

Paper CC06

Using PRX to Search and Replace Patterns in Text Strings

Wenyu Hu, Merck Research Labs, Merck & Co., Inc., Upper Gwynedd, PA
Liping Zhang, Merck Research Labs, Merck & Co., Inc., Upper Gwynedd, PA

ABSTRACT

Programmers often need to search for patterns in text strings in order to change specific text. Perl regular expressions (PRX) introduced in SAS® version 9 provides a convenient and powerful tool to locate, extract and replace text strings in DATA step. PRX can provide simple solutions to complex string manipulation tasks and are especially useful for reading highly unstructured text strings. This paper explains the basics of PRX and how PRX functions work in SAS 9. It further explains how to code useful PRX functions and to use them to search and replace patterns.

Keywords: Perl regular expressions (PRX), Regular expressions (RX), Pattern match

INTRODUCTION

One may wonder about the need to use regular expressions when there is a rich set of string manipulation functions available in SAS. Most of the string processing tasks could be accomplished by using traditional string character functions. However there are situations where patterns in the text are so complex that it takes an advanced programmer to write many lines of codes to build sophisticated logic using INDEX, SUBSTR and other string manipulation functions. Situations like these are where regular expression functions come into use.

Regular expressions allow searching and extracting multiple pattern matches in the text string in one single step. It can also make several string replacements. SAS regular expressions (RX functions, i.e. RXPARSE, RXCHANGE and RXMATCH) have been around for a while. Version 9 introduces the PRX functions and call routines. They include PRXPARSE, PRXCHANGE, PRXMATCH, CALL PRXCHANGE, CALL PRXSUBSTR and the others.

BASICS OF PERL REGULAR EXPRESSIONS

Perl regular expressions are constructed using simple concepts like conditionals and loops. They are composed of characters and special characters called metacharacters. SAS searches a source string for a substring matching the specified Perl regular expressions. Using metacharacters enables SAS to perform special actions when searching for a match.

The following are a few basic features of Perl regular expressions:

Simple word matching

The simplest form of regular expression is a word, or more generally, a string of characters. A regular expression consisting of a word matches any string containing that word.

```
/world/
```

This would search for any string that contains the exact word 'world' anywhere inside it.

Using character classes

A character class allows a set of possible characters, rather than just a single character, to match at a particular point in a regular expression. Character classes are denoted by brackets [...] with the set of characters to be possibly matched inside.

`/[bcr]at/`

This would match 'bat', 'cat', and 'rat'. Only the characters listed inside the square brackets can match the single character in the pattern. Using character class, one can specify the possible values that the pattern will match in a particular position. This is an advantage over the typical wildcard search, which could only match characters.

There are several abbreviations for common character classes:

- `\d` matches a digit and represents [0-9]
- `\s` matches a whitespace character, including tab
- `\w` matches a word character (alphanumeric or `_`) and represents [0-9a-zA-Z_]
- `\D` is a negated `\d` and represents any character but a digit [`^0-9`]
- `\S` is a negated `\s` and represents any non-whitespace character [`^\s`]
- `\W` is a negated `\w` and represents any non-word character [`^\w`]
- The period `.` matches exactly one character.

Using alternation and grouping

The alternation metacharacter `|` allows a regular expression to match different possible words or character strings. This could be used to match a whole regular expression. But if one just wants to alternate part of a regular expression, grouping metacharacters `()` need to be added as well. Grouping allows parts of a regular expression to be treated as a single unit. Parts of a regular expression are grouped by enclosing them in parentheses. For example,

`/c(a|o)t/`

would match 'cat' and 'cot'.

Matching repetitions

The quantifier metacharacters `?`, `*`, `+`, and `{}` allow the determination of the number of repeats of a portion of a regular expression considered to be a match. Quantifiers are put immediately after the character, character class, or grouping to be specified.

