PhUSE EU Connect 2018

Paper CT12

Increase Defensiveness of Your Code: Regular Expressions

Valeriia Oreshko, Covance, Kyiv, Ukraine
Daryna Khololovych, Intego Group, LLC, Kyiv, Ukraine

ABSTRACT

While dealing with unfixed textual formats, sometimes it is not evident how to create flexible code that will process data in the right way. In this paper there are some examples of Perl regular expressions usage for clinical data handling (e.g., checking when entries are not according to the rules, parsing data with unobvious delimiters). The PRXMATCH, PRXSUBSTR, PRXPOSN, PRXNEXT and PRXPAREN functions are considered. Additionally, some cases of regular expressions usage in SQL are discussed. This paper contains examples on small and big data, comparing different approaches to regular expressions, their effectiveness and speed- thus giving quantitative results to help understand which approach is better, and under which circumstances.

INTRODUCTION

A regular expression is a special text string for describing a search pattern. This paper is focused on the use of Perl regular expression in conjunction with the PRX family of functions/routines that are available in SAS® from version 9. Our goal is to provide some practical cases where regular expressions can be implemented and simplify the handling of clinical data. Instead of giving a theoretical overview of the Perl language tools, we provide detailed explanation of regular expression logic for each example.

REMOVING NON-PRINTABLE CHARACTERS

The first easiest example of using regular expressions is dealing with the conversion from one type of data or code to another. Suppose we have an Excel file, which after importing into a SAS dataset contains non-printable characters that should be replaced with a space. First, we create the pattern that defines the non-printable characters and changes them to a space symbol " ", added in the pattern after "/":

```
non_print_pattern=prxparse('s/[\b\e\f\n\r\t\v]+/ /');
```

The meaning of the metacharacters included in the described pattern is explained in the Table 1.

Metacharacter	Non-printable character
\b	Match word boundary
\e	Escape
\f	page break
\n	new line
\r	carriage return
\t	horizontal tabulation
\v	vertical tabulation

Table 1. *List of Perl metacharacter to match special symbols in file*

Then we use CALL PRXCHANGE to perform a "match and replace" procedure:

```
call prxchange (non_print_pattern, -1, variable_name);
```

We have here three required arguments:

- Regular expression id - the identification number returned by the PRXPARSE function for the regular expression to be used or a PERL regular expression;
- Times – setting to -1 means that replacement will be performed as many times as possible;
- Old string - the character expression in which to search and replace. If new-string is omitted then all changes will be made to old-string. If new-string is specified then old-string will remain unchanged.

DATE VALIDATION

Another area, where regular expression can be used, is date validation in particular – validation of dates reported in character YYYY-MM-DD format considering the possible number of days in each month and leap years. The idea of the regular expression is to review every symbol of the input date one by one and to build the resulting date that complies with the rules:

- date is from 20th or 21st century;
- date is in YYYY-MM-DD format;
- January, March, May, July, August, October and December have 31 days; April, June, September and November have 30 days; February has 28 or 29 days depending on if this is a leap year.

The following tools of the Perl language were used to design appropriate pattern:

- the metacharacters […], […-…] define the list or range of digits that can be present in the certain location;
- the modifier {n} controls a match of previous subexpression n times;
- the metacharacter | enclosed by parentheses () enables matching one of a set of alternatives.

The code for date validation and output (figure 1) are provided below:

```
data date_valid;
   set source;
   /*as it's DATE validation we can limit the length to 10*/
   if length(dat)<=10 then
   do;
   ex_date_valid = prxparse("/(((19|20)([2468][048]|[13579][26]|0[48])|2000)02-
29|((19|20)[0-9]{2}-(0[469]|11)-(0[1-9]|[12][0-9]|30)|(19|20)[0-9]{2}-
(0[13578]|1[02])-(0[1-9]|[12][0-9]|3[01])|(19|20)[0-9]{2}-02-(0[1-9]|1[0-
9]|2[0-8])))/o");
   end;
   valid_date = prxmatch (ex_date_valid, dat);
run;
```

dat	valid_date
2018-06-07	1
2005-04-31	0
2005-03-31	1
2018-06--7	0
1994-13-25	0
2016-02-29	1

Figure 1. *Date validation results*

Figure 2 shows the scheme of regular expression. Let's explore how it works for the input date "2018-05-30" (orange boxes in figure 1). The date starts with "20". Table 2 presents the correspondence between parts of "2018-05-30" date and subexpressions to which they were matched.

