SESUG 2022 Paper 221

The PRXMATCH Function: A Perl of Great Price

Lauren Rackley, Syneos Health; Joshua Horstman, Nested Loop Consulting

ABSTRACT

The PRXMATCH function is one of many hidden pearls in the SAS language. By harnessing the power of Perl regular expressions, PRXMATCH provides rich pattern-matching functionality useful for a variety of tasks involving text data. This paper provides an overview of Perl regular expressions and the PRXMATCH function as well as several examples demonstrating their use. If you work with text data at all, this is a tool you definitely want to have in your toolbox!

INTRODUCTION

Defensive and efficient programming is always appreciated and valued. When writing code, one should make it flexible and dynamic so it can be reused on different data with minimal revisions. The use of Perl regular expressions within the PRXMATCH function can facilitate such dynamic programming, but they remain underutilized by many SAS programmers. Your authors hope that after reading this paper, you will consider using them in your daily programming tasks.

ABOUT PERL REGULAR EXPRESSIONS

A regular expression is simply a character string that defines a search pattern for locating text. While many programming languages include the concept of regular expressions, Perl is particularly well-known for its powerful implementation of regular expressions. The regular expression capabilities of Perl have been adapted into many software systems, including SAS.

A complete tutorial on the syntax of Perl regular expressions is beyond the scope of this paper. However, this information is easily found using common Internet search engines. Moreover, a Table of Perl Regular Expression Metacharacters is included as an appendix to SAS 9.4 Functions and CALL Routines: Reference, Fifth Edition (2016). In addition, Cody (2004) provides a useful tutorial on Perl regular expression syntax. Refer also to Kunwar (2020) and Xie and LaBore (2022).

THE PRXMATCH FUNCTION

The PRXMATCH function is one member of a family of SAS DATA step functions for working with Perl regular expressions. Specifically, the PRXMATCH function accepts a search pattern (specified as a Perl regular expression) and a character string and returns the first position at which a match for the specified pattern is found. If no match is found, the function returns a zero. The syntax is shown below.

PRXMATCH(*perl-regular-expression*, *source-string*)

Often, we are not particularly interested in the specific location at which the pattern match was found, but we simply want to know whether a match was found at all. In that case, we can simply treat the return value of this function as a Boolean value. If a match was found, the function will return a non-zero value, and non-zero values are treated by SAS as being true. If no match was found, the function will return zero, which is treated as false.

EXAMPLE #1 – PARSING TIMEPOINT DESCRIPTIONS

Our first example utilizes pharmacokinetic (PK) data collected during a clinical trial. Such data is often structured around samples taken at a series of timepoints that occur relative to a particular dosing. This example demonstrates the use of the PRXMATCH function to categorize timepoints based on the text found in the timepoint description (variable PCTPT). The desired algorithm is as follows:

- Any timepoint beginning with "FU" is categorized as "followup"
- Any timepoint ending in "0HR" is categorized as "predose"
- Any timepoint ending in "EOI" is categorized as "EOI" (end of infusion)

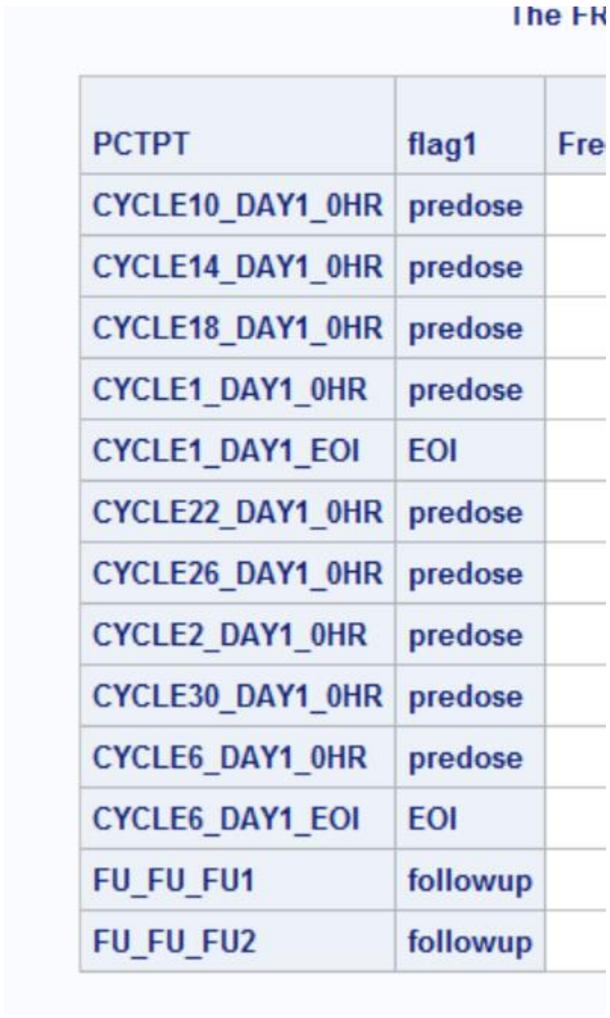
The code below shows how this logic can be implemented using the PRXMATCH function:

```
data pk2;
   merge pk1(in=a) supppc;
   by pcseq usubjid;
   if a;
   length flag1 $ 20;

   if prxmatch('/^FU/i',pctpt) then flag1='followup';
   if prxmatch('/0HR$/i',strip(pctpt)) then flag1='predose';
   if prxmatch('/EOI$/I',strip(pctpt)) then flag1='EOI';
run;
```

