PharmaSUG China 2024 - Paper AP-10051

Perl Regular Expressions using in SAS

Ping Gong, Everest Corporation

ABSTRACT

Perl regular Expressions (regex) can be applied to many languages environments and has powerful string handling capabilities when combined with PRX functions in SAS environments. This paper will introduce some basic and advanced syntax of regex (Greedy and Lazy, Backreferences, Lookahead, and Lookbehind Zero-Length Assertions). This paper will also illustrate how to apply regex by using PRX functions (e.g. PRXPARSE, PRXMACTH, PRXCHANGE, PRXSUBSTR and PRXPOSN) to find, replace, and extract characters in SAS environment with examples. These are difficult to achieve by using INDEX, TRANWRD, SUBSTR functions. The goal of this paper is to help programmers to get a handle of regex, understand when can regex be used and how to apply it in SAS environment. SAS®9.4 M7 software are used in examples presented.

INTRODUCTION

Basically, a regular expression (regex) is a pattern describing a certain amount of text. Their name comes from the mathematical theory on which they are based. As usual in the software world, different regular expression engines are not fully compatible with each other. The syntax and behavior of a particular engine is called a regular expression flavor.

As we known, regex can be used in SAS by calling PRX function, which is a powerful character processing function. In our daily work, we often encounter some complex strings. When we need to extract or replace them, the general SAS function has limitations and cannot be processed in batches. Therefore, we often need to use PRX function to achieve this result, but calling PRX function requires a basic understanding of regex and we often neglect how to properly write regex. This paper will introduce the basic and advanced syntax of regex. It also shows how to apply it in PRX function with specific examples.

METACHARACTERS

In regex, a metacharacter is a special character that has a specific meaning that you can build complex patterns to match a wide range of combinations. For example, the dot or period is one of the most commonly used metacharacters in regular expressions.

The dot matches a single character, without considering which character it is. When we use dot to match “Hello World!”, each letter, including spaces and exclamation marks, will be matched, but when we used \w will only match each letter, \s will match space. Metacharacters are defined differently in different development environments, Table 1 listed some metacharacters can be used in SAS environment.

Metacharacters	Description
.	matches any characters.
\w	matches any word character or alphanumeric character, including the underscore.
\s	matches any whitespace character, including space, tab, form feed, and so on, and is equivalent to [\f\n\r\t\v].
\d	matches a digit character that is equivalent to [0−9].
\b	matches a word boundary (the position between a word and a space).
\r	matches a return character.
\n	matches a newline character.

\t	matches a tab character.

Table 1. Metacharacters

The metacharacters listed in Table 1 are often represent to antonymy when capitalized, for example when we used \S to match “Hello World!”, any character except spaces will be matched. Table 2 listed some antonymy metacharacters.

Metacharacters	Description
\W	matches any non-word character or nonalphanumeric character, and excludes the underscore.
\S	matches any character that is not a whitespace character [^\f\n\r\t\v].
\D	matches any character that is not a digit.
\B	matches a non-word boundary.
[^x]	matches a character is not “x”.
[^a-z]	matches any characters that is not in the range “a” through “z”.

Table 2. Antonymy Metacharacters

There are some qualifiers used in regex, “\” marks the next character as either a special character, a literal, a back reference, or an octal escape (*, \(, \?, \n, \x0A), “^” matches the position at the beginning of the input string, “$” matches the position at the end of the input string, “^\d$” can only match one digit like “1”, “2”, and cannot match “abc1”, “1abc”, “ab1c”, or “123”.

REPETITION

As we know that “\d” can match the number, when we need to match the phone number which usually have 11 digits, we can write the regex as “\d\d\d\d\d\d\d\d\d\d\d”. It seems redundant, and we can take advantage of repetition factors in regular expressions and use “\d{11}”. Some of the repetition factors are listed in Table 3.

Metacharacters	Description
*	matches the preceding subexpression zero or more times.
+	matches the preceding subexpression one or more times.
?	matches the preceding subexpression zero or one time.
{n}	matches *n* times.
{n,}	matches a pattern at least *n* times.
{n,m}	m and n are nonnegative integers, where n<=m. They match at least *n* and at most *m* times.

Table 3. Repetition Factors

GROUPING

We can group part of the regular expression together by placing part of a regular expression inside round brackets or parentheses. This allows us to apply a quantifier to the entire group or to restrict alternation to part of the regex.

Only parentheses can be used for grouping. Square brackets define a character class, and curly braces are used by a quantifier with specific limits.