Metacharacter	Behavior	Examples
<code>?</code>	Match 1 or 0 times	<code>/y(es)?/</code> matches 'y' or 'yes'
<code>*</code>	Match 0 or more time, i.e. any number of times	<code>/hat*/</code> matches 'hat', 'hats', 'ham' (as long as the first 2 characters matched, in this case 'ha')
<code>+</code>	Match 1 or more times	<code>/mat+/</code> matches 'mat', 'matt', 'mats'
<code>{n}</code>	Match exactly n times	<code>/\d{3}/</code> matches any 3-digit number and is equivalent to <code>/\d\d\d/</code>
<code>{n,}</code>	Match at least n times	<code>/\d{3,}/</code> matches any 3-digit or more number and is equivalent to <code>/\d\d\d+/</code>
<code>{n, m}</code>	Match at least n but not more than m times	<code>/\d{2,4}/</code> matches at least 2 digit number, but not more than 4 digit

Position matching

Perl also has another set of special characters `^`, `$`, `\b`, `\B` that do not match any character at all, but represent a particular place in a string. One major advantage of using regular expressions over other text matching functions is the ability to match text in specific locations of a string.

Metacharacter	Behavior	Examples
<code>^</code>	Match beginning of line, before the first character	<code>/^c/</code> matches 'cat' or 'cats' but not 'a cat'

\$	Match end of line, after the last character	/t\$/ matches 'hat' or 'a cat', but not the 'cats' or 'a cat and a dog'
\b	Match word boundary	/\b/ matches 'a cat' or 'a cat and a dog', but not 'cats'
\B	Match non-word boundary	/\B/ matches 'cats', but not 'cat' or 'a cat and a dog'

SYNTAX OF PERL REGULAR EXPRESSIONS

Creating regular expression in DATA step is a two-step process. First, the PRXPARSE function is used to create a regular expression. The regular expression id created by the PRXPARSE function is then used as an argument to other PRX functions. A good programming practice is to create regular expression only once by using the combination of if _N_=1 then a retain statement to retain the value returned by the PRXPARSE function.

One could also use PRXMATCH and PRXCHANGE with a Perl regular expression in a WHERE clause and in PROC SQL. There is no need to call PRXPARSE beforehand. This can be quite powerful in selecting and changing data that matches certain conditions. The disadvantage is that the perl regular expression used has to be assumed as well-formed, since no error checks are added to check whether the value returned by PRXPARSE function is missing.

The following two examples search for each observation in a data set for a 9-digit zipcode and output to the zipcode dataset. The two different approaches generate the same results. Only the first record John with zipcode 34567-2345 matched search criteria.

```
data zip;
    length name $20 zip $10;
    input name zip;
    datalines;
    John 34567-2345
    Smith 887701234
    Mary 56789
    ;
run;

data zipcode;
    set zip;
    if _N_=1 then do;
        retain re;
        re=prxparse('/\d{5}-\d{4}/');
        if missing(re) then do;
            put "Error: regular expression is malformed";
            stop;
        end;
    end;
    if prxmatch(re, zip);
    drop re;
run;

proc sql;
    create table zipcode as
    select name, zip from
    zip
    where prxmatch('/\d{5}-\d{4}/', zip);
quit;
```

APPLICATIONS OF PERL REGULAR EXPRESSIONS:

Example 1: Simple search

Suppose the medication 'Ambien' needs to be searched, but it is known that many misspellings exist in the file. The following example shows how to use regular expressions to find all records having different variations of 'Ambien'.

```
%* create regular expression only once;
retain pattern_num;
if _n_=1 then pattern_num=prxparse("/(a|e)mbi[ae](m|n)/i");
```

The above code first searches for letter 'a' or 'e', followed by letters 'mbi', then letter 'a' or 'e', and finally letter 'm' or 'n'. It will find the following different spellings: 'Ambien', 'ambian', 'ambiem', 'ambiam', 'embiem', 'embian', 'embien' and 'embian'. Option 'i' is used in this example to perform a case insensitive search. Without regular expression, each possible combination would have to be spelled out.

