PharmaSUG 2023 - AP-126

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 that 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 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 is 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 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 “[]” (left and right brackets) 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.

When defining a character set using “[]”, a preceding “^” can be used to indicate its 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, the pattern '/go+d/' can match any word that starts with 'g', has one or more 'o's in the middle, and ends with 'd'. To specify an exact number of repeats, use {m} or {m,n}. For instance, the pattern “/go{1,2}d/” matches “god” and “good”, but not “gd” or “goood”.

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
```

To match a metacharacter itself, it must be escaped from its metacharacter meaning. This can be done by placing a backslash in front of the metacharacter. For example, "()" is a metacharacter used for grouping, but you can use "\" to escape it and match parenthesis:

```
data _null_;
   text = "My phone number is (555)123-4567";
   pattern = "/\(\d{3}\)\s?\d{3}-\d{4}/";
   pos = prxmatch(pattern, text);
   put pos=;
run;
```

Produced in log:

```
pos=20
```

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);
```

It is worth noting that if the function is called from a macro, such as using %SYSFUNC, the pattern will be recompiled with each call, regardless of whether '/o' is used.

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
```

When modifier m is used, if you want to match the beginning of the string, you can use “\A” instead of “^”. If you want to match the end of the entire string, you can use “\Z” or “\z” instead of “$”. The difference between "\Z" and "\z" is "\Z" ignores an optional newline at the end: `%let newline = '0a'x;`

```
data _null_;
   x = "First line" || &newline ||
       "Second line" || &newline;
   /* Matched at the beginning of the string */
   first  = prxmatch("/\AFirst/m", x);

   /* Cannot match the second line */
   second = prxmatch("/\ASecond/m", x);

   /* Matched, the tailing newline is ignored  */
   big_z   = prxmatch("/line\Z/m", x);

   /* Cannot match because of the last newline */
   small_z = prxmatch("/line\z/m", x);

   put first= / second= / big_z= / small_z=;
run;
```

Produced in log:

```
first=1
second=0
big_z=7
small_z=0
```

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

```
%let newline = '0a'x;
data _null_;
   x = "First line" || &newline ||
       "Second line" || &newline;
   without_s = prxmatch("/line.Second/", x);
   with_s = prxmatch("/line.Second/s", x);

   put without_s= / with_s=;
run;
```

Produced in log:

```
without_s=0
with_s=7
```

FUNCTIONS AND CALL ROUTINES

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

Let us see how we can use these PRX functions and call routines in SAS. PRX functions return either numeric or character results. PRX call routines do not return a value, they pass values out by altering variable values. Functions can be used in assignment statements or expressions, but CALL routines need to be invoked with CALL statements.

PATTERN COMPILATION AND RELEASE

PRXPARSE(perl-regular-expression)

CALL PRXFREE(regular-expression-id)

The PRXPARSE function returns a pattern identifier number that is used by other Perl functions and CALL routines to match patterns. If an error occurs in parsing the regular expression, it returns a missing value.

When using PRXPARSE, it should be noted that if the incoming pattern is a constant string, PRXPARSE only compiles the pattern when it is called for the first time and returns the identifier generated for the first time in subsequent loop calls:

```
%let loop=3;

data _null_;
  do i=1 to &loop;
    id=prxparse("/hello/");
    put id=;
  end;
run;
```

Produced in log:

```
id=1
id=1
id=1
```

Otherwise, if the pattern is in a variable, PRXPARSE compiles the pattern every time it is called and returns unique identifiers:

```
data _null_;
  var="/hello/";
  do i=1 to &loop;
    id=prxparse(var);
    put id=;
  end;
run;
```

Produced in log:

```
id=1
id=2
id=3
```

To avoid the re-compile, you can use the modifier "o" we talked about above.

The identifiers and memory will be automatically freed at the conclusion of the DATA step processing. You can also CALL PRXFREE(identifier) to release the identifier manually. The CALL routine also sets the identifier to a missing value:

