SAS165-2022

Facilitating Complex String Manipulations Using SAS PRX Functions

Edwin Xie, John M. LaBore, SAS Institute Inc.

ABSTRACT

SAS programmers learn to use many basic SAS functions within the DATA step, but surprisingly few learn about the SAS PRX (**P**erl **R**egular E**x**pression) functions and call routines. The SAS PRX functions provide a powerful means to handle complex string manipulations, by enabling the same end result with fewer lines of code, or by enabling the analysis of data previously out of reach of the basic string manipulation functions. The PRX functions and call routines became available in SAS version 9, are accessible within the DATA step, and are tools that every advanced SAS programmer should have in their toolkit. Examples are provided to give a quick introduction to the syntax, along with a review of the resources available to the programmer getting started with SAS PRX functions.

INTRODUCTION

SAS PRX functions are extremely valuable to SAS users that understand how to use them. This paper will explain key concepts and provide several examples, along with a discussion of resources readily available to the programmer interested in leaning into the learning curve about SAS PRX functions.

WHAT IS A REGULAR EXPRESSION?

A regular expression is a sequence of characters that specifies a search pattern in the text. Each character in a regular expression is either a metacharacter, has a special meaning, or a regular character that has a literal meaning. The pattern syntax in PRX functions is similar to Perl.

Here is an example of a simple regular expression:

```
data _null_;
   text = "A tidy tiger tied a tie tighter.";
   pattern = "/ti(dy|ger)/";
   pos = prxmatch(pattern, text);
   put pos=;
run;
```

Produced in log: `pos=3`

In this example, PRXMATCH is one of the PRX functions, it performs a pattern match and returns the position at which the pattern is found, or 0 on failure. The first argument is the regular expression pattern, and the second argument is the source string to search. The “/” is a delimiter. Both the “|” and “()” are metacharacters, the former one specifies the OR condition, the later one indicates grouping.

METACHARACTERS

A metacharacter is a character that has a special meaning in the pattern. This section introduces some basic metacharacters.

Assume you desire to match the word “tie”. You cannot simply use the pattern “/tie/” because it will also match other words such as “tied”. A better approach is to use the metacharacter “\b” to indicate the word boundary:

```
data _null_;
   text = "A tidy tiger tied a tie tighter.";
   pattern = "/\btie\b/";
   pos = prxmatch(pattern, text);
   put pos=;
```

```
run;
```

Produced in log: `pos=21`

If you want to match any word that starts with “ti”, this can also be easily achieved with metacharacters:

```
data _null_;
   text = "A tidy tiger tied a tie tighter.";
   pattern = "/\bti[a-z]*/";
   pos = prxmatch(pattern, text);
   put pos=;
run;
```

Produced in log: `pos=3`

In the case above, the “[]” is a metacharacter who specifies a character set that matches any one of the enclosed characters, so “[a-z]” matches any lowercase alphabetic character in the range a through z. The “*” is also a metacharacter, it is a repetition factor that matches the preceding subexpression zero or more times.

In addition to specifying the character set yourself, there are also some predefined character sets, such as “[[:alpha:]]” matches an alphabetic character, “\w” matches a “word” character including alphanumeric and underscore, “\d” matches a digit character that is equivalent to [0−9], etc. The period “.” matches any single character except newline.

For a character set specified by using “[]”, a preceding “^” can be used to specify it complement. For example [^a-z] matches a character that other than lowercase a to z, and [[:^alpha:]] matches a nonalphabetic character.

For a character set specified by using \w, \d, \b, and \s, you can specify their complements by using the corresponding capital letters. For example, “\W” matches any non-“word” character.

Both characters and character sets can be used with repetition factors. The repetition factor “*” matches the preceding subexpression zero or more times. The “+” matches one or more times. And the “?” matches zero or one time. For example, “/go+d/” can match both "god" and “good”.

