(\d{4})(\/|-){0,1}(\d{2}|\D{3})(\/|-
){0,1}(\d{2})(\b))|((\b)(\d{2})(\/|-){0,1}(\d{2}|\D{3})(\/|-){0,1}(\d{4})(\b))/");
  TestString = "The date of collection is: 2011-Dec-03";
  PatternMatch = PRXMATCH(PE, TestString);
  if PatternMatch > 0 then FoundPattern = PRXPOSN  (PE, 0, TestString);
  run;
```

After running this code, the following values will be available:

- PE = 1
- TestString = "The date of collection is: 2011-Dec-03"
- PatternMatch = 28
- FoundPattern = "2011-Dec-03"

PERL EXPRESSIONS VERSUS TRADITIONAL SAS

The following code shows a simple comparison between the FIND function and using Perl Expressions. Using the FIND function requires the programmer to keep tracking variables as overhead, while using the PRX functions, the programmer does not need to program these tracking variables.

```
*** setup a string to search ***;
data TestString;
  format TestString $255.;
  TestString = "Page test of <PGTAG> ; Page 1 of <PGTAG> ";
  run;

*** serching for 'Page xxx of' ***;

***  USING FIND FUNCTION ***;
***  This will loop twice ***;
data find;
  set TestString;
```

```
  *** Get the first occurrence of 'Page' ***;
  StartPOS = find(TestString, 'Page', 1);
  *** Continue to search while 'Page' is still in the rest of the line ***;
  do while (StartPOS>0);
    *** Get the first occurance of 'of' after 'Page' ***;
    ofPOS = find(TestString,'of',StartPOS);
    *** Get the number between 'Page' and 'of' ***;
    NumberNext   =   input(scan(substr(TestString,StartPOS,ofPOS-StartPOS),2,'   '),   ??
best32.);
    if NumberNext then do;
      *** If the value is a number then stop searching ***;
      leave;
      end;
    *** If the value is not a number then get the next occurrence of 'Page' ***;
    StartPOS = find(TestString, 'Page', ofPOS);
    end;
  run;

*** PERL EXPRESSIONS ***;
data prx;
  set TestString;
  retain found prx 0;
  format pagestr $20.;
  prx = PRXPARSE("/Page \d+ of/");
  if PRXMATCH(prx, TestString) then do;
    NumberNext = input(scan(PRXPOSN(prx, 0, TestString),2,' '), ?? best32.);
    end;
  run;
```

Here is an example written for parsing a PDF document for annotations:

```
data PDF_ANNOTATIONS
     PDF_PAGES
        ;
  format obn 8. waste $1. pdfdata $32000.;
  retain obn pdfdata;
  infile "&XMLroot.\blankcrf.pdf"   recfm=n end=eof;
  input waste  $1.  ;
  etest = (waste = '0D'x);
*** setup the Perl expressions for parsing the PDF text ***;
  ob = PRXPARSE("/(\d+) 0 obj/");
  pg = PRXPARSE("/(\/)Pages(\/)Kids(\[)/");
  anno = PRXPARSE("/(\/)Subtype(\/)FreeText(\/)C/");

  if ^etest then do;
    ** since reading the data in one character at a time we need to identify and KEEP
       spaces **;
    if waste = ' ' then do;
      **Change the space value to a set value of characters to represent a space**;
      pdfdata = cats(pdfdata,'||||||||||');
      end;
    else do;
      pdfdata = cats(pdfdata,waste);
      end;
    end;
  else do;
    obn + 1;
    **Change the set value for spaces back to a space character**;
    pdfdata = tranwrd(pdfdata,'||||||||||',' ');
    if PRXMATCH(ob, pdfdata) then output PDF_PAGES;
    if PRXMATCH(pg, pdfdata) then output PDF_PAGES;
    if PRXMATCH(anno, pdfdata) then output PDF_ANNOTATIONS;
      etest = 0;
      call missing(pdfdata);
      end;
    keep pdfdata;
```

```
run;
```

CONCLUSION

Perl Expressions represent patterns of text and can be used to search for those patterns within text strings. PRX expressions can be tricky to generate and it takes time to completely understand how they work and when to use them. However, once these expressions are created and with a little practice, they can be a powerful and useful tool.

REFERENCES

"Pattern Matching Using SAS Regular Expressions (RX) and Perl Regular Expressions (PRX)"
http://support.sas.com/onlinedoc/913/getDoc/en/lrdict.hlp/a002288677.htm

AKNOWLEDGEMENTS

I would like to offer many thanks to Jim Johnson for encouraging me to write this paper. And to my wife, Shannon, I would like to express my gratitude for her patience and expertise as I undertook this endeavor.

CONTACT INFORMATION

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

Eric Larson
919 Magdeline Dr
Madison WI 53704
Phone: (608) 469-1107
E-mail: larsonej@me.com