Example 2: Data Validation

To validate data, a pattern of characters within a string can be tested. Suppose some medicine names were given in free text format. One wishes to ensure that they contain product name, dosage and unit and are separated by a space, additionally only certain keywords are allowed in the units and the strings end with unit name. The sample data are like the following:

```
zomig 5 mg
Iron tabs
Tylenol 1000 mg
Advil 10000 mg
Motrin 2 caps
albuterol 2 puffs
ibuprofen 1600
Excedrin ES 3 tabs
Calcium 2 tabs daily
asprin81mg
multivitamin with iron 3 units
```

One could construct regular expression like the following to search for the medicine names meeting the criteria.

```
%* create regular expression only once;
retain pattern_num;
if _n_=1 then pattern_num=prxparse("/^\\D* \\d{1,4}
(tabs|mg|puffs|caps)$/");
```

The regular expression in this code searches for records that start with non-digits, followed by space, then followed by one to four digit number and white space. Finally the pattern ends with one of the four measurements: 'tabs', 'mg', 'puffs', and 'caps'.

To find the records that do not match the pattern, one could look for records where PRXMATCH return a zero.

```
%* use subsetting to get invalid records;
if prxmatch(pattern_num, trim(string))=0;
```

The following records are the non-matches:

```
Iron tabs
Advil 10000 mg
ibuprofen 1600
Calcium 2 tabs daily
asprin81mg
multivitamin with iron 3 units
```

Reasons for including the above records are as follows. The record 'Iron tabs' does not contain any digits. The record 'Advil 10000 mg' has 5 digits instead of 1-4 digits. The record 'ibuprofen 1600' does not have any units. The record 'Calcium 2 tabs daily' does not end with units. The record 'asprin81mg' does not have any space between medicine name and digits, or between digits and units. The record 'multivitamin with iron 3 units' does not end with correct measurements.

Example 3: Search and replace

A phrase such as "CONMED" could be described by many different ways, and they need to be replaced by consistent wording "concomitant medications". The sample text is like the following:

```
concom med
concomit medications
concomitant meds
concam medicine
```

One could create the following regular expression for use in PRXCHANGE function.

```
%* create regular expression only once;
retain pattern_num;
if _n_=1 then
    pattern_num=prxparse("s/conc[o|a]m(m)?(it|itant)?med(s|ici
ne|ications)?/concomitant medicine/");
```

The regular expression and the replacement string is specified in the PRXPARSE function, using the substitution operator "s" before the first '/'. Any time the pattern

```
/conc[o|a]m(m)?(it|itant)?med(s|icine|ications)?/
```

is found, it will be replaced with 'concomitant medicine'.

The variable pattern_num is then used as the first argument in the call routine PRXCHANGE.

```
infile 'c:\pattern match\med.txt' _infile_=line;
input;
newline=prxchange(pattern_num, -1, line);
```

The above code will replace the pattern at every occurrence since -1 is specified.

CONCLUSION

Perl regular expression provides many choices in pattern matching as shown in the previous examples. Perl regular expression functions can be easily used to search for patterns and replace strings in the text file. Any

SAS programmer who processes text data should consider adding Perl regular expressions to their programmer toolbox.

REFERENCES

Cody, Ronald (2004), "An Introduction to Perl Regular Expression in SAS 9", Proceedings of the 29th Annual SAS Users Group International

Pless, Richard F. (2005) "An Introduction to Regular Expression with Examples from Clinical Data"

TRADEMARKS

SAS® and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries

Other brand and product names are registered trademarks or trademarks of their respective companies.

CONTACT INFORMATION

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

Wenyu Hu
UG 1D-88
Merck Research Labs
Merck Co., & Inc.
Upper Gwynedd, PA 19454
(267) 305-6847
wenyu_hu@merck.com

Liping Zhang
UG 1CD-44
Merck Research Labs
Merck Co., & Inc
Upper Gwynedd, PA 19454
(267) 305-7980
liping_zhang@merck.com