There is also a group of metacharacters that matches position. The “^” matches the position at the beginning of a string. The “$” matches the end of a string. The “\b” matches a word boundary. For example, “/^a$/” only matches a string that contains a single word “a”. If there are any extra characters in the string, the match will fail:

```
data _null_;
   text="ab";
   id=prxparse("/^a$/");
   pos=prxmatch(id, text);
   put pos=;
run;
```

Produced in log: `pos=0`

The PRX functions and routines support more than one hundred metacharacters; a full list is available in SAS documentation online; see the link to “Tables of Perl Regular Expression (PRX) Metacharacters” in the References section.

MODIFIERS

Modifiers are the letters appended after the pattern. They change the way that a pattern is used by PRX functions and call routines. SAS PRXs support 5 modifiers, as shown in Table 1:

Modifier	Description
i	Use case-insensitive pattern matching.
o	Compile pattern once.
x	Use extended regular expressions.
m	Treat string as multiple lines.
s	Treat string as single line.

Table 1: Supported Modifiers

The modifier “i” means case **i**nsensitive. It changes pattern matching from default case sensitive to case insensitive. For example, matching the character “a” with or without the modifier “i”, the results are different:

```
data _null_;
   text = "A tidy tiger tied a tie tighter.";
   without_i = prxmatch("/a/", text);
   with_i = prxmatch("/a/i", text);
   put without_i= with_i=;
run;
```

Produced in log: `without_i=19 with_i=1`

The modifier “o” means **o**ptimization. By default, if a pattern is a string literal, it will be compiled only once:

```
prxparse("/a/");
```

But if a pattern is a variable, it will be compiled for every observation:

```
pattern = "/a/";
re=prxparse(pattern);
```

If the pattern will not be changed during the loop, it actually only needs to be compiled once when it is first used. To avoid recompiling, the code can be optimized to:

```
pattern = "/a/";
if _N_=1 then do;
   retain re;
   re = prxparse(pattern);
end;
```

Or just use the modifier "o" after the pattern:

```
pattern = "/a/o";
re=prxparse(pattern);
```

The modifier “x” indicates a pattern is in an e**x**tended form: a whitespace that is neither backslashed nor within a bracketed character class is ignored. Also, the “#” character is treated as a metacharacter introducing a comment that runs up to the pattern's closing delimiter. Usually, you can use this modifier when you want to break up your regular expression into multiple lines:

```
data _null_;
   prx = prxparse("/foo
                   bar     #to match foobar, not foo bar
```

```
                    /x");
   str = "foo bar foobar";
   pos = prxmatch(prx, str);
   put pos=;
run;
```

Produced in log: `pos=9`

The modifier “m” changes the default behavior of the metacharacters “^” and “$”. Without it, “^” matches the start of a string, and “$” matches the end of a string. If “m” is used, “^” and "$" match the start and end of each line within a string:

```
%let newline = '0a'x;
data _null_;
   x = "First line" || &newline ||
       "Second line" || &newline;
   without_m = prxmatch("/^Second/", x);
   with_m = prxmatch("/^Second/m", x);
   put without_m= / with_m=;
run;
```

Produced in log:
```
without_m=0
With_m=12
```

At this time, if you want to match the end of the entire string, you can use \z instead.

The modifier “s” changes the metacharacter “.” to match any character whatsoever, even a newline, which normally it would not match.

TAKING THE NEXT STEP

After understanding basic concepts of how PRX functions work, programmers should review the numerous resources available to gain a more in-depth understanding of both the functions and the call routines. Functionality of the functions and call routines are as shown in Table 2:

Functionality	Functions	CALL Routines
Pre-compile a pattern	PXPARSE	
Release the compiled pattern		CALL PRXFREE
Pattern match	PRXMATCH	CALL PRXSUBSTR CALL PRXNEXT
Match and substitution	PRXCHANGE	CALL PRXCHANGE
Capture buffer related	PRXPOSN PRXPAREN	CALL PRXPOSN
Debug output toggle		CALL PRXDEBUG