Note that each regular expression begins with a forward slash delimiter and ends with another forward slash delimiter followed by the "i" modifier. This modifier specifies that the pattern matching is to be case-insensitive. In addition, the "^" metacharacter is used to indicate that the specified pattern must be found at the beginning of the source string in order to constitute a match. Conversely, the "$" metacharacter indicates that the pattern must occur at the end of the source string.

The resulting values of the FLAG1 variable are shown in Figure 1.



The FR

PCTPT	flag1	Fre
CYCLE10_DAY1_0HR	predose	
CYCLE14_DAY1_0HR	predose	
CYCLE18_DAY1_0HR	predose	
CYCLE1_DAY1_0HR	predose	
CYCLE1_DAY1_EOI	EOI	
CYCLE22_DAY1_0HR	predose	
CYCLE26_DAY1_0HR	predose	
CYCLE2_DAY1_0HR	predose	
CYCLE30_DAY1_0HR	predose	
CYCLE6_DAY1_0HR	predose	
CYCLE6_DAY1_EOI	EOI	
FU_FU_FU1	followup	
FU_FU_FU2	followup	

Figure 1. Output data set for Example 1

EXAMPLE #2 – CALCULATING ELAPSED TIME

Next, we turn to a slightly more complex example, again using pharmacokinetic data. Here, we are given a text string representing the elapsed time since a dose was taken (variable PCELTM). The values all begin with "PT", followed by a numeric value and then either an "M" or an "H" indicating whether the numeric value represents minutes or hours. We would like to derive the elapsed time, in hours, as a numeric variable.

Unlike the first example, we are not simply interested in knowing whether a particular pattern is matched. We also wish to retrieve a certain portion of the source string for use in further processing. To that end, we make use of the PRXPARSE and PRXPOSN functions.

The PRXPARSE function allows us to compile a Perl regular expression to be used later in other function calls. The PRXPARSE function returns a numerical identifier that can be used to refer to that regular expression. Since a regular expression need be compiled only once, it is common to add logic such that the PRXPARSE function is only called during the first iteration of a DATA step (when the automatic variable _N_ has a value of 1). The syntax is shown below:

PRXPARSE(*perl-regular-expression*)

The PRXPOSN function is used to return a portion of the source string that matched a specified element of the pattern given by the regular expression. This is accomplished by including parentheses in the regular expression itself to define a capture buffer. In addition to the regular expression and the source string, the PRXPOSN function requires an additional argument, which is a numeric index that specifies which capture buffer to return. For example, a value of 1 for this argument indicates that the function should return the value corresponding to whatever is matched by the portion of the regular expression enclosed between the first open parenthesis and its corresponding close parenthesis. The syntax is as follows:

PRXPOSN(*regular-expression-id*, *capture-buffer-id*, *source-string*)

The code below utilizes these two functions in conjunction with the PRXMATCH function to derive the elapsed time. The regular expression is included in the call to the PRXPARSE function. It begins by looking for the characters "PT". Next, the regular expression uses the metacharacters "\d+" to indicate that one or more digits must follow. Since this portion of the regular expression is enclosed in parentheses, it is stored in a capture buffer that can be subsequently accessed using the PRXPOSN function. Finally, the "[HM]" portion of the regular expression will match either the character "H" or the character "M". Once the PRXPARSE function has compiled this regular expression, it is assigned a numeric identifier that is stored in the variable R.

