PAPER CC02

Optimized Lookup Using Regular Expressions

Jules van der Zalm, OCS Consulting, Rosmalen, the Netherlands

ABSTRACT

Regular expressions have been available in SAS® since version 6.12, but have become mature in SAS version 9 when SAS introduced Perl regular expressions. They provide a flexible and powerful way to look up and manipulate data that is (or should be) saved in a certain format, like patient numbers, barcodes or sample numbers, but also phone numbers or e-mail addresses. Next to data manipulation they allow for easy find and replace functionality within SAS code and log files when using a text editor that supports regular expressions, like UltraEdit. This makes it easier to replace certain pieces of code or to look up messages in the log file. This paper provides a summary of the available SAS functions, an explanation of the expressions and some practical uses in the pharmaceutical sector.

INTRODUCTION

Regular expressions are not a SAS-specific language. Many programming languages provide the option to use regular expressions to look up data or verify certain formatted fields on a form. Most of these programming languages have their own interpretation of the language that complies with the syntax of the programming language itself. So does SAS. SAS actually supports two different regular expression languages: its own, and the widely accepted standard Perl regular expressions.

This paper will give a short explanation of the functions that SAS provides for using Perl regular expressions, a simplified explanation of regular expressions and some practical uses in the pharmaceutical sector. SAS' own regular expressions language will not be discussed in this paper.

It is often said that code written using regular expressions is worn. Or actually: WORN, which is an abbreviation for Write Once, Read Never. Due to the nature of regular expressions this is, how disappointing that may sound, a bit true. For example: the following is a regular expression that may be used to 'recognize' a phone number like (555)123-4567: `/\(\d{3}\)\d{3}-\d{4}/`. Although you may be an inexperienced user and think that it is your lack of knowledge that keeps you from being able to read this relatively simple expression, please rest assured: even seasoned regular expression users find this difficult to interpret. Therefore it is very important to always thoroughly comment your regular expressions: split them up into pieces and explain what every part does.

There is definitely more to regular expression than what is explained in this paper. This paper is just a first step into this powerful technique. It explains the basics of writing regular expressions (and trying to read them) and introduces you into the basic regular expression functions that SAS offers. After reading this paper you will be able to write useful regular expressions, both in and outside SAS, and you'll have a basis that can get you on your way to learning the more advanced regular expression techniques.

BASIC EXPRESSIONS

To start understanding regular expressions, it is important to know the very basics. While explaining the basics no SAS syntax will be explained, nor will any SAS code be shown. Regular expressions are built up from metacharacters: characters that have a special meaning and function. Instead of explaining them all one-by-one, let's go through some examples to get to know them. Each example shows one or more regular expression(s), followed by an explanation telling what it will or will not match, and why. Important to know is that a regular expression always starts and ends with a delimiter, the forward slash ('/') in these examples.

```
/ache/
```

This matches any text containing the text 'ache'.

```
/^ache/ resp. /ache$/
```

This matches text starting with respectively ending with the text 'ache'. The caret matches the beginning of the input string; the dollar sign matches the end.

```
/h(ea|ae)dache/
```

This matches both 'headache' and 'haedache' – the latter of course being a misspell of the first. The pipe functions as an OR.

```
/h[iao]t/
```

This matches 'hit', 'hot' and 'hat', but not 'het' or 'hiat'. The characters within the square brackets form a set of which any character is a valid one to match.

```
/h[iao]{1,3}t/
```

This matches 'hit', 'hot', 'hiat' and 'hoait' but not 'heart'. The {1,3} refers to the preceding statement, which is [iao], and tells the interpreter that the preceding statement may occur no less than once and no more than three times.

```
/\(h(i|o)t\)/
```

This matches '(hit)' or '(hot)', including the round parenthesis. Because round parentheses have a special meaning in regular expressions but we do not want to use this special meaning in the example, they are 'escaped' by preceding them with backslashes. Any character that has a special meaning in regular expressions like question marks, dots, asterisks or any type of parenthesis must be escaped using a backslash to match them literally, even backslashes themselves.

So, until here we have learned that:

- A caret (`^`) represents the absolute beginning of a string, provided that it is used outside square brackets;
- A dollar-sign (`$`) represents the absolute ending of a string;
- A pipe (`|`) has an or-like meaning;
- Round parentheses are used to distinguish subparts of an expression;
- Items enclosed in square brackets – `[` and `]` – represent a character set that matches any of the enclosed characters;
- Numbers enclosed in curly brackets – `{` and `}` – tell how many time the preceding statement can be matched;
- Precede special characters with a backslash (`/`) to match them literally.