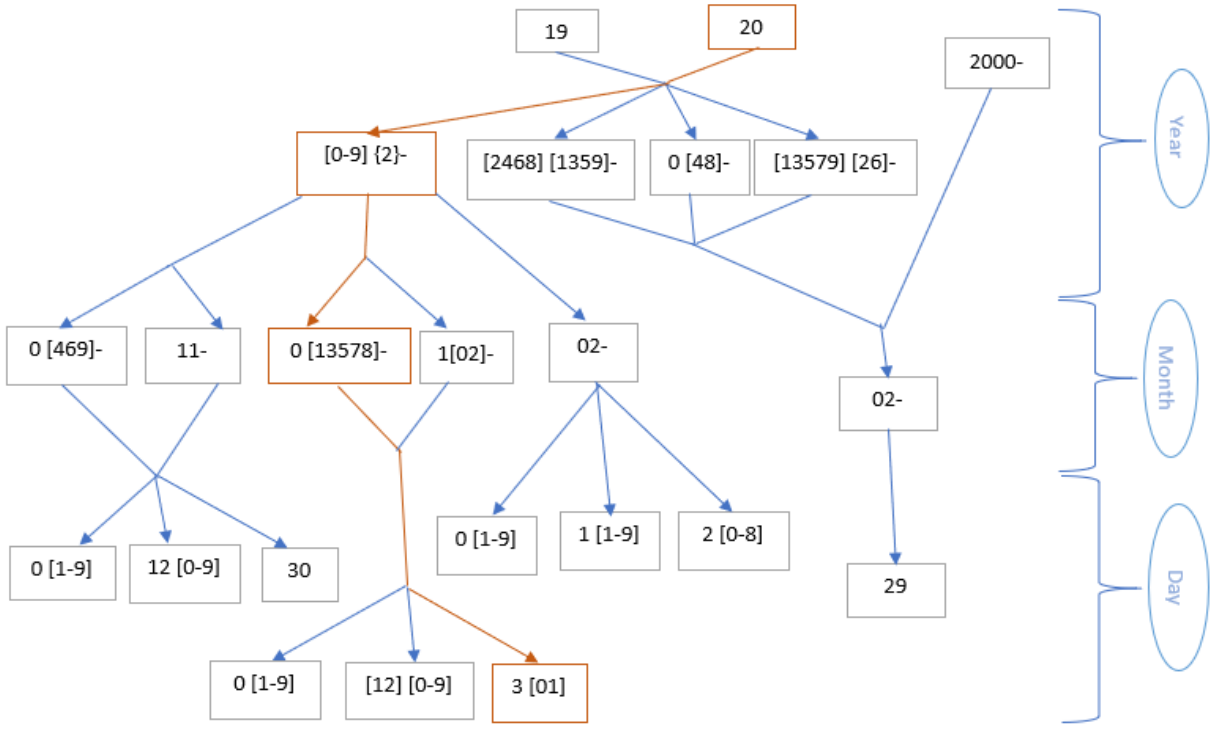

Figure 2. *Scheme of regular expression for date validation*

Date part	Regular expression part
20	20
18-	[0-9]{2}
05-	0[1359]
30	3[01]

Table 2. *Date parts and matching subexpression*

There are multiple paths of further development for the provided regular expression and its adoption according to specific needs: add time part, change/expand the list of delimiters between date parts, consider partial dates, etc.

DELETING DUPLICATES FROM THE STRING

The following example demonstrates how to get rid of repetitive parts of a string. In this case the word separated by a comma is considered as a separate 'part of the string'. Imagine we have the following dataset with adverse events listed, separated by commas (Figure 3 – column "aes").

Obs	aes	aes_final
1	Headache, nausea, headache, vomiting, nausea, fever	Headache, nausea, vomiting, fever
2	Fever, fever	Fever
3	Wheezing, fever, hypertension, wheezing	Wheezing, fever, hypertension
4	Wheezing, ulcer, ulcer, wheezing, headache, headache, ulcer	Wheezing, ulcer, headache