For the Example 1 the data set named *old_id* and include 4 different ID numbers in variable *oldid* and we want to capture the numbers of the innermost bracket between 2 hyphens as *newid*, we can use grouping

of regex and call PRX function to achieve this. Expression '(\d+)' is a grouping, can use PRXPOSN function to extract it.

oldid	newid
100-(3456)-789	3456
900-(3(4567))-900	4567
(300)-(890)-123	890
345-(456(3))-909	3

Example 1

SAS code to achieve the purpose of Example 1:

```
data test1;
  set old_id;
  regex=prxparse('/.+\-.*\((\d+)\)?\-.+/');
  if prxmatch(regex, oldid) then do;
       call prxposn(regex, 1, position, length);
       newid=substr(oldid, position, length);
    end;
run;
```

GREEDY AND LAZY

When a regex contains a qualifier that accepts repeats, the usual approach is to match as many characters as possible (provided that the entire expression is matched). Take this expression: *a.*b*, which matches the longest string that starts with *a* and ends with *b*. If you use it to search for *'aabab'*, it matches the entire string *'aabab'*. This is called greedy matching. A greedy matching also be used in Example 1, regex '\-.*\(' means match the longest string starts with '-' and ends with '(', so the innermost bracket will be matched.

Sometimes, we need lazy matching, which means matching the shortest possible string. Any of the qualifiers given above can be used to a lazy match pattern by adding a *'?'* in the end. *'.*?'* means matching any number of duplicates, but using the least number of duplicates necessary to make the entire match successful. *a.*?b* matches the shortest string that starts with *a* and ends with *b*. If applied to *'aabab'*, it matches *'aab'* (first to third characters) and *'ab'* (fourth to fifth characters). Table 4 is a list of the lazy repetition factors.

Syntax	Description
*?	matches a pattern zero or more times.
+?	matches a pattern one or more times.
??	matches a pattern zero or one time.
{n,}?	matches a pattern at least *n* times.
{n,m}?	matches a pattern at least *n* times but not more than *m* times.

Table 4. Lazy Repetition Factors

From the introduction above, we can read or write most of the regex, but sometimes we also need some advanced syntax.

BACKREFERENCES

Backreferences match the same text as previously matched by a capturing group. Suppose you want to match a string like *'axaxa', 'bxbxb'* and *'cxcxc'*, you can use expression '([a-c])x\1x\1'. The backreference \1 (backslash one) references the first capturing group. \1 matches the exact same text that was matched by the first capturing group. And in our daily work, if you want to extract the unit from a string like *'500mg/30mg/6mg'*, you can use regex like '\d+([a-z]+)\/\d+\1\/\d+\1' to match it and then calling PRX function as below, the return value of unit is mg, the expression '([a-z]+)' is a grouping match to *'mg'*, and

backreference \1 references the first capturing group.

SAS code to extract the unit from string '*500mg/30mg/6mg*':

```
data test;
   string='500mg/30mg/6mg';
   re=prxparse('/\d+([a-z]+)\/\d+\1\/\d+\1/');
   if prxmatch(re,string) then do;
      unit=prxposn(re,1,string);
    end;
run;
```

Most regex flavors support up to 99 capturing groups and double-digit backreferences. So \99 is a valid backreference if your regex has 99 capturing groups.

LOOKAHEAD AND LOOKBEHIND ZERO-LENGTH ASSERTIONS

Lookahead and lookbehind, are zero-length assertions, actually matches characters, but then gives up the match, returning only the result match or no match. That is why they are called "assertions". They do not consume characters in the string, but only assert whether a match is possible or not.

POSITIVE AND NEGATIVE LOOKAHEAD

Positive lookahead is indispensable if you want to match something followed by something else. For example, if you want match a word which followed by **'***ing'*, then you can use expression **'**\b\w+(?=ing\b)**'**, it can match the word *dancing*, *sing*.

Negative lookahead is indispensable if you want to match something not followed by something else. For example, if you want to match a word which not followed by '*ing*', then you can use expression '\b(\w(?!ing))+\b', it doesn't match the word *dancing*, *sing*.

POSITIVE AND NEGATIVE LOOKBEHIND

Lookbehind has the same effect, but works backwards. It tells the regex engine to temporarily step backwards in the string, to check if the text inside the lookbehind can be matched there. Positive lookbehind, like expression '(?<=\bre)\w+\b' can match a word starting with *"re"*, for example the word *reading*. Negative lookbehind, like expression '\b((?<!re)\w)+\b' can match a word not starting with *"re"*. '(?<=(abc)).*(?=\1)' will match the string *abcxxxabc*.

This is the introduction of the basic and advanced syntax of regex, and the following will be combined with examples to show how it is applied in our daily work in SAS environment.

EXAMPLES

EXAMPLE 1: FIND THE UNREADABLE CODES

When we create the SDTM datasets using raw datasets, we often get some unreadable codes from the source datasets. These unreadable codes are non-ASCII characters. We need to find these error codes and then query to DM after data extraction. We can use PRX function to do a quick search.

For example, some error codes appear in CM.CMTRT (reported name of drug) as Table 5, we can use character class grouping expression [[:^ascii:]] to match the string with error codes, and use PRXSUBSTR function to print the position of the error codes. Firstly, we identify the perl-regex as '/[[:^ascii:]]/' which can match a non-ASCII character, and then we call PRXSUBSTR function to get the non-ASCII character position in a string, like the first record the error codes appear at the 10th bit, and the value of position returns 10.

