nPaper TS03-2011

What to Do with a Regular Expression

Scott Davis, Experis, Portage, MI

Abstract

As many know, SAS has provided support for regular expressions for some time now. There are numerous papers that expose the basic concepts as well as some more advanced implementations of regular expressions. The intent of this paper is to bridge the gap between the very beginning and the advanced.

In the past you might have solved a programming problem with a combination of SUBSTR/SCAN and other functions. Now a regular expression may be used to greatly reduce the amount of code needed to accomplish the same task. Think of this paper as a recipe or guide book that can be referenced for some real-life examples that will hopefully get you thinking about ways to create your own regular expressions.

Introduction

Regular expressions are a powerful tool for programmers of all experience levels to use. As such, there are many different flavors of regular expressions out there though they mostly share a common thread. For SAS programmers there are two choices on the big board at the regular expressions deli. There are SAS regular expressions and Perl regular expressions. Although SAS regular expressions are still available for use, it appears that SAS has decided to throw its weight towards Perl regular expressions going forward. This is evidenced by the removal of SAS regular expressions from the online documentation.

Having a focus on Perl regular expressions is a good thing for SAS programmers as it allows us to focus on one standard instead of multiple ones. It also gives us something that transcends the SAS language that we can use in other applications.

One of the things I found challenging when I started to work with regular expressions was trying to figure out what to do with them. I was intrigued from the moment I was introduced to them, but I wasn't exactly sure how I might use them. All I knew was that I wanted to learn more and hoped I'd have a chance to use them sooner rather than later.

The result of some of my early forays into the world of regular expressions (or regex as they are often referred to) is what follows in this paper. The regex examples that you'll see are far from complex; they are more like the tip of the iceberg. As with many facets of programming, the examples only display one way to accomplish the particular task; there are many ways to reach the same end and there are many reasons for how we choose to get there. My hope is that these regex examples will help you figure out what to do with a regular expression.

I've Got a Date (or Two)

I'd be surprised if there isn't one among us who wasn't manipulating dates within the first week of learning SAS. SAS has given us a wide array of functions to manipulate full dates, partial dates, character -to-numeric conversions, etc. In the two examples below I was faced with something I hadn't encountered before with a character date manipulation. Both examples arose from the same task where SAS output was being compared to external data and the date fields were stored as character data.

Some of the code examples have been enhanced so that you can cut and paste and see how it works. The original code did not have the various assignment statements as the data were being read from different data sets. You'll also notice that, when using the various 'PRX' functions, I interchange using a previously defined regular expression id with a complete inline regex. This is done to demonstrate that the functions can be used either way. For more information you can consult the SAS documentation for the 'PRX' functions.

Removing '20' from '2011'

In the first case, the data had mixed entries for the year portion of a date value stored in a character variable. Some fields might have 15-Jan-11 while others were 15-Jan-2011 00:00 (the source system needlessly added the 'blank' time). When the line by line comparison was done, these were obviously tagged as mismatches. I decided my best bet was to construct a regex that would check the character date field for the year format and make any changes, if applicable.

Here is the code related to the manipulation of the yyyy portion of the date field:

```
data temp;
   comparedt='25-JAN-2010 00:00';
   newcomparedt=scan(stdt,1,'');
   sourcedt='25-JAN-10';
   pos=prxmatch('/-\d\d\d\d/', newcomparedt);
   if pos then do;
      newyr=substr(newcomparedt,pos+3,2);
      comparedt1=substr(newcomparedt,1,pos)||newyr;
   end;
run;
```

Running the code above gives you gives you six variables in the data set 'temp'. The first variable, *comparedt*, has the date field with the yyyy format along with the erroneous time. The second variable , *newcomparedt*, strips off the time portion that we are not concerned with. The third variable, *sourcedt*, has the format we are comparing to. The fourth variable , *pos*, returns the position in *newcomparedt* where the match, if any, is found. The fifth variable , *newyr*, contains our two-digit year. Finally, the sixth variable, *comparedt1*, is the yyyy formatted date newly formatted using the regex.

Through the use of the regex, `'/-\d\d\d\d/'`, we are asking SAS to locate four digits that follow a dash in specified variable. When the PRXMATCH function finds a match we can use the variable *pos* to do some further manipulation to get the desired resulting variable.

Here is the final result:

comparedt	newcomparedt	sourcedt	pos	newyr	comparedt1
25-JAN-2010 00:00	25-JAN-2010	25-JAN-10	7	10	25-JAN-10

Now the original *sourcedt* variable can easily be compared with *comparedt1*.

Zapping Out Zero-Filled Dates

In the same file there was a date of birth field that contained two similar date formats. While they both appeared to be in the MMDDYY10 format, there were some that did not have zero-filled dates for those months less than October and days less than 10. Here you might see 08/01/11 or 8/1/11.