There are many special characters that can be used when composing regular expressions. The most important ones are `\d` (digits), `\D` (non-digits), `\b` (word boundary, the beginning or ending of a word), `\B` (non-word boundary), `\s` (any white space), `\w` (word characters, letters, numbers and underscores) and `\W` (the opposite of \w). These special characters can be used on any place within the regular expression. To confuse you more, most of them have an equivalent character set as defined in square brackets:

- \d equals [0-9]
- \D equals [^0-9] (where the caret says: all but …)
- \s equals [\f\n\r\t\v]
- \S equals [^\f\n\r\t\v]
- \w equals [a-zA-Z0-9]
- \W equals [^a-zA-Z0-9]

Another regular expression example:

```
/^\d{1,5}\s{1,}\w{1}$/
```

This is where it gets more difficult: this matches any text that has one to five digits (`\d{1,5}`), followed by at least one white space (`\s{1,}`) (being it a space, a tab, a new line or any other white space) and then followed by one word character (`\w{1}`). As said before, the numbers within the curly brackets tell how many times the preceding statement is allowed to be repeated. In three special cases these can be replaced by any of the three following special characters:

- * to replace {0,} meaning zero or more
- + to replace {1,} meaning one or more
- ? to replace {0,1} meaning zero or one

Knowing that, the previous example could also be written as `/^\d{1,5}\s+\w{1}$/`.

Let us now get back to the very first example that was given in the introduction. The exact expression was `/\(\d{3}\)\d{3}-\d{4}/`. Using what was learned so far we can decompose this expression as follows:

```
/         = expression begin
\(        = opening parenthesis  (
\d{3}     = three digits         555
\)        = closing parenthesis  )
\d{3}     = three digits         123
-         = dash                 -
\d{4}     = four digits          4567
/         = expression close
```

One more important basic: at any time you may add a lower-case ‘i' after the last delimiter to ignore upper- and lower case differences:

```
/headache/i
```

This matches both 'headache' and 'Headache'.

Now that you know the most important basics of regular expressions, let's look into the functions that SAS provides. The next few chapters will describe some of the functions to use regular expressions in SAS.

COMPILING YOUR REGULAR EXPRESSION

Before you can use a regular expression that you have written in your SAS code, it must be compiled. When compiling, which is done using the SAS function PRXPARSE, SAS interprets the regular expression and assigns to it a regular expression id. This id is actually just a number, starting from 1 for the first expression and incrementing by 1 for each next expression that you define. The correct syntax of the PRXPARSE function is `regex = prxparse("/ache/");`, used inside a data step. Despite what many examples tell there is no need to put this code inside an `if _n_ = 1 then do;` loop. When a constant is used as regular expression – a constant being the expression itself within quotes, as opposed to a variable containing the expression – SAS does not repeatedly compile the expression. Instead, it will return the id that belongs to that expression.

STRING MATCHING

After compiling the regular expression you can start using SAS regular expression functions that use the compiled expression. One of these functions is to check whether or not (a part of the) text in a variable matches your expression using the SAS function PRXMATCH. Assume the following: on a row in a dataset the variable TEXT contains the text 'headache'. The regular expression that is compiled into variable REGEX reads "`/ache/`". The PRXMATCH function is then called as `position = prxmatch(regex, text);`, after which the variable POSITION contains the position of the first match, which is 5.

```
data _null_;
  regex = prxparse("/ache/");
  text = "headache";
  position = prxmatch(regex, text);
  if position > 0 then put text= position=;
run;

_____________________
text=headache position=5
```

Instead of assigning the return value of the PRXMATCH function to a variable you can also use it in, for example, an IF statement: `if prxmatch(regex, text) then do;`.