Figure 3. *Input and output variables for demonstrating of duplicates removing*

The following code removes all duplicate parts from the string (Picture 2 - column "aes_final"):

```
proc sql;
   *&n will be used as maximum possible repetition of one word in string;
   select max(countw(aes)) into: n
   from source;
run;
data dup_del;
    set source;
    aes_= strip(aes);
    *regexp to remove not first occurrence from string;
    same_word_subst=prxparse('s/(\b\w+\b)(.*?)(\b\1+\b)/\1\2/io');
```

```
    *regexp to detect not first occurrence from string;
    same_word_detect=prxparse('/(\b\w+\b)(.*?)(\b\1+\b)/io');
   *regexp to delete unwanted commas;
    spare_comma=prxparse('s/((\,)(\s*)(?=\,))|(\,\s*$)//o');
    do i=1 to &n;
        aes_=prxchange(same_word_subst, -1, compbl(aes_));
        if not prxmatch(same_word_detect, compbl(aes_)) then leave;
    end;
    aes_final = prxchange(spare_comma, -1, aes_);
run;
```

The regular expression to detect repeated words consists of three logical groups:

1) (\b\w+\b) – matches any number of characters delimited by word boundaries – used to memorize a word for further checking if the duplicate exists;
2) (.*?) – corresponds to any characters as many times as possible – was used in example to match substring between two duplicates;
3) (\b\1+\b) – backreference \1 catches the first group – to verify if the current word is a duplicate for the memorized in first group one.

Using the PRXCHANGE function, we display only the first and second capturing group, so the duplicated word (third group) will be excluded from final string. The global variable &n is a possible number of duplicates for the same word (e.g., 3 repetitions of "ulcer" in the fourth observation). It can't be greater than the maximum number of words in one observation of the source dataset. After executing the loop we get the interim result that contains spare commas where duplicates were deleted (figure 4):

Obs	aes	s
1	Headache, nausea, , vomiting, , fever	
2	Fever,	
3	Wheezing, fever, hypertension,	
4	Wheezing, ulcer, , , headache, ,	

Figure 4. *Interim result for duplicated removing*

To get rid of unneeded commas, we use the PRXCHANGE function and `spare_comma` regular expression. It matches and deletes commas at the end of string or commas followed by another comma:

1) `((\,)(\s*)(?=\,))` – with the help of positive lookahead we define if a comma is followed by another one, but don't include the second comma in the match.
2) `(\,\s*$)` – anchor $ signals that the comma is met at the end of a line.

CLASSIFICATION BASED ON SPECIFIC WORDS IN A STRING

Assume we have a dataset with the description Adverse events that can include the level of severity: mild, moderate and severe. The input data contains only `ae_descr` variable (see figure 5). The purpose is to get a variable that classifies severity from 1 to 3.

At first, we create a pattern that matches one of possible severity levels taking into account some cases of misspelling. The levels are listed in the regular expression in ascending order of severity. If the defined pattern is found in the input string, we call the function PRXPAREN that returns the value of the largest capture buffer that found a match `(matched_class` variable). As the levels were listed in regular expression in ascending order of its severity, the value of capture buffer is equal to its severity. The PRXPAREN function has only one input parameter - identifier of regular expression, so PRXPARSE and PRXMATCH should be called before PRXPAREN is used. The function PRXPOSN displays additional information about matched substring: position and length. The code described is represented below:

```
data class (drop = class_pat);
   set source;
   class_pat=prxparse('/(m[i*e*]ld)|(moderate*)|(sev[ie]re*)/io');
   if prxmatch(class_pat, ae_descr) then do;
       *class equals to number of group matched by regexp;
      matched_class=prxparen(class_pat);
      call prxposn(class_pat,  matched_class, position, length);
      matched_substr = substr(ae_descr, position, length);
   end;
run;
```

ae_descr	matched_class	position	length	matched_substr
sudden sever nausea	3	8	5	sever
MILD headache occured once	1	1	4	MILD
Moderat nausea	2	1	7	Moderat

Figure 5. *Output for classification ae severity.*

SPEED OF CODE EXECUTION

The defining of a regular expression takes time and memory. To compare the processing speed of code with and without regular expressions we performed the same task using just the SAS index function:

```
data class1;
   set source;
   if index(propcase(ae_descr), "Sever") ne 0 then position = 3;
   else if index(propcase(ae_descr), "Moder") ne 0 then position = 2;
   else if index(propcase(ae_descr), "Mild") ne 0 then position = 1;
run;
```

Analyzing the obtained results (table 3), we can conclude that code containing regular expressions executes twice as slow as code without regular expressions. But code with regular expression can handle different misspelling in class names, so it wins in functionality.