```
data _null_;
  id=prxparse("/hello/");
  put id=;
  call prxfree(id);
  put id=;
run;
```

Produced in log:

```
id=1
id=.
```

MATCHING

PRXMATCH(regular-expression-id | perl-regular-expression, source)

CALL PRXSUBSTR(regular-expression-id, source, position)

CALL PRXNEXT(regular-expression-id, start, stop, source, position, length)

PRXMATCH searches for a pattern match and returns the beginning position at which the pattern is found. The regular expression pattern can be a pattern or an identifier returned by PRXPARSE.

The input pattern of CALL PRXSUBSTR can only be an identifier. CALL PRXSUBSTR returns the beginning position by arguments. In addition, it can also return the length of the substring that is matched by the pattern. For example, we want to get the phone number from a sentence:

```
data _null_;
  text="Our customer service phone number is 800-727-0025.";
  id=prxparse("/\d{3}-\d{3}-\d{4}/");
  call prxsubstr(id, text, pos, len);
  number=ksubstr(text, pos, len);
  put number=;
run;
```

Produced in log:

```
number=800-727-0025
```

We can also use PRXMATCH and PRXPOSN to get the same result. When the second argument is 0, PRXPOSN returns the entire match:

```
data _null_;
  text="Our customer server phone number is 800-727-0025.";
  id=prxparse("/\d{3}-\d{3}-\d{4}/");
  pos=prxmatch(id, text);
  number=prxposn(id, 0, text);
  put number=;
run;
```

Produced in log:

```
number=800-727-0025
```

CALL PRXNEXT can specify the start and end position of the source string for pattern matching, so it is usually used to match strings consecutively. The start position will be updated by the CALL routine after each successful match. For example, to match all the words that begin with "ti":

```
data _null_;
  text = "A tidy tiger tied a tie tighter.";
  id = prxparse("/\bti\w*\b/");
  start = 1;
  stop = -1;
  call prxnext(id, start, stop, text, pos, length);
  do while (pos > 0);
    found = ksubstr(text, pos, length);
    put found= pos= length=; call prxnext(id, start, stop, text, pos,
    length);
  end;
run;
```

Produced in log:

```
found=tidy pos=3 length=4
found=tiger pos=8 length=5
found=tied pos=14 length=4
found=tie pos=21 length=3
found=tighter pos=25 length=7
```

SUBSTITUTION

PRXCHANGE(perl-regular-expression | regular-expression-id, times, source)

CALL PRXCHANGE(regular-expression-id, times, old-string > > >)

PRXCHANGE performs pattern-matching and replacement. It returns the new string after the replacement. The case below reserves the first and last names of the inputs:

```
data _null_;
  input name & $32.;
  name=prxchange('s/(\w+), (\w+)/$2 $1/', -1, name);
  put name=;
  datalines;
  Jones, Fred
  Kavich, Kate
  Turley, Ron
  Dulix, Yolanda
;
run;
```

Produced in log:

```
name=Fred Jones
name=Kate Kavich
name=Ron Turley
name=Yolanda Dulix
```

Besides returning the new string, CALL PRXCHANGE can also return the length (without trailing blanks) of the new string, a numeric value to indicate if the new string is truncated, and the total number of replacements that were made.

CAPTURE BUFFER

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

PRXPAREN(regular-expression-id)

CALL PRXPOSN(regular-expression-id, capture-buffer, start <,length>)

In the case above, the metacharacter “()” creates capture groups (also referred to as capture buffers). The $1 and $2 in the replacement string are backreferences. They represent the matched content in the first and second groups respectively. The PRXPOSN function returns the matched string in the specified capture buffer and the PRXPAREN function returns the largest capture-buffer number that can be passed to the PRXPOSN function:

```
data _null_;
  fullname="Fred Jones";
  id=prxparse("/(\w+)\s(\w+)/");
  pos=prxmatch(id, fullname);
  paren=prxparen(id); if paren=2 then do;
    first=prxposn(id, 1, fullname);
    last=prxposn(id, 2, fullname);
    put first= last=;
  end;
run;
```

Produced in log:

```
first=Fred last=Jones
```

If you only want to group without capturing, you can use the non-capturing group (?:). For example, /ti(?:dy|ger)/ matches the same words with /ti(dy|ger)/, but it does not create capture groups as "()" does.

DEBUGGING

CALL PRXDEBUG(on-off)

The CALL PRXDEBUG enables or disables the regular expression debugging log. You can turn the debugging on and off multiple times in your program if you want to see debugging output for particular PRX function calls.

USING PRX FUNCTIONS IN VIYA

CAS DATA STEP

All the PRX functions and call routines can be used in a CAS DATA step, but only selected PRX functions are supported by CAS server. In other words, if a data step utilizes a PRX function or call routine that is not supported by CAS server, the data step will be executed on the Compute server in Viya. The PRX functions available in CAS server are: PRXPARSE, PRXMATCH, PRXCHANGE, and PRXPOSE. The PRXPAREN and all PRX call routines are not supported by CAS server.

For example, this program runs on the CAS server, and produces a note "NOTE: Running DATA step in Cloud Analytic Services." in the log:

```
data _null_/ sessref="MySession";
  /*Note:  MySession is an active CAS session*/
  id=prxparse('/(magazine)|(book)|(newspaper)/');
  pos=prxmatch(id, 'find book here');
  put pos=;
run;
```

Produced in log:

```
NOTE: Running DATA step in Cloud Analytic Services.
pos=6
```

As a point of comparison, the program below will not be executed because it utilizes PRXPAREN, which is not supported by CAS server:

```
data _null_/ sessref="MySession";
  /*Note:  MySession is an active CAS server session*/
  id=prxparse('/(magazine)|(book)|(newspaper)/');
  pos=prxmatch(id, 'find book here');
  paren=prxparen(id);
   put paren=;
run;
```

Produced in log:

```
NOTE: Running DATA step in Cloud Analytic Services.
ERROR: The function PRXPAREN is unknown, or cannot be accessed.
```

However, if the same program above is run in Viya but not in a CAS server session (below), it will execute (on the Compute server in Viya). Note the lack of a NOTE indicating the DATA step is running in Cloud Analytic Services.

```
data _null_;
  id=prxparse('/(magazine)|(book)|(newspaper)/');
  pos=prxmatch(id, 'find book here');
  paren=prxparen(id);
  put paren=;
run;
```

Produced in log:

```
paren=2
```

THE CAS PROCEDURE

The CAS procedure supports the following PRX functions: PRXPARSE, PRXMATCH, PRXCHANGE, PRXPOSE and PRXFREE. Please note that PRXFREE is a function here, not a call routine. You can directly call these functions in a CAS procedure:

```
proc cas;
   id=prxparse('/world/');
   position=prxmatch(id, 'Hello world!');
   print "pos=" position;
   prxfree(id);
run;
```

Produced in log:

```
pos=7
```

In the case above, the functions are executed in the Compute server. If you want to execute PRX functions on the CAS server, you will need to submit the program to the CAS server as an action (the action set is “sscasl.runcasl” in this example):

```
proc cas;
  sessref="MySession";
  source pgm;
     position=prxmatch('/world/', 'Hello world!');
     print 'pos=' position; run;
  endsource;
  sccasl.runcasl /
  code=pgm;
  run;
run;
quit;
```

Produced in log:

```
         pos=7
```

OTHER PROCEDURES

If the data is stored on CAS server, running the PRX functions directly on the server can improve efficiency. Generally, if a procedure is running on the server, the PRX functions called by the procedure will also run on the server.

The example using the DS2 procedure selects all car models with 2 doors by using PRXMATCH. It runs on CAS server by specifying SESSREF=. And it is a thread program that runs in parallel:

```
/* Load the sashelp.cars table to CAS server */
proc casutil outcaslib="casuser";
   load data=sashelp.cars casout="cars" replace;
run;

/* Define a method to select 2dr models and run it in parallel */
proc ds2 sessref=mysess;
 thread cars_thd / overwrite=yes;
   method run();
     set cars;
     if (prxmatch('/2dr/', model) > 0) then do;
       put make= model= msrp=;
       output;
     end;
   end;
 endthread;

 data cars_2dr / overwrite=yes;
   dcl thread cars_thd t;
   method run();
     set from t threads=4;
   end;
 enddata;

run;
quit;
```

This program uses the SQL procedure to perform a similar task. Since the table is in the CAS server, the SQL procedure will be executed on the server side:

```
libname caslib cas;
```

```
data caslib.cars;
   set sashelp.cars;
run;

proc sql;
 create table cars_2dr as
 select make, model, msrp from caslib.cars
 where prxmatch("/2dr/", model);
quit;
```

MORE METACHARACTERS

LOOK AROUND

Sometimes, if you need to check the context of the matched strings, but don't want to actually match them, you can use look around. Look around includes 4 metacharacters: look ahead positive "(?=...)", look ahead negative "(?!...)", look behind positive "(?<=...)", and look behind negative "(?<=red)", which means the left text should be "red ", so the whole pattern in the code matches a word that follows "red ":

```
data _null_;
  text="Yellow bananas, red apples, and green grapes";
  id=prxparse("/(?<=red )\w+/i");
  call prxsubstr(id, text, pos, len);
  word=ksubstr(text, pos, len);
  put word=;
run;
```

Produced in log:

```
word=apples
```

GREEDY AND LAZY REPETITION FACTORS

The repetition factors like "*", "+", and "?" are greedy repetition factors, which means the repetition factor matches a subexpression as many times as it can. Corresponding to greedy repetition factors are lazy repetition factors; they match a subexpression the minimum number of times that is needed to satisfy the match. Lazy repetition factors add a question mark after greedy repetition factors, for example, "*?" is the lazy version of "*".

For example, the pattern “/a.*b/” uses the greedy repetition factor “*”, causing the PRXMATCH to attempt to match as many characters as possible before reaching the final occurrence of “b”:

```
data _null_;
  id=prxparse("/a.*b/");
  text="abcbc";
  pos=prxmatch(id, text);
  matched=prxposn(id, 0, text);
  put matched=;
run;
```

Produced in log:

```
matched=abcb
```

In contrast, the lazy version matches the minimum number of characters possible before reaching the final “b”:

```
data _null_;
  id=prxparse("/a.*?b/");
  text="abcbc";
  pos=prxmatch(id, text);
  matched=prxposn(id, 0, text);
  put matched=;
run;
```

Produced in log:

```
matched=ab
```

When using the greedy repetition factor "*", PRXMATCH first matches as many characters as possible until it reaches the end of the string, and then backtracks when it discovers that the remaining characters cannot match the final "b". In contrast, when using the lazy repetition operator "*?", the function checks if the next character is "b" at each step. You can enable the PRX debug log to observe the difference during matching. In addition to producing different matching results, the two types of repetition operators may also result in performance differences due to their distinct matching steps.

By mastering metacharacters, you can fully harness the power of regular expressions. 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.

TAKING THE NEXT STEP

For those just starting out, the book “Unstructured Data Analysis: Entity Resolution and Regular Expressions in SAS” (2018) by K. Matthew Windham 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 by Amy Alabaster and Mary Anne Armstrong titled “Cracking Cryptic Doctors’ Notes with SAS PRX Functions” (2020) and “Interpreting Electronic Health Data Using SAS PRX Functions” (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 800 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

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

Perl regular expressions. Available at https://perldoc.perl.org/perlre

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

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