PRXChange: Accept No Substitutions

Kenneth W. Borowiak, PPD, Inc.

Abstract

SAS® provides a variety of functions for removing and replacing text, such as COMPRESS, TRANSLATE & TRANWRD. However, when the replacement is conditional upon the text around the string the logic can become long and difficult to follow. The PRXCHANGE function is ideal for complicated text replacements, as it leverages the power of regular expressions. The PRXCHANGE function not only encapsulates the functionality of traditional character string functions, but exceeds them because of the tremendous flexibility afforded by concepts such as predefined and user-defined character classes, capture buffers, and positive and negative look-arounds.

Introduction

SAS provides a variety of ways to find and replace characters or strings in character fields. Some of the traditional functions include the TRANWRD and TRANSLATE functions, but these are limited to static strings and characters. The COMPBL function can be used to reduce consecutive whitespace characters to a single whitespace character. The COMPRESS function can used to eliminate characters, and this function was enhanced in Version 9.0 to include a third argument to delete or keep classes of characters. While these traditional character functions are useful for relatively easy tasks, more difficult tasks involving text substitutions and extractions often involve nesting of functions and conditional logic, which can be cumbersome to follow and maintain.

With the release of Version 9.0, SAS has introduced Perl-style regular expressions through the use of the PRX functions and call routines. This rich and powerful language for pattern matching is ideal for text substitutions, as they allow the user to leverage predefined sets of characters, customized boundaries and manipulation of captured text. This paper explores some of the key functionality of the PRXCHANGE function. Some of the basic concepts of regular expressions (a.k.a regex) are discussed in introductory papers by Borowiak [2008] and Cassell [2007] and those unfamiliar with PRX should refer to those papers before proceeding with this paper.

The PRXCHANGE function takes the following form:

prxchange(regular expression|id, occurrence, source)

- regular expression or id : The substitution regex can either be entered directly in the first argument or a variable with the precompiled result from a call to the PRXPARSE function. The substitution regex takes the form:
  s/matching expression/replacement expression/modifiers
  The s before the first delimiter is required[1]
  The matching expression is the regex of the string to match
  The replacement expression applied to the matched string
  Modifiers control the behaviour of the matching part of the regex (i.e. such i, x or o)

[1] This is unlike the PRXMATCH function, where the m is optional before the first delimiter in the regex (e.g 'm/^\d/').

- occurrence : The number of times to perform the the match and substitution. Valid values are positive integers. A value of -1 is also valid, which performs replacement as many times as it finds the matching pattern.
- source - The character string or field where the where the pattern is to be searched.

Consider the example in Figure 1, where a new variable NAME2 is created in a PROC SQL step to replace all occurrences of the letter `a` with the letter `e`.

Figure 1 - Replacement of `a` to `e`

```
proc sql outobs=5 ;
  select   name
         , prxchange( 's/a/e/i', -1, name ) as name2
  from     sashelp.class
  order by name2
  ;
  quit ;
```

Name	name2
Barbara	B**e**rb**e**r**e**
Carol	C**e**rol
Henry	Henry
Jeffrey	Jeffrey
James	J**e**mes

Since the `i` modifier is used, it makes the pattern matching case-insensitive, so occurrences of `a` and `A` will be replaced by `e`. The value of the second argument is -1, so all occurrences of a are replaced when found in the variable NAME.

Compression

A special case of a find-and-replace operation is *compression*, where the replacement is nothing. This is an operation that is often performed using the COMPRESS function. In the query below in Figure 3, both the COMPRESS and PRXCHANGE functions are used to remove the vowels from the variable NAME into the variables NAME2 and NAME3, respectively.

Figure 3 - Remove all vowels

```
proc sql outobs=4;
   select     name
            ,  compress( name, 'aeiou', 'i' ) as name2
            ,  prxchange( 's/[aeiou]//i', -1, name ) as name3
   from       sashelp.class
```

```
  order by name3
  ;
  quit ;
```

Name	name2	name3
B**a**rb**a**r**a**	Brbr	Brbr
C**a**r**o**l	Crl	Crl
H**e**nry	Hnry	Hnry
J**u**dy	Jdy	Jdy

Now consider a slightly more restricted case where you want to remove vowels but the vowel must be preceded by the letters `l`, `m` or `n`, while ignoring case. This is relatively easy condition to implement with regular expressions by using a *positive look-behind assertion* `(?<=)`, as demonstrated in Figure 4.