CMTRT	POSITION
PIROXICAMÂ Â BETA CYCLODEXTRINE	10
PARACÃ?TAMOL	6

Table 5. Unreadable codes in CM.CMTRT and return the position values by using PRXSUBSTR

SAS code to find the position of unreadable codes in CM.CMTRT:

```
data test2;
  set raw.cm;
  re=prxparse('/[[:^ascii:]]/');
  call prxsubstr(re,cmtrt,position,length);
run;
```

EXAMPLE 2: REPLACE THE SPECIAL CHARACTERS

In addition to the unreadable characters, we usually can find a newline character or non-print space in one record from the raw dataset, they can be replaced with space, we can also use the regex to match these special characters and replace them.

For example, a newline character appeared after 'KMNO4 - POTASSIUM' in CM.CMTRT as Table 6, we can use metacharacter '\n' to match it, and then call PRXCHANGE function to replace it as a space, CMTRT_ is the new value. In this function, 's/\n/ /' means we need to use space replaced the newline character, the value between first two slash is the regex for what need to be replaced, and the value between second two slash is new value, and '-1' means that the character need to be replaced from the start to the end.

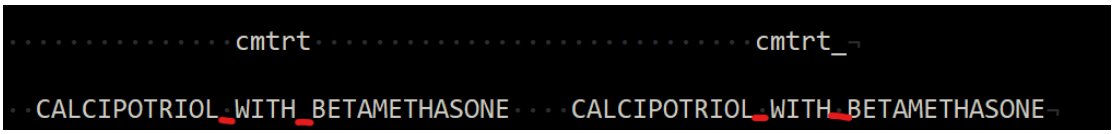
CMTRT	CMTRT_
KMNO4 - POTASSIUM PERMANGANATE	KMNO4 - POTASSIUM PERMANGANATE

Table 6. Before and after replace the newline character from CM.CMTRT

SAS code to replace the newline character from CM.CMTRT:

```
data test3;
  set raw.cm;
  cmtrt_=prxchange('s/\n/ /',-1,cmtrt);
run;
```

We can also call this function to replace the non-print space. For example as Display 1, one space display as one gray dot in the editor, and the CM.CMTRT has one non-print space after the word "WITH" so there is no gray dot appeared. The non-print space is also a non-ASCII character, we can use '/[[:^ascii:]]/' to match, and call PRXCHANGE function to replace it by a space and get the new value in CMTRT_, as the normal space appears after the word "WITH".



Display 1. Before and after replace the non-print space from CM.CMTRT

SAS code to replace the non-print space from CM.CMTRT:

```
data test4;
  set raw.cm;
  cmtrt_=prxchange('s/[[:^ascii:]]/ /',-1,cmtrt);
run;
```

EXAMPLE 3: EXTRACT THE CHARACTERS

Often the medication dose and unit are collected as one variable in the source data and we want to extract the characters as unit from CM.CMDOSU as Table 7, we can create a regex to match the string and then used PRXPOSN function to extract the character as unit. For example, we used the '/\d+\/?\d*(\w+\/?[A-Z a-z]+)/' as regex. The expression '\d+\/?\d*' match the pattern like one or more digits with or without a slash, like 1, 150 or 150/30; '\w+\/?[A-Z a-z]+' match the pattern like one or more words with or without a slash, like g, mg, or mg/g but cannot match mg/30 since the expression restrict the slash must be followed by words. The overall expression matches the pattern one or more digits with or without a slash and follow one or more words with or without a slash. And then we use PRXMACTH function to filter the records that match the regex and use PRXPOSN function to extract the first group which is the value in parentheses.

CMDOSU	UNIT
150/30mcg	mcg
150mg	mg
100mcg	mcg
500mg	mg
10mg	mg
500mg/30mg/6mg	mg
50mg/g	mg/g
55mcg/spray	mcg/spray

Table 7. CM.CMDOSU and extracted unit results

SAS code to extract the unit from CM.CMDOSU:

```
data test5;
  set raw.cm;
  re=prxparse('/\d+\/?\d*(\w+\/?[A-Z a-z]+)/');
  if prxmatch(re,cmdosu) then do;
    unit=prxposn(re,1,cmdosu);
  end;
run;
```

CONCLUSION

Perl regular expression can be used in many scenarios. This paper only listed some problems I encountered in my daily job. Perl regular expression can be very powerful in processing the data if programmers know how to use it.

REFERENCES

Windham, K. Matthew. 2014. Introduction to Regular Expressions in SAS®. Cary, NC: SAS Institute Inc.

Jan Goyvaerts. "Regular-Expression Info". 2003-2024. https://www.regular-expressions.info/repeat.html

CONTACT INFORMATION

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

Ping Gong
Everest Clinical Research
Email: ping.gong@ecrscorp.com