RETURN A SUBSTRING

Who has never had the need to get just a part of a text? And who has never puzzled with STRPOS, LENGTH and SUBSTR functions to obtain the required part of that text?
Suppose you have visit texts defined in your visit mapping that are defined as G1V3 (where G1 stands for Group 1 and V3 for Visit 3), which may also read G1-3V12 (for blinded Groups 1 to 3, Visit 12) or G2P3 (Group 2, Phone Visit 3). In this case you can check whether the text contains either a V or a P to determine if it is a 'normal' or a phone visit, then substring the group number based on the location of either the V or the P in the visit text, after which you perform another substring to extract the visit number. Or you can use this regular expression: `/^G([0-9-]{1,3})(V|P)(\d{1,2})$/`.

```
^              Starting with ...
G              ... a G.
([0-9-]{1,3})  CB1: Then 1 to 3 characters, but only numbers 0 to 9 and dashes
(V|P)          CB2: Either a V or a P.
(\d{1,2})      CB3: 1 or 2 digits ...
$              ... at the end.
```

Very important in this regular expression are the parentheses. The parentheses are the key in identifying the different parts of the visit text. In SAS, each part of an expression that is enclosed in parenthesis is called a capture buffer. To read the actual content of those capture buffers you can use the SAS function PRXPOSN. The function requires three parameters: the regular expression id as returned by PRXPARSE, the capture buffer number (CB1, CB2 and CB3 in the code above) and the source text, which is the visit map in this example. It implicitly uses the result of, amongst others, PRXMATCH, so that the function must be called first to ensure that the text matches the given pattern.

```
data work.vistext;
  length group type visit $ 5;
  set work.vistext;

  regex = prxparse("/^G([0-9-]{1,3})(V|P)(\d{1,2})$/");

  if prxmatch(regex, strip(vistext)) then do;
    group = prxposn(regex, 1, vistext);
    type  = prxposn(regex, 2, vistext);
    visit = prxposn(regex, 3, vistext);
  end;

run;


_____________________________
vistext    group    type    visit

G1V1         1         V       1
G1-3V1       1-3       V       1
G3V12        3         V       12
G3P8         3         P       8
G1-3P12      1-3       P       12
G3P18        3         P       18
```

NOTE: Use 0 for the capture buffer number to have the PRXPOSN function return the entire match, which might be very useful.

PRXPOSN also exists as a CALL routine. The call routine requires three or four parameters: the regular expression id, the capture buffer, and one or two variable names. These variables are populated respectively with the starting position and the length of the capture buffer in the matched input. Subsequently the starting position and length can be used in a SUBSTR call to obtain the actual result. As the CALL routine, like the function, uses the result of PRXMATCH (and alike), this function must be called prior the using the CALL routine.

POSITION AND LENGTH OF A SUBSTRING MATCH

Regular expressions do not necessarily match a complete string. Quite the contrary: its strength is to get the matching part of a larger whole. The previous chapters demonstrated how PRXMATCH would return the position and also demonstrated the use of PRSPOSN as a CALL routine to obtain the position and length of a match using the 0 capture-buffer. For the latter you must first perform a PRXMATCH. Using the PRXSUBSTR CALL routine allows you to find this information without performing a match and thus with less code.

The CALL routine PRXSUBSTR has four arguments: the regular expression id, the source text string, the variable name that will be populated with the matching position and optionally the variable name that will be populated with the length of the match. If no match is found, both variables are populated with the value 0.

The example below describes how to find matches in lines of text – which could be investigator comments. It looks for the text 'headache', ignoring the case and ignoring misspelling of the 'ea' in 'headache'.

```
data work.testdata;
  infile datalines dsd;
  length line $ 50;
  input line $;
  datalines;
Subject suffered from headache and nausea.
Heeadache probably caused by treatment.
;;;;
run;

data work._null_;
  set work.testdata;

  regex = prxparse("/h([ea]+)dache/i");
  /*
  /h      An H ...
  ([ea]+) ... followed by one or more E's or A's, in no particular order  ...
  dache   ... followed by DACHE ...
  /i      ... but ignore the letter case.
  */

  /* This call routine populates POS and LEN with position and length. */
  call prxsubstr(regex, line, pos, len);
  /* This SUBSTR returns the matching part from LINE. */
  match = substr(line, pos, len);

  put "Match found: " match= pos=;

run;
_____________________________
Match found: match=headache pos=23
Match found: match=Heeadache pos=1
```

REPLACING TEXT

The last SAS function that will be discussed in this paper is the PRXCHANGE function. PRXCHANGE matches a string by the regular expression that is passed to it and returns a string that has the matching text replaced. This function could be used to correct spelling errors – to a certain extent, of course – by writing a regular expression that finds the misspelled notations of e.g. 'headache' (such as 'haedache', 'headahce' or 'hedache') and having the matches replaced by the correct notation.

