PharmaSUG 2012 - Paper TA08

Perl Regular Expressions in SAS® 9.1+ - Practical Applications

Joel Campbell, Advanced Analytics, Wilmington, NC

ABSTRACT

Perl Regular Expressions (PRX) are powerful tools available in SAS® and many other programming languages and utilities which allow precise and flexible pattern matching. This paper discusses the relative advantages of PRXMATCH() versus INDEX() and presents practical applications to serve as a starting point for programmers of all experience levels to begin incorporating the PRX functions in their own code. Also, a method to create dynamic regular expressions from data step variables is presented in addition to a method of pulling information from a string with a single line of code via PRXCHANGE(). The main aim of this paper is to provide all programmers with resources and motivation to begin using the PRX functions in their code.

INTRODUCTION

SAS first incorporated a form of regular expressions in version 6 and with the advent of the Perl Regular Expressions (PRX) family of functions in version 9.1 their utility has evolved tremendously. Perl Regular Expressions are widely recognized for their power and flexibility – some form of PRX pattern matching is nearly ubiquitous across modern programming languages and user interfaces. For example, PRX are available in PHP, VBA, and C#. Moreover, SAS has added the ability to search your code in the program editor using a regular expression and many, many more examples abound. It is time all SAS programmers become familiar with, and begin to use, the PRX functions. In particular, programmers should use PRXMATCH() instead of INDEX() in almost all cases.

It should be noted that there are many ways to implement PRX functions and I tend to use single line approaches wherever it makes sense. This is in lieu of using the multi-line PRXPARSE()-PRXMATCH() as is the typical approach presented in the SAS documentation.

PRXMATCH() VERSUS INDEX()

Despite the availability of the PRX functions, many SAS programmers still lean on INDEX() when they need to search a character expression for a string of characters. There may also exist a perception that regular expressions are complicated and should only be used when sophisticated pattern matching is required. PRXMATCH() is the most fundamental of the PRX functions and may be used interchangeably with the INDEX function. In order to assess their relative benefits, let's compare their descriptions from the SAS online documentation (9.2) for INDEX() and PRXMATCH():

- INDEX function: *Searches a character expression for a string of characters, and returns the position of the string's first character for the first occurrence of the string.*
- PRXMATCH function: *Searches for a pattern match and returns the position at which the pattern is found.*

Since "a string of characters" is a very simple regular expression, it's true that PRXMATCH() does exactly the same thing as INDEX()… and then some! Said a different way, the capability of INDEX() is a *subset* of the capability of PRXMATCH(). In my experience, the only time that INDEX() may provide an advantage over PRXMATCH() is when searching very long strings for target matches as PRXMATCH() may perform prohibitively slow in these cases.

But using PRX functions is complicated and time consuming, right? In order to address this complaint, let's look at Traditional and PRX equivalent implementations for some common applications:

Basic Task: Search a text variable for the presence of a target string (case sensitive).

```
if index(myvar,”Target”);
```

is generally equivalent to:

```
if prxmatch(“/Target/”,myvar);
```

Basic Task: Search a text variable for the presence of a target string (case insensitive).

```
if index(upcase(myvar),”TARGET”);
```

is generally equivalent to:

```
if prxmatch(“/Target/i”,myvar);
```

More Complex Task: Search a text variable for the presence of any of two target strings (case insensitive).

```
if index(upcase(myvar),”TARGET1”) or index(upcase(myvar),”TARGET2”);
```

is generally equivalent to:

```
if prxmatch(“/Target1|Target2/i”,myvar);
```

More Complex Task: Determine if a text variable *begins* with a target string.

```
if index(myvar,”Target”)=1;
```

is generally equivalent to:

```
if prxmatch(“/^Target/”,myvar);
```

More Complex Task: Determine if a text variable *ends* with target string.

```
if index(myvar,”Target”)=length(myvar)-length(“Target”)+1;
```

is generally equivalent to:

```
if prxmatch(“/Target$/”,myvar);
```

More Complex Task: Determine if a text variable contains one target string prior to the occurrence of a second target string.

```
if 0 < index(myvar,”Target1”) < index(myvar,”Target2”);
```

is generally equivalent to (the “.*” construct allows for any combination of characters):

```
if prxmatch(“/Target1.*Target2/”,myvar);
```

As shown in basic implementations, using PRXMATCH successfully does not require the programmer to learn a bunch of complicated PRX-speak. Meanwhile in more complex applications, PRXMATCH allows for easy implementation where the programmer may otherwise be required to jump through hoops for using a different approach. While it’s true that complex regular expressions can make even seasoned experts go cross-eyed, beginners can begin to leverage the power of PRX without having to learn any complicated PRX-specific syntax! It all starts by using PRXMATCH() instead of INDEX() for simple pattern matching!!

DYNAMIC PRX IN THE DATA STEP

Broadly, there are two different ways to perform a PRXMATCH in a data step. First, as in the examples previous, the regular expression is defined manually and is constant for all the data. That is, the same regular expression is matched against all the data. In some cases, it may be helpful to construct a dynamic regular expression that is customized based on data found for each record. In the example below, the regular expression is constructed based on the variable FirstName and will return true if the value of FirstName is found within the searched expression (in this case, the AECOMM variable) and will produce a message in the log.