Table 2: Functionality of SAS PRX functions and CALL routines

These PRX functions and call routines are described in a wide range of online resources, technical papers, and books. For those just starting out, the book “Unstructured Data Analysis: Entity Resolution and Regular Expressions in SAS” (Windham, 2018) provides practical examples and techniques to use regular expressions in dealing with unstructured data. Programmers in the health and life sciences might appreciate reviewing the papers titled “Cracking Cryptic Doctors’ Notes with SAS PRX Functions”

(Alabaster and Armstrong, 2020) and “Interpreting Electronic Health Data Using SAS PRX Functions” (Alabaster and Armstrong, 2018).

SAS, of course, has multiple sources of information about the PRX functions and call routines. The first stop would be at https://support.sas.com/en/documentation.html. A search for the term “PRX” on that page turns up a wealth of links to a wide range of resources including tables of Perl Regular Expression Metacharacters and much more. A handy tip sheet is available at https://support.sas.com/content/dam/SAS/support/en/products-solutions/base-sas/tip-sheets/regexp-tip-sheet.pdf.

Plugging “PRX” into the search bar at https://www.lexjansen.com pops up more than 700 results, including those authored by Alabaster and Armstrong mentioned above. Also included in the list are multiple papers by David Cassell, one of the earliest to sing the praises of the capabilities of the PRX functions and call routines. Another paper that will turn up in that same search, presented recently at PharmaSUG 2022, is “Functions (and More!) on CALL!” by Richann Watson and Louise Hadden in which there is a discussion of the use of PRXCHANGE and PRXPARSE.

CONCLUSION

SAS PRX functions broaden the toolset of the skilled SAS programmer, permitting more efficient handling of numerous complex text string manipulations, compared to using basic string manipulation functions. Programmers should study these functions, and the related call routines, to be ready to apply them in appropriate programming situations.

REFERENCES

Alabaster, Amy, and Mary Anne Armstrong. 2020. “Cracking Cryptic Doctors’ Notes with SAS PRX Functions.” *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/2020/4638-2020.pdf.

Alabaster, Amy, and Mary Anne Armstrong. 2018. “Interpreting Electronic Health Data Using SAS PRX Functions.” *Proceedings of the Western Users of SAS Software 2018 Conference.* Sacramento, CA. Available at https://www.lexjansen.com/wuss/2018/92_Final_Paper_PDF.pdf.

Book, Dan. 2015. “Perl regular expressions.” Available at https://perldoc.perl.org/perlre

Cassell. David L. 2007. “The Basics of the PRX Functions.” *Proceedings of the SAS Global Forum 2007 Conference*, Cary, NC: SAS Institute Inc. Available at https://support.sas.com/resources/papers/proceedings/proceedings/forum2007/223-2007.pdf

SAS Institute Inc. “Perl Regular Expressions Tip Sheet.” Available at https://support.sas.com/content/dam/SAS/support/en/products-solutions/base-sas/tip-sheets/regexp-tip-sheet.pdf

SAS Institute Inc. “Tables of Perl Regular Expression (PRX) Metacharacters.” Available at https://documentation.sas.com/?docsetId=lefunctionsref&docsetVersion=v_001&docsetTarget=p0s9ilagexmjl8n1u7e1t1jfnzlk.htm

Watson, Richann, and Louise Hadden. 2022. “Functions (and More!) on CALL!” *Proceedings of the PharmaSUG 2022 Conference.* Austin, TX. Available at https://www.lexjansen.com/pharmasug/2022/AP/PharmaSUG-2022-AP-111.pdf

Windham, Matthew. 2018. *Unstructured Data Analysis: Entity Resolution and Regular Expressions in SAS.* Cary, NC: SAS Institute Inc.

CONTACT INFORMATION

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

Edwin (You) Xie
SAS Institute Inc.
you.xie@sas.com

John LaBore
SAS Institute Inc.
john.labore@sas.com