Regular expressions that are used in PRXCHANGE are different from other regular expressions. First of all, they must start with a 's', before the first delimiter. The comes the regular expression and after the second (and normally the last) delimiter comes the replacing text, that is also followed by a delimiter. So to replace 'dog' with 'god' the expression would read `s/dog/god/`.

The following example uses PRXCHANGE to correct misspelled notations of 'headache'.

```
data work.comment;
  input line $ 50.;
  datalines;
Subject suffered from headache and nausea.
This heeadache was probably caused by treatment.
After two weeks no more hedahce.
;;;;
run;

data _null_;
  set work.comment;

  comment = prxchange("s/h([ea]+)da([ch]+)e/headache/i", -1, line);
  /*
  s/       Start the replace expression.
  h        First character is always an H ...
  ([ea]+)  ... followed by one or more A's or E's in any order.
  da       Then DA.
  ([ch]+)  Followed by one or more C's or H's in any order.
  e        And an E at the end.
  /
  headache Replace it with this exact text.
  /i       And don't care about letter case.
  */
  put comment=;

run;
_____________________________________
comment=Subject suffered from headache and nausea.
comment=This headache was probably caused by treatment.
comment=After two weeks no more headache.
```

As you can see in the code, the misspelled notations have been corrected and can now, for example, be recognized by AE coding applications.

The PRXCHANGE function lets you use the capture buffer that the regular expression contains. This is demonstrated by the following example. In this example there is a list of investigator names that is written in some different ways: <First Last>, <Last, First> or <Last,First>. By using a regular expression with capture buffers that only matches names that contain a comma, the position of the capture buffers in the result can be switched: **(\w+)**,\s?**(\w+)**. The two identical capture buffers, marked in bold, can be used in the replacement by referring to their id. The buffers are identified by an incremental number, where the first buffer has number 1. To switch positions of last and first names in names that contain a comma, the expression would read the following: `s/(\w+),\s?(\w+)/$2 $1/`.

```
s     Start of replace-expression.
/     First delimiter.
(\w+) One or more characters or numbers (last name) - buffer #1.
,     A comma.
\s?   Zero or one white space(s).
(\w+) One or more characters or numbers (first name) - buffer #2.
/     End of expression, start of replace.
$2 $1 Put buffer #2, a space and then buffer #1.
/     Closing delimiter.
```

Because names that are written as <First Last> do not have a comma in them, the position of the first and last name will not be switched because the text does not match our regular expression. The following and final example demonstrates the use of this regular expression.

```
data work.invest;
  input name $ 30.;
datalines;
Chester Bump
Soine,Tyler
Peterson, Clifford
Frank Wilcox
Gail, Thomas
;;;;
run;

data _null_;
  set work.invest;

  replace = prxchange("s/(\w+),\s?(\w+)/$2 $1/", -1, name);
  put replace=;

run;


_____________________________
replace=Chester Bump
replace=Tyler Soine
replace=Clifford Peterson
replace=Frank Wilcox
replace=Thomas Gail
```

OTHER SAS FUNCTIONS

The functions discussed in this paper are a selection of what the author of this paper thinks are the most useful and understandable SAS functions for Perl regular expressions. There are some more functions available that have not been discussed in this paper. These are explained on the SAS support website at http://support.sas.com/documentation/cdl_main/index.html.

CONCLUSION

Regular expressions are a powerful means of matching, finding and replacing strings of text or numbers. The techniques and functions described in this paper form a solid basis if you want to get a grip on regular expressions. Unless you are programming for job security it is very important to carefully describe the regular expressions in your SAS code, because regular expressions are difficult to read by nature. Once you've got regular expressions in SAS under control you are not limited to just SAS, because SAS supports the widely accepted Perl regular expressions.

REFERENCES

The content of this document is based on information taken from the SAS Support website, the SAS Help and Documentation of SAS 9.1.3 and from personal knowledge.

ACKNOWLEDGEMENTS

Thanks to my colleagues Bas van Bakel and Raymond Ebben for creative input and reviewing this paper, and to the PhUSE organization for giving me the opportunity to present my paper at PhUSE 2009 in Basel.

CONTACT INFORMATION

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

Author name	:	Jules van der Zalm
Company	:	OCS Consulting
Address	:	P.O. Box 490 5240 AL Rosmalen The Netherlands
Work phone	:	+31 (0)73 523 6000
Facsimile	:	+31 (0)73 523 6600
E-mail	:	sasquestions@ocs-consulting.com
Website	:	www.ocs-consulting.nl