To speed up the code execution, it is common practice to initialize a regular expression on the first observation and retain generated pattern id:

```
If _n_ = 1 then do;
      Pattern = prxparse("/search/");
      Retain pattern;
End;
```

The “compile once” option can also be used to avoid the method above. The “o” following the closing slash in the end of regular expression tells SAS to compile it only once:

```
Pattern = prxparse("/search/o");
```

The speed of code execution with “compile once” option is also represented in table 3.

Number of observations	Regular expressions		Once compiled regular expression		Index function (no regular expressions)	
	CPU time	Real time	CPU time	Real time	CPU time	Real time
1 000	0.01	0.01	0.01	0.01	0.00	0.00
10 000	0.04	0.03	0.03	0.03	0.02	0.01
100 000	0.19	0.18	0.11	0.11	0.09	0.09
1 000 000	1.80	1.79	1.17	1.16	0.82	0.82
3 000 000	5.04	5.09	4.45	4.49	2.49	2.50

Table 3. *Comparison of the code execution speed*

EXTRACTING SPECIFIC PARTS FROM A STRING

The following sections of code demonstrate approaches for dividing string: applying the CALL PRXNEXT and CALL PRXSUBSTR routines for diving strings into several parts. As a delimiter, we used the Anatomical Therapeutic Chemical (“ATC”). For the testing purpose the following dataset has been created:

```
data base;
length var0 $200;
        var0="ATC|C10AA05";
        output;
        var0="J05AE05";
        output;
        var0="ATC|L|ATC|L04|ATC|L04A|ATC|L04AC|ATC|L04AC01";
        output;
run;
```

Following code shows how regular expressions can complement SQL calls. In given example variable `match` will contain a position of the first occurrence of the pattern in variable `var0` of the dataset `base`. In PRX functions used within PROC SQL, pattern should be indicated with keyword `calculated`.

```
proc sql;
   create table example1_sql as
   select *,
      prxparse("/atc\w*/io") as pattrn,
      /*matches "ATC" in combination with word”*/
       ifc(
         prxmatch(calculated pattrn, var0),
         prxposn(calculated pattrn, 0, var0),
         " "
       ) as match
   from base;
quit;
```

The next example defines the pattern using CALL PRXSUBSTR, and we detect length and position of a matched pattern:

```
data example2_prxsubstr;
       set base;
       retain var1 var5;
/* pattern for ATC|D|, 1 level, main anatomical group will be prxparse ("/ATC\|[A-
Z]\|/o"), or pattern for ATC|D11AB11, 5 level, chemical substance:*/
       var1=prxparse("/ATC\|[A-Z]\|/");
       var5=prxparse("/ATC\|[A-Z]\d\d\D\D\d\d/");
/* check if the variable contains the mentioned pattern if yes – we’ll get a starting
position and a length of it;*/
       call prxsubstr(var1,var0,st1,len1);
       call prxsubstr(var5,var0,st5,len5);
run;
```

The following piece of code demonstrates the usage of the PRXNEXT function for extracting the values between the defined delimiter:

```
proc sql;
   select max(countw(var0, "ATC")) into: n
   from base;
run;
data example3_prxnext;
  set base;
  atc_pttrn=prxparse("/(?<=atc\|)\w*/io");/* defines word after ATC delimiter*/
  start=1;
  call prxnext(atc_pttrn,start,length(var0),var0,position,length);
  array num $200 num1-num%eval(&n);
    do i = 1 to &n while (position > 0);
       num[i]= substr(var0, position, length);
             call prxnext(atc_pttrn,start,length(var0),var0,position,length);
    end;
run;
```

The result of dividing ATC Classification of drug into levels is provided in figure 6.

Obs	var0	num1	num2	num3	num4	num5
1	ATC|C10AA05	C10AA05				
2	J05AE05					
3	ATC|L|ATC|L04|ATC|L04A|ATC|L04AC|ATC|L04AC01	L	L04	L04A	L04AC	L04AC01

Figure 6. *Dividing of string into parts using “ATC” delimiter*

CHECKING LOG

This last example lets us check the log for unwanted notes, which are easily overlooked, but it’s good programming practice to get rid of them. First , we add everything that we don't want to see in our log file, so here we create patterns to detect it. As an example, it can be:

1) errors, warnings and some names which are undesirable;
2) checking for "kill";
3) some hardcoded values (e.g., “if usubjid=…”) ;
4) undesirable options like "noerrors";