As in some examples above, the PRX implementation in this example can be easily implemented using INDEX(). This example has been selected to show the basic syntax. Once a programmer begins to incorporate more complexity into the dynamic PRX function, its power far surpasses that of an equivalent INDEX implementation. Note that for dynamic PRX, it is required to use the multi-statement PRXPARSE-PRXMATCH approach:

```
data _null_;
 set narrative;
    length pattern $50.;
    pattern=’/’||strip(FirstName)||’/i’;
    RE=prxparse(pattern);
    if prxmatch(RE,AECOMM)
     then put “WAR” “NING: Subject identified in comment. ” FirstName= AECOMM=;
run;
```

THE PRXCHANGE FUNCTION

Once a programmer possesses the ability to use PRXMATCH successfully, a supplemental task may be to modify the matched portion in some way. For example, we might want to replace all instances of the name "Bob" in an AE narrative so that it reads "REDACTED" instead:

```
AECOMM = prxchange('s/\bBob\b/REDACTED/i',-1,AECOMM);
```

Note first, that the "s" at the beginning of the regular expression indicates that a substitution will be performed. Secondly, the second parameter of "-1" indicates that all occurrences will be found and replaced. The metacharacter "\b" which surrounds the name "Bob" matches a word boundary so that, for example, the word "bobsled" would not be replaced by "REDACTEDsled."

Also note that if using traditional implementation, the fact that the target string is being replaced by a longer string is not a problem here so long as the length of the final AECOMM with the added length doesn't go over the allocated length of the variable. This is not the case with some traditional implementations which would require more code.

In addition to the simple example just provided, it is possible to "group" portions of a matched string with the "\(" and "\)" for use in the replacement with PRXCHANGE. In this example, the string "Goodnight, Jim" is replaced by "Jim Goodnight."

```
NAME = prxchange('s/\(Goodnight\), \(Jim\)/\2 \1/i',-1,NAME);
```

Matching a portion versus matching the whole

As you'll see in the next section, it's important to understand the difference between matching only a portion of the searched expression versus matching the whole expression when a target string is found. In this example, both regular expressions return true if the target string is found.

```
if prxmatch("/Target/i",myvar); ** Only matches target string **;
```

returns true for the same cases as:

```
if prxmatch("/^.*Target.*$/i",myvar); ** Matches the entire value of MYVAR **;
```

In this example, the "^" and "$" match the beginning and end of the searched string, respectively, and the two instances of ".*" allow for any combination of characters, including no characters at all. So the regular expression might be read as: starts with anything, contains "Target," and ends with anything… which is only slightly, yet importantly different from: contains "Target."

SINGLE-LINE METHOD TO PULL DATA FROM A STRING USING PRXCHANGE()

It is possible to "group" portions of a matched string for recall by the PRXPOSN function in a later data-step statement. The advantage of the PRXPOSN approach is that the regular expression itself is only constructed once, but this method requires the use of PRXPARSE, PRXMATCH, and PRXPOSN functions together (i.e., extra lines of code). In this example, the PRXPOSN approach is used to pull the numeric dose, dose unit, and dose frequency from the verbatim DOSEC variable:

```
data _null_;
 set StudyDrugExposureData;

    ** Matches and groups (case insensitive): "30 mg BID" **;
    RE=prxparse('/^\([0-9]+\) *\([a-z]+\) *\([a-z]+\)$/i');

    if prxmatch(RE,strip(DOSEC)) then do;
        DOSEN =prxposn(RE1,1,strip(DOSEC));
        DOSEU =prxposn(RE1,2,strip(DOSEC));
        FREQC =prxposn(RE1,3,strip(DOSEC));
    end;
run;
```

Some may find the additional overhead required by the PRXPOSN approach undesirable (as it requires the presence of PRXPARSE and PRXMATCH functions prior to using PRXPOSN to set the values). Luckily, there is a single line approach that uses PRXCHANGE instead of PRXPOSN to pull data from a matched string. In this approach it's important to match the whole string so that only the value of interest is returned which is already the case with the regular expression used above – note the use of “^” and “$”. Note also, that the same regular expression is used as the target portion in all examples in this section.

```
data _null_;
 set StudyDrugExposureData;

   ** Matches and groups (case insensitive): “30 mg BID” **;
   DOSEN =prxchange('s/^\([0-9]+\) *\([a-z]+\) *\([a-z]+\)$/\1/i',-1,strip(DOSEC));
   DOSEU =prxchange('s/^\([0-9]+\) *\([a-z]+\) *\([a-z]+\)$/\2/i',-1,strip(DOSEC));
   FREQC =prxchange('s/^\([0-9]+\) *\([a-z]+\) *\([a-z]+\)$/\3/i',-1,strip(DOSEC));
run;
```

In this approach the regular expression matches the whole string and changes it to only the value of interest, which is then passed to the variable being set. In this way, a single line of code may be used to pull data from a string using a PRX function in order to set a variable.

CONCLUSION

The PRX functions, available beginning in SAS version 9.1, are powerful tools which allow for precise and flexible pattern matching. With the examples provided in this paper, a programmer will be able to begin using PRXMATCH() in lieu of INDEX(), use PRX implementation for many common practical applications including constructing dynamic regular expressions in the data step, and pull data from a string with the power of PRX in a single line of code.

CONTACT INFORMATION

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

Joel Campbell
Advanced Analytics, Inc.
P.O. Box 966
Wilmington, NC 28402
Joel.Campbell@AdvancedAnalyticsCRO.com
On the web at www.AdvancedAnalyticsCRO.com