This variable R is passed as the first argument to the PRXMATCH function in place of a regular expression. If the PRXMATCH function finds that the value of PCELTM matches the specified pattern, then the portion of the source string corresponding to the first capture buffer will be returned by the PRXPOSN function and stored in the variable TIME. This value will represent the numeric portion of the string that comes after the "PT" and before the "H" or "M". An additional set of calls to the PRXMATCH function is used to determine whether that numeric value represents hours or minutes, and the TIME1 variable is calculated accordingly.

The input dataset is shown in Figure 2 and the output in Figure 3.

PCELTM
PT30M
PT2H
PT4H
PT6H

Figure 2. TIMES data set – input for Example 2

```
data ex2;
   retain r;
   if _N_ = 1 then r = prxparse('/PT(\d+)[HM]/');
   set times;
   if prxmatch(r, pceltm) then time = prxposn(r, 1, pceltm);

   *Get time in hours so divide minutes by 60;
   if prxmatch('/M/',pceltm) then time1=time/60;
   if prxmatch('/H/',pceltm) then time1=time;
run;
```

PCELTM	TIME	TIME1
PT30M	30	0.5
PT2H	2	2
PT4H	4	4
PT6H	6	6

Figure 3. EX2 data set – output for Example 2

EXAMPLE #3 – SEARCHING FOR MULTIPLE SUBSTRINGS

Our final example provides an opportunity to compare the use of the PRXMATCH function with use of the INDEX function. In some cases, we will find PRXMATCH gives us additional flexibility and can make our code more compact.

Suppose the standardized names of concomitant medications taken by subjects during a clinical trial are stored in a variables CMDECOD. We would like to subset our data to keep only those records for which CMDECOD includes the substrings "codone", "morphone", or "methadone". Furthermore, we wish to make this comparison case-insensitive as this will make our code more robust in case future data received are formatted differently than we expected.

This subsetting could be performed using multiple calls to the INDEX function as follows. Note the use of the nested UPCASE function calls to make the comparison case-insensitive.

```
where index(upcase(cmdecod),'CODONE') or
      index(upcase(cmdecod),'MORPHONE') or
      index(upcase(cmdecod),'METHADONE');
```

The PRXMATCH function provides an alternate and simpler way to encode the same logic:

```
where prxmatch('/codone|morphone|methadone/i',cmdecod);
```

This regular expression will match any value of CMDECOD that contains any of those three substrings, and the "i" modifier specifies that the comparison is performed without regard to case.

CONCLUSION

These examples illustrate the power and flexibility available using regular expressions in SAS. While some learning is required upfront to become proficient with the unique syntax of Perl regular expressions, this investment pays off by allowing the programmer to create more robust, efficient, and compact solutions to certain coding problems. It is recommended that the reader explore the references cited to delve deeper into the use of Perl regular expressions and the PRX family of functions.

REFERENCES

Cody, Ron. 2004. "An Introduction to Perl Regular Expressions in SAS 9." *Proceedings of the SAS Users Group International (SUGI) 29 Conference*. Cary, NC: SAS Institute Inc. Available at https://support.sas.com/resources/papers/proceedings/proceedings/sugi29/265-29.pdf.

Kunwar, Pratap Singh. 2020. "RegExing in SAS® for Pattern Matching and Replacement." *Proceedings of the SAS Global Forum 2020 Conference*. Cary, NC: SAS Institute Inc. Available at https://www.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2020/5172-2020.pdf.

SAS Institute Inc. 2016. SAS® 9.4 Functions and CALL Routines: Reference, Fifth Edition. Cary, NC: SAS Institute Inc.

Xie, Edwin and John M. LaBore. 2022. "Facilitating Complex String Manipulations Using SAS PRX Functions." *Proceedings of the Western Users of SAS Software (WUSS) 2022*

Conference. Available at https://www.wuss.org/proceedings/2022/WUSS-2022-Paper-165.pdf.

CONTACT INFORMATION

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

Lauren Rackley
Syneos Health
Lauren.g.rackley@gmail.com

Josh Horstman
Nested Loop Consulting
josh@nestedloopconsulting.com