MWSUG 2018 – Paper SB-145

Perl Regular Expression – The Power to Know the PERL in Your Data

Kaushal Chaudhary, Eli Lilly and Company, Indianapolis, IN

Dhruba Ghimire, Eli Lilly and Company, Indianapolis, IN

ABSTRACT

Perl regular expression is one of the powerful and efficient techniques for complex string data manipulation. SAS® offers regular expression engine in the base SAS without any additional license requirement. This would be a great addition to a SAS programmers' toolbox. In this paper, we present basics of the Perl regular expression and various Perl regular functions and call routine such as PRXPARSE(), PRXMATCH(), and CALL PRXCHANGE () etc. with examples. The presentation is intended for beginner and intermediate SAS programmers.

INTRODUCTION

Regular expression is a great tool for text data manipulation. Many other programming languages have regular expression engine in them to facilitate text data analysis. SAS also introduced regular expression since version 9. SAS has few functions and call routines available to use the regular expression. Some of these functions perform similar functions as regular SAS character functions such as substr () and scan (), however, in many situations they add extra flexibility as compared to those. The prxparse () function enables to handle the complex string using the wealth of metacharacters and types of regular expression as argument.

A non-exhaustive list of metacharacters and regular expressions (character class, grouping, alternation, repetition, and anchored expressions) are presented in table 1. A thorough understanding of these are necessary to master the regular expressions for any programming languages as they are more or less similar across languages.

Metacharacters and Regular Expressions

Metacharacters are characters that have special meaning at regular expressions. They are escaped with backward slash (\) to match literally, for example, \. would match '.'. The following table has the metacharacters and different type of regular expressions with description, and example use.

Table 1

Metacharacters/regular expression types	Description	Example
.	Matches any one character	Matches 1, a etc.
+	Matches the preceding character one or more times	abc+ matches 'abc', 'abcc', 'abccc' etc.
?	Matches the preceding character zero or one time	abc? matches 'abc' , 'ab'
*	Matches the preceding character zero or more times	abc* matches 'ab', 'abc', 'abcc', 'abccc'
+?	Matches at least as possible	\a+? matches 'a' in text 'aaaaa'.
\d	Matches digit characters	\d matches '1' in 'a1'
\D	Matches non-digit characters	\D matches 'a' in 'a1'
\w	Matches word characters	\w matches 'a' in 'a1'
\W	Matches non-word characters	\W matches '1' in 'a1'
\s	Matches white space	
\S	Matches nonwhite space	

Metacharacters/regular expression types	Description	Example
\b	Word boundary	\bMWSUG\b matches 'MWSUG' in 'This is MWSUG 2018 but not in 'MWSUG2018'.
\B	Non word boundary	
^	Matches at the beginning of the string	^This is MWSUG 2018. matches 'This'
$	Matches at the end of the string	This is MWSUG 2018$. matches 2018
\(	Matches '('	\(MWSUG matches (MWSUG
\)	Matches ')'	MWSUG)\ matches MWSUG)
\\	Matches '\'	abc\\123 matches abc\123
[abc]	Character set	Matches a, b, or c
[^abc]	Character set	Matches other than a, b, or c
[A-Z1-9]	Character set	Matches all alphabets and digits
()	Grouping	/(abc)+/ matches 'abc' and 'abcabcabc'
|	Alternation	/abc|def/ matches 'abc' or 'def'
\d{m}	Quantified expression- matches m number of digits	\d{2} matches '12'
\d{m,}	Quantified expression – matches at least m number of digits	\d{2,} matches '12', '123'
\d{m, n}	Quantified expression – matches minimum m and maximum n number of digits	\d{2,3} matches '12', '123'
\w{m}	Quantified expression- matches m number of word characters	\w{2} matches 'ab'
\w{m,}	Quantified expression- matches at least m number of word characters	\w{2,} matches 'ab', 'abc'
\w{m, n}	Quantified expression- matches m minimum number of word characters and n maximum number of word characters	\w{2,3} matches 'ab' and 'abc'
[[:alpha:]]	POSIX character expressions	Matches all alphabets (a-z A-Z) and underscore (_).
[[:digits:]]	POSIX character expressions	Matches all digits (0-9)

FUNCTIONS

PRXPARSE (), PRXMATCH (), PRXCHANGE (), PRXPOSN (), and PRXPAREN () will be illustrated below with examples.