The code example is going to take a fully-formatted character date field as MMDDYY10. There are two regexes here to keep it clear what each is doing.

```
data temp;
   dob='01/01/2011';
   newdob1=prxchange('s/^0(\d)/$1/', -1, dob);
   newdob2=prxchange('s/(\d+)\/0(\d)/$1\/$2/', -1, newdob1);
run;
```

The first regex will check the month (mm) portion of the date by searching for '0' followed by a digit at the beginning of the string (that's what the carat (^) means). The PRXCHANGE function will replace '01' with '1'. The $1 in the 2nd part of the regex is utilizing the capture buffer which is created by using the parentheses around the digit metacharacter(\d). The -1 in the function tells SAS to keep searching and replacing until the end of the source. It doesn't really apply in either case here, but is a necessary argument for the function.

The second regex is very similar to the first. It will keep the month (mm) portion of the date and strip off a zero of the day (dd) portion if it exists.

Here is the result:

dob	newdob1	newdob2
01/01/2011	1/01/2011	1/1/2011

Case in Point

Older than time itself, it seems, is the problem of the 'wrong' case. I was somewhat embarrassed to have released code that had this problem in the first place, but, it happens (job security, right?). I had written a regex that looked at a file path that ended with a SAS program and extracted the program name.

Here is how the original code appeared (within a larger macro):

```
%macro temp;
   %let pgmfolder_sysin =
c:\verylong\superlong\stillcantbelieve\howlong\thispath_is\some-program.sas;

   %global temp_pgmname;

   data _null_;
      retain re1;
      length pgmname $50;
      if _n_ = 1 then do;
         re1=prxparse('/\\([^\\\.]+)\.sas/');
      end;
      if prxmatch(re1,"&pgmfolder_sysin") then do;
         pgmname=prxposn(re1,1,"&pgmfolder_sysin");
         call symput('temp_pgmname',pgmname);
      end;
      put pgmname=;
      %put _user_;
   run;
%mend temp;
%temp;
```

This worked beautifully while testing since all my programs ended in lower case .sas. You can see in the log how the variables get set as expected:

```
GLOBAL PGMFOLDER_SYSIN c:\verylong\superlong\stillcantbelieve\howlong\thispath_is\some-program.sas
GLOBAL TEMP_PGMNAME some-program

pgmname=some-program
```

But, alas, when some-program.sas became SOME-PROGRAM.SAS the code bombed. Luckily, there are a number of modifiers available, one of which ignores case. Adding the modifier **i**, for ignore case, made my regex code happy again.

```
re1=prxparse('/\\([^\\\.]+)\.sas/i');
```

Fun with Special Characters

One day I was asked to assist with a problem that had a special character, a tilde (~), in a string that signified a line break with white space that followed that was to be used as indentation on the next line. The output would be viewed as html. The trick was that not all non-white spaces should be replaced with the html literal, , only those that followed the tilde leading up to the next word or phrase.

Suppose our variable is this: string=This is line 1~ Here is indented line 2;

In the final output we want it to appear as:

This is line 1

  Here is indented line 2

The code:

```
data _null_;
  if _n_=1 then do;
     re1=prxparse('/~([ ]+)(.+)/');
  end;
  length string $200;
  string= 'This is line 1~ Here is indented line 2';
  position=prxmatch(re1,string);
  text=prxposn(re1,2,string);
  call prxposn(re1, 1, pos, length);
  do i=1 to length;
     if i=1 then newpos=pos;
     else if i>1 then newpos=newpos+6;
     substr(string,newpos)=' ';
  end;
  newstring=trim(string)||left(text);
  put newstring=;
run;
```

Our regex identifies the number of spaces after the tilde, ~([ ]+), and before the next text section, (.+), and puts them both into capture buffers, 1 and 2 respectively.

We get the remaining text from the 2nd capture buffer and place it into the variable text using the PRXPOSN function. Then, in a variation on a theme, we use the CALL PRXPOSN function to get the number of spaces between the tilde and the remaining text from the 1st capture buffer. Our html literal, ** **, is 6 characters long. Every time we replace a space with the literal we need to add 6 to the line position counter.
The PUT statement from the log appears like this:

```
newstring=This is line 1~ Here is indented line 2
```

The code will insert the html literal for as many spaces as there are after the tilde and before the remaining text. For example, three spaces after the tilde would produce this:

```
newstring=This is line 1~   Here is indented line 2
```

Do You Know The Extension You Are Trying to Parse?

Suppose you have a need to find out what the file extension is for an input file. You know that it will typically be 3 or 4 characters, such as **rtf** or **html**, preceded by a period, but you don't know if your filename or path might have periods elsewhere so a simple SCAN function won't cut it.

This example demonstrates another way that a regex can be implemented within macro code.

```
%macro temp;
   %let input_file = something.rtf;
   %let regex = %sysfunc(prxparse(/\.(\w+)$/));
   %let prxm = %sysfunc(prxmatch(&regex,&input_file));
   %if &prxm %then %do;
      %let chkExt = %sysfunc(prxposn(&regex,1,&input_file));
   %end;

   %put;
   %put input_file = &input_file;
   %put chkExt = &chkExt;
%mend temp;
%temp;
```

The regex begins with a period (.) and ends with a dollar sign ($). You'll notice that the period is 'escaped' with a \. This is because a period by itself is a regex metacharacter that means match any character. In this case we really want to match the period that is right before our file extension so we 'escape it' with a \. The $ is a metacharacter telling us to look/match from the end of the string. In between we have a capture buffer that uses the \w metacharacter along with the + metacharacter that will give us the actual extension.

The resulting log looks like this:

```
input_file = something.rtf
chkExt = rtf
```

If our input file was actually some.thing.html you'd see that the code still functions as expected:

```
input_file = some.thing.html
chkExt = html
```

And Now for a Change of Pace

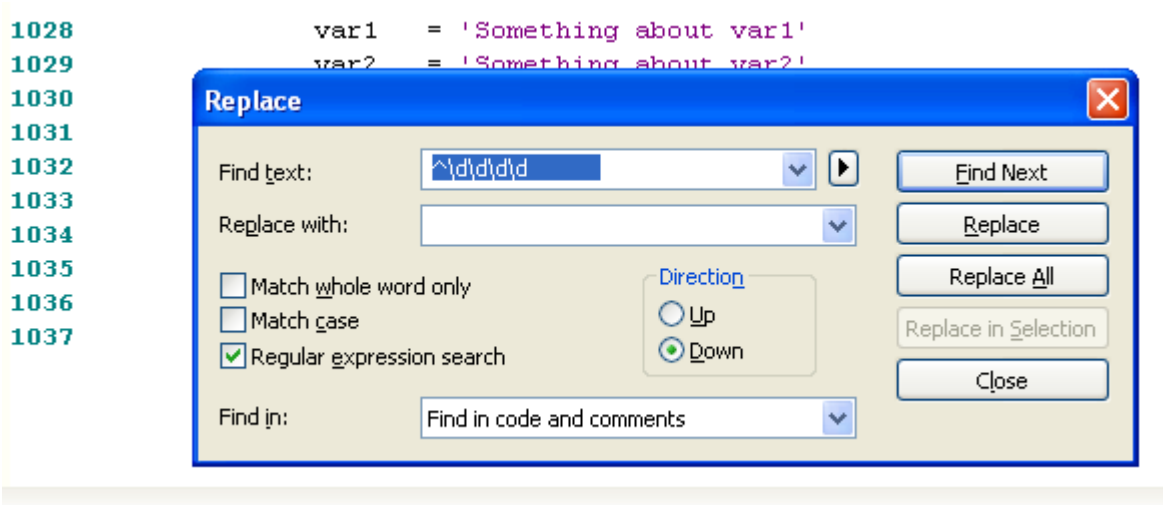
The last example is a slight departure since it still is about regexes, but it does not involve running SAS code. In this case I had some code displayed in a log that I wanted to use in a program, but I didn't want to retype it.

Imagine a log that looks like this:

```
1024
1025      data temp1;
1026         set temp;
1027         label
1028               var1    = 'Something about var1'
1029               var2    = 'Something about var2'
1030               var3    = 'Something about var3'
1031               var4    = 'Something about var4'
1032               var5    = 'Something about var5'
1033               var6    = 'Something about var6'
1034               var7    = 'Something about var7'
1035               var8    = 'Something about var8'
1036               var9    = 'Something about var9'
1037               var10   = 'Something about var10'
1038         ;
1039      run;
```

When the code is copied and pasted into the editor it can be a pain to remove each line number along with the intervening blanks.

Once the text from the log is pasted into the editor, hit Ctrl-H and you'll notice that the 3rd option down on the lower left side is 'Regular expression search'. To get rid of the line numbers and blanks I enter the regex '^\d\d\d\d ' into the 'Find text' field in the pop-up window as shown below:



Notice the extra spaces highlighted following the metacharacter portion of the regex. I did this to follow good programming practices and indent a few spaces.

Here's the code in the editor after the replace:

```
26
27     var1    = 'Something about var1'
28     var2    = 'Something about var2'
29     var3    = 'Something about var3'
30     var4    = 'Something about var4'