Instead of `datalines` you can define checks with PROC IMPORT.

```
data list_of_notes;
       length notes $200;
       infile datalines truncover;
       input notes 1-200;
       datalines4;
error
warning
uninitialized
not found
lost card
went to a new line
invalid
is unknown
repeats of by values
remerging
at least one w.d. format
truncated to a length
have been converted
missing values
could not be performed
abnormally terminated
does not exist
could not be loaded
division by zero
has become more than 262
;;;;
run;
/* Adding all mentioned notes into a variable for using it in prxmatch, e.g.:
proh_notes=prxmatch('/error|warning|uninitialized|not found/i', content);*/
proc sql;
select notes into: notes separated by "|" from list_of_notes;
quit;

%macro _check_log(dirctry=,fileextension=,macvar_prohmsgs=);
/* get all names of program in the folder, is valid for UNIX, but has to be
adapted when program is used on other platforms */
   filename foldrnm pipe "ls -R '&dirctry.'" ;
   data dirfile;
      infile foldrnm truncover;
      input;
      length file $2000;
      file=_infile_;
      if index(file,"&type");
   run;
   proc sql noprint;
       select file into: _file separated by "@"
       from dirfile;
   quit;
   %let i=1;
       %do %until (%scan(&_file,&i,@)=);
           %let _name_var&i=%scan(&_file,&i,@);
           filename _log "&dirctry./&&_name_var&i";
           data step1;
           length content $2000;
              infile _log truncover;
              input;
              content=_infile_;
           run;
           data _log0_&i;
           length pathname $2000;
              set step1;
              pathname="&dirctry.&&_name_var&i";
       /* check for undesirable notes/warnings/errors ignoring the case;*/
              proh_notes=prxmatch("/&_notes/i",content);
              if proh_notes>0 then
```

```
                do;
                   put "INFO: log contains undesirable messages";
                end;
                if proh_notes>0;
                drop proh_notes;
            run;
        %let i=%eval(&i+1);
        %end;
        data _log_all_in_one;
              set _log0_:;
        run;
%mend _check_log;
%_check_log(dirctry=%str(path…/log/), fileextension=%str(.log), macvar_prohmsgs
=%str(&notes));
```

This method is very flexible and easily allows extensions to the list of undesirable phrases to the log search. In macro calling it is necessary to indicate the parameters path and type without quotation marks, and indicate in parameter _notes a string of unwanted log phrases, with phrases separated by pipe-symbols. This string may also be supplied via a macro variable, as is done here.

CONCLUSION

At first glance, regular expressions look like a set of strange symbols, which you “write once” and then forget about. That is why working with them requires patience and attentiveness. The disadvantage of regular expressions is that if the pattern is created incorrectly - you can miss something. That’s why it is highly recommended to check a pattern’s correctness (which could be done easily online – examples of tools are provided in references), and also to leave comments so your code will be understandable and easily readable to others.
The advantages of regular expressions are:

1. Once your pattern is known and clearly defined - your code works properly, and it doesn’t depend on the new loads of raw data;
2. Instead of many rows of defined conditions, you have one or two rows, so your code is much more concise;
3. The considerable benefit of pattern matching techniques is their ability to identify and handle data errors caused by human error while gathering information.

Even though we showed only several functions of regular expressions (PRXMATCH, PRXSUBSTR, PRXPOSN, PRXNEXT, PRXPAREN), it gives us a wide variety of techniques to handle data in a very defensive and accurate way. The rest depends on your imagination.

REFERENCES

1) Windham, K. Matthew. 2014. Introduction to Regular Expressions in SAS®. Cary, Nc: SAS Institute Inc**.**
2) Cassell, David L., “The Basics of the PRX Functions” SAS Global Forum 2007
   http://www2.sas.com/proceedings/forum2007/223-2007.pdf
3) Introduction to Regular Expressions in SAS (2014)
   http://apprize.info/programming/sas_1/4.html
4) Several online resources for checking how regular expressions work:
   https://regex101.com/
   https://regexr.com/
   http://txt2re.com/

CONTACT INFORMATION

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

Valeriia Oreshko
Covance
6 Oleny Telihy str.,
Kyiv / 04112
Email: valeriia.oreshko@chiltern.com

Daryna Khololovych
Intego Group, LLC
23 Baggoutovskaya street
Kyiv / 04107
Email: daryna.khololovych@intego-group.com