PRXPARSE

Syntax: PRXPARSE (Perl-regular-expression)

Perl-regular-expression: The pattern to be parsed

PRXPARSE uses metacharacters to construct the regular Perl expression. It compiles a Perl regular expression that can be used by other Perl regular expression functions/call routines for pattern matching of a character value.

Program 1

```
data have;
     input patient $1-15 string $50.;
     datalines;
     WWW-100-01001  CERVICAL PAIN (MUSCULAR)
     XXX-200-02001  MUSCULO-SQUELETTIC PAIN
     YYY-300-03001  back pain
     ZZZ-400-04001  ABDOMINAL PAIN CHEST PAIN
  ;
run;

data want;
     set have;
     retain pattern;

     if _n_ = 1 then
           pattern = prxparse('/pain/I');
     pos = prxmatch (pattern,string);
run;
```

In the program above, regular expression (pattern) is created during the first iteration (_n_ = 1) of the data step and retaining it. Another alternative would be using modifier 'o'. The example program has single regular expression, however, multiple regular expressions can be created in a single data step. Modifier 'i' makes the pattern matching case-insensitive and matches all string having 'pain' or 'PAIN'.

PRXMATCH

Syntax: PRXMATCH (pattern-id or regular-expression, string)

Pattern-id: Returned value from PRXPARSE function, regular-expression: Perl regular expression, string: Character value

PRXMATCH function is used to search a pattern match and returns the position at which the pattern is found. If there is no match found, PRXMATCH returns a zero but if there are multiple matches found, only the position of the first match is returned.

Program 2

```
data want;
     set have;

     if _n_ = 1 then
           pattern = prxparse('/PAIN/');
     retain pattern;
     pos = prxmatch (pattern, string);
run;
```

Output:

	patient	string	pattern	pos
1	WWW-100-01001	CERVICAL PAIN (MUSCULAR)	1	10
2	XXX-200-02001	MUSCULO-SQUELETTIC PAIN	1	20
3	YYY-300-03001	back pain	1	0
4	ZZZ-400-04001	ABDOMINAL PAIN CHEST PAIN	1	11

Case I: In string CERVICAL PAIN (MUSCULAR), the position of the first character of the pattern match (PAIN) returns 10.
Case II: In string back pain, no pattern found and returns to zero.
Case III: In ABDOMINAL PAIN CHEST PAIN, pattern matches twice but the position of the first match returns to 11.

PRXCHANGE

Syntax: PRXCHANGE (Perl-regular-expression |regular-expression-id, times, source)

Perl-regular-expression: The pattern to be parsed, regular-expression-id: Returned value from PRXPARSE function, times: The number of times to perform the match and substitution
Source: The character string where the pattern is to be searched

The PRXCHANGE function performs a replacement for a matched pattern. The ‘s’ before the first delimiter indicates substitution in the code. The first argument of the function has two components-find and replace.

Program 3

```
data want;
    set have;
    update = prxchange('s/ pain/ ACHE/I',  -1, string);
run;
```

Output:

	patient	string	update
1	WWW-100-01001	CERVICAL PAIN (MUSCULAR)	CERVICAL ACHE (MUSCULAR)
2	XXX-200-02001	MUSCULO-SQUELETTIC PAIN	MUSCULO-SQUELETTIC ACHE
3	YYY-300-03001	back pain	back ACHE
4	ZZZ-400-04001	ABDOMINAL PAIN CHEST PAIN	ABDOMINAL ACHE CHEST ACHE

In program 3, ‘pain’ is replaced by ‘ACHE ‘from the string. The modifier ‘i’ makes the string case-insensitive so that all ‘PAIN’ from the string are also replaced here. The second argument -1 indicates that all occurrences are replaced when found in the variable string.

PRXPOSN

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

Regular-expression-id: Returned value from PRXPARSE function, capture-buffer: Number indicating which capture buffer is to be evaluated, source:The character string where the pattern is to be searched.

PRXPOSN function returns the matched information from identified capture. PRXMATCH, PRXSUBSTR, PRXNEXT or PRXCHANGE functions are used before PRXPOSN function to reference the capture buffer. In addition, regular expression id is required for this function.

Program 4

```
data want;
     length study site patid $ 10;
     keep study site patid;
     retain re;

     if _n_ = 1 then
           re = prxparse('/(\w+)-(\d{3})-(\d{5})/');
     set have;

     if prxmatch(re, patient) then
           do;
                 study = prxposn(re, 1, patient);
                 site = prxposn(re, 2, patient);
                 patid=prxposn(re, 3, patient);
           end;
run;
```

output:

	study	site	patid
1	WWW	100	01001
2	XXX	200	02001
3	YYY	300	03001
4	ZZZ	400	04001

In program 4, the regular expression id ‘re’ is created using PRXPARSE function. If the match exists, capture buffers 1, 2, 3 are used to extract study, site and patid from the source (Patient) using PRXPOSN function.

PRXPAREN

Syntax: PRXPAREN (regular-expression-id)

Regular-expression-id**:** Returned value from PRXPARSE function

PRXPAREN function returns a value of the largest capture buffer that contains the data of the first match. PRXMATCH, PRXSUBSTR, PRXNEXT or PRXCHANGE functions (routines) are used with PRXPAREN together. It requires the regular expression id rather than the regular expression.

Program 5

```
data want;
     set have;
     pattern = prxparse ('/(PAIN)|(CERVICAL)|(ABDOMINAL)/');
     pos = prxmatch (pattern, string);
     if pos then paren=prxparen(pattern);
run;
```

Output:

	patient	string	pattern	pos	paren
1	WWW-100-01001	CERVICAL PAIN (MUSCULAR)	1	1	2
2	XXX-200-02001	MUSCULO-SQUELETTIC PAIN	1	20	1
3	YYY-300-03001	back pain	1	0	.
4	ZZZ-400-04001	ABDOMINAL PAIN CHEST PAIN	1	1	3

In program 5, 'PAIN', 'CERVICAL', 'ABDOMINAL' are enclosed by parenthesis in the pattern to create capture buffer location. In the first observation, CERVICAL matches in the second parenthesis of the pattern with pos = 1. In the second observation, PAIN matches in the first parenthesis with pos = 20, however, in the third observation, pain does not match in the pattern so that the paren is missing.

CALL ROUTINES

Some of the Perl Regular functions have their call routine counterpart. There call routines are similar to the functions, but they yield more information. We will discuss some the commonly used call routines next.

CALL PRXCHANGE

Syntax: CALL PRXCHANGE (regular-expression-id, times, old-string, new-string, result-length, truncation-value, number-of-changes)

Regular-expression-id: Unique numeric regular expression id, times: Number of times the matching patterns replaced, old-string: Source text string, new-string: New variable created after matching pattern replaced, result-length: a numeric variable representing the number of characters that are copied into the result, truncation-value: The Boolean value (1 or 0) whether replacement result is longer than new string.

CALL PRXCHANGE () is similar to the PRXCHANGE () function. It, however, takes only regular expression id as argument and can also create a new variable (new string as in the syntax) after replacing the desired pattern.

In program 6, we are replacing '2018' by '2019' from the txt variable. 'newtxt' variable is created to store the new string. In program 7, the resultant string will be stored in txt variable without creating any new variable. We can also change the order of the parts of the string by creating capture groups and referencing them by the numbers respective to their position in the regular expression pattern preceded with dollar sign within the same expression as shown in program 8. The 'newtxt' variable has reversed order of the original text.

Program 6

```
data have;
    txt = 'MWSUG 2018';
run;

data want;
    length newtxt $ 14.;
    set have;
    retain re;
    if _n_ = 1 then re = prxparse('s/\d+/2019/');
    call prxchange(re, -1, txt, newtxt);
    keep txt newtxt;
```

```
run;
```

Output:

	newtxt	txt
1	MWSUG 2019	MWSUG 2018

Program 7

```
data want;
     set have;
     retain re;
     if _n_ = 1 then re = prxparse('s/\d+/2019/');
     call prxchange(re, -1, txt);
run;
```

Output:

	txt
1	MWSUG 2019

Program 8

```
data want;
     length newtxt $ 14.;
     set have;
     retain re;
     if _n_ = 1 then re = prxparse('s/(\w+)\s(\d+)/$2 $1/');
     call prxchange(re, -1, txt, newtxt);
     keep txt newtxt;
run;
```

Output:

	newtxt	txt
1	2018 MWSUG	MWSUG 2018

CALL PRXPOSN

Syntax: CALL PRXPOSN (regular-expression-id, capture-buffer, start, length)

Regular-expression-id: Unique numeric regular expression id, capture-buffer: A numeric variable for representing the number of capture buffer, start: A numeric variable for the position of the capture buffer, length: A numeric variable for the length of the capture buffer.