Figure 4 - Remove vowels preceded by `l` ,`m` or `n`

```
proc sql ;
  select    name
          ,  prxchange( 's/(?<=[lmn])[aeiou]//i', -1, name ) as name3
  from     sashelp.class
  where   prxmatch( 'm/(?<=[lmn])[aeiou]/i', name )>0
  order by name3
  ;
  quit ;
```

Name	name3
A**l**i**e**e	Alce
Ja**me**s	Jams
Ja**ne**	Jan
Ja**ne**t	Jant
L**ou**ise	Luise
Mary	Mry
Phi**l**ip	Philp
Ro**na**ld	Ronld
Tho**ma**s	Thoms

Will**li**am	Willam

Look-arounds are *zero-width assertions* (i.e. they do not consume characters in the string) that only confirm whether a match is possible by checking conditions around a specific location in the pattern . Look-arounds can be positive or negative and can be forward or backward looking, and they are discussed in more detail in Borowiak [2006] and Dunn [2011]. To implement a solution to the problem in Figure 4 with the COMPRESS function one need to use a DO loop with the SUBSTR function to check the condition, which rules out using a SQL step.

Dedaction

Another useful type of substitution example is *dedaction*, or, the 'blacking out' of sensitive information. Consider the case where a pattern is matched and the characters are replaced by a static string, as in Figure 5 below, where you want to match the actual digits in the social security number in the free-text field and replace them with the letter `x`.

Figure 5 - Replace digits of social security numbers with an `x`

```
data SSN ;
  input SSN $20. ;
datalines ;
123-54-2280
#987-65-4321
S.S. 666-77-8888
246801357
soc # 133-77-2000
ssnum 888_22-7779
919-555-4689
call me 1800123456
;
run ;

proc sql feedback ;
   select    ssn
           , prxchange( 's/(?<!\d)\d{3}[-_]?\d{2}[-_]?\d{4}(?!\d)/xxxxxxxxx/io', -1, ssn)
                as ssn2
    from    ssn
   ;
   quit ;
```

SSN	ssn2
123-54-2280	xxxxxxxxx
#987-65-4321	#xxxxxxxxx
S.S. 666-77-8888	S.S. xxxxxxxxx

246801357	xxxxxxxxx
soc # 133-77-2000	soc # xxxxxxxxx
ssnum 888_22-7779	ssnum xxxxxxxxx
919-555-4689	919-555-4689
call me 1800123456	call me 18001234567

Capture Buffers

Like many other programming languages, regular expressions allow you to use parenthesis to add clarity by grouping logical expressions together. A result of using grouping parentheses is that it creates temporary variables, which can be used in the substitution part of a regex. Consider an extension of the example in Figure 5 where we continue to replace the digits of social security numbers with an `x`, but want to maintain any of the dashes or underscores in the original variables.

Figure 6 - Maintain dashes and underscores in social security number dedaction

```
proc sql feedback ;
  select    ssn
          , prxchange('s/(?<!\d)\d{3}(-|_)?\d{2}(-|_)?\d{4}(?!\d)/xxx$1xx$2xxxx/io', -1
              , ssn ) as ssn2
  from      ssn
  ;
quit ;
```

SSN	ssn2
123-54-2280	xxx-xx-xxxx
#987-65-4321	#xxx-xx-xxxx
S.S. 666-77-8888	S.S. xxx-xx-xxxx
246801357	xxxxxxxxx
soc # 133-77-2000	soc # xxx-xx-xxxx

ssnum 888_22-7779	ssnum xxx_xx-xxxx
919-555-4689	919-555-4689
call me 18001234567	call me 18001234567

Note the regular expression actually has four pairs of parentheses. The negative look-behind (i.e. `(?<!\d)`) and negative look-ahead assertions (i.e. `(?!\d)`) are *non-capturing*. The sub-expression `(-|_)`, which appears the second and third pairs of parentheses, are the two capturing parentheses, which correspond to the variables `$1` and `$2`. And though the match of the sub-expressions in the two capturing parentheses are optional, as they are followed by the `?` quantifier, the variables `$1` and `$2` can be populated with null values.

Case-Folding Prefixes and Spans

When performing a replacement, users have the ability to manipulate captured variables by changing their case. Consider the example in Figure 7, where the titles `Dr`, `Mr`, and `Mrs` are entered in various ways but need to be 'propcased'.

Figure 7 - Propcase courtesy titles

```
data guest_list ;
 input attendees $30. ;
datalines;
MR and MRS DRaco Malfoy
mr and dr M Johnson
MrS. O.M. Goodness
DR. Evil
mr&mrs R. Miller
;
run ;

proc sql ;
  select   attendees
         , prxchange( 's/\b(d|m)(r(?!a)s?)/\u$1\L$2\E/io', -1, attendees ) as attendees2
  from    guest_list
  ;
  quit ;
```

attendees	attendees2
MR and MRS DRaco Malfoy	**Mr** and **Mrs** DRaco Malfoy
mr and dr M Johnson	**Mr** and **Dr** M Johnson

MrS. O.M. Goodness	**Mrs**. O.M. Goodness
DR. Evil	**Dr**. Evil
mr&mrs R. Miller	**Mr**&**Mrs** R. Miller

The regular expression looks for a `d` or `m` at the beginning of a word and the matched character is put in the first capture buffer. It then looks to match `r` (as long as it is not followed by an `a`) followed an optional `s` and puts the characters in the second capture buffer. The replacement expression then makes use of *case-folding prefixes* and *spans* to control the case of the capture buffers. `\u` makes the next character that follows it uppercase, which would be the `d` or `m` in first capture buffer. The case-folding span `\L` makes characters that follow lowercase until the end of the replacement or until disabled by `\E`, which is applied to the second capture buffer. Case-folding functionality was introduced in SAS V9.2.

More on Lookarounds

The final example in Figure 9 demonstrates inserting text at a location where both a positive look-behind `(?<=)` and look-ahead `(?=)` assertion is satisfied. Consider the task where a space character is to be inserted between two consecutive colons. One might be tempted to write a regex to match the two colons and replace it with three characters (i.e : :). However, an efficient regex would be able to identify the location that is preceded and followed by a colon and insert a single space character.

Figure 8 - Inserting a space between consecutive colons

```
data colons;
 length string string2 $200 ;
 string='a::::b:::::';
 string2=prxchange("s/(?<=:)(?=:)/ /",-1,string) ;
 run ;
```

string	string2
a::::b:::::	a: : : :b: : : : :

This solution will not provide the correct result in versions prior to SAS V9.2, as there was bug in the Perl 5.6.1 engine that the PRX functions use.

Conclusion

For simple text substitutions, using traditional SAS functions may suffice. However, as a substitution task becomes more complicated, multiple lines of code can often be reduced to a single regular expression within PRXCHANGE due to the tremendous flexibility they offer.

References

Borowiak, Kenneth W. (2006), "Perl Regular Expressions 102". Proceedings of the 19th Annual Northeast SAS Users Group Conference, USA.
http://www.nesug.org/proceedings/nesug06/po/po16.pdf

Borowiak, Kenneth W. (2008), "PRX Functions and Call Routines: There is Hardly Anything Regular About Them!". Proceedings of the Twenty First Annual Northeast SAS Users Group Conference, USA.
http://www.nesug.org/proceedings/nesug08/bb/bb11.pdf

Cassell, David L., "The Basics of the PRX Functions" SAS Global Forum 2007
http://www2.sas.com/proceedings/forum2007/223-2007.pdf

Dunn, Toby, "Grouping, Atomic Groups, and Conditions: Creating If-Then statements in Perl RegEx" SAS Global Forum 2011
http://support.sas.com/resources/papers/proceedings11/245-2011.pdf

Friedl, Jeffrey E.F., *Mastering Regular Expressions 3rd Edition*

Acknowledgements

The authors would like to thank Jenni Borowiak, Kipp Spanbauer and Kunal Agnihotri for their insightful comments on this paper.

Disclaimer

The content of this paper are the works of the author and do not necessarily represent the opinions, recommendations, or practices of PPD, Inc.

Contact Information

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

Ken Borowiak
3900 Paramount Parkway
Morrisville NC 27560

ken.borowiak@ppdi.com
ken.borowiak@gmail.com