CALL PRXPOSN () creates the position and length of the capture buffer as variables thus enabling us to extract the desired part of the string using regular SAS functions such as substr() or substrn() later. In program 8, we have word as capture buffer 1 and digits as capture buffer 2. Based on the position and

length of these capture buffers we can extract the substring representing those capture buffers. CALL PRXPOSN () is used after matching pattern is found by PRXMATCH ().

Program 9

```
data want;
     set have;
     retain re;
     if _n_ = 1 then re = prxparse('/(\w+)\s(\d+)/');

     if prxmatch(re, txt) then do;

     call prxposn(re, 1, pos, len);
     call prxposn(re, 2, pos1, len1);
     Conf_name = substr(txt, pos, len);
     Conf_year = substr(txt, pos1, len1);
     end;
     keep txt Conf_name Conf_year;
run;
```

Output:

	txt	Conf_name	Conf_year
1	MWSUG 2018	MWSUG	2018

CALL PRXNEXT

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

Regular-expression-id: Unique numeric regular expression id, start: A numeric variable for the start position to find the matching pattern, stop: A numeric variable for the position of last character to find the matching pattern, source: The input text, position: A numeric variable where matching pattern is found, length: A numeric variable for the length of string matched by pattern.

CALL PRXNEXT () searches for the given pattern of a substring repeatedly yielding the position and length of the each matching pattern in the string. In program below, we are looking for words followed by space.

Program 10

```
data have;
     txt = 'This is MWSUG 2018';
run;

data _null_;
     set have;
     retain re;
     if _n_ = 1 then re = prxparse('/\w+\s/');

     start = 1;
     stop = length(txt);
     call prxnext(re, start, stop, txt, pos, len);
           do while (pos > 0);
                 found = substr(txt, pos, len);
```

```
            put found = pos = len =;
            call prxnext(re, start, stop, txt, pos,len);
        end;

run;
```

Log output:

```
found=This pos=1 len=5
found=is pos=6 len=3
found=MWSUG pos=9 len=6
```

CALL PRXSUBSTR

Syntax: CALL PRXSUBSTR (regular-expression-id, source, position, length)

Regular-expression-id: Unique numeric regular expression id, source: The input text, position: A numeric variable where matching pattern is found, length: A numeric variable for the length of string matched by pattern.

CALL PRXSUBSTR () finds the location and length of the matching pattern substring we are interested in a given character string. Two numeric variables position, and length as in the syntax are created. Once we know those two parameters, substring can be extracted.

Program 11

```
data have;
    txt = 'MWSUG 2018';
run;

data want;
    set have;
    retain re re1;
    length Conf_: $ 50.;

     if _n_ = 1 then
          do;
               re = prxparse('/\w+/');
               re1 = prxparse('/\d+/');
          end;


    call prxsubstr(re, txt, pos, len);
    call prxsubstr(re1, txt, pos1, len1);

    if pos ^= 0 then Conf_name = substr(txt, pos, len);

    if pos1 ^= 0 then Conf_year = substr(txt, pos1, len1);

    keep txt Conf_:;
run;
```

Output:

	txt	Conf_name	Conf_year
1	MWSUG 2018	MWSUG	2018

CONCLUSION

In this paper, we introduced Perl Regular Expression in SAS with functions and call routines. We used rather simple examples to explain them lucidly. Hopefully, this get you started to use them and explore more in depth. Soon you will find this is powerful.

REFERENCES

1. Windham, K. Matthew. 2014. Introduction to Regular Expressions in SAS®. Cary, NC: SAS Institute Inc.
2. Cody, Ron. An Introduction to Perl Regular Expression in SAS 9, Proceedings of the 29th Annual SAS Users Group International.
3. Pless, Richard. An Introduction to Regular Expressions with Examples from Clinical Data, Proceedings of the 29th Annual SAS Users Group International
4. SAS Institute. (2010). SAS® 9.4 Functions and CALL Routines Reference. Cary, NC: SAS Institute.

CONTACT INFORMATION

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

Kaushal Raj Chaudhary
Eli Lilly and Company
Lilly Corporate Center, Indianapolis
Email: Chaudhary_kaushal_raj@lilly.com

Dhruba R Ghimire
Eli Lilly and Company
Lilly Corporate Center, Indianapolis
Email: ghimire_dhruba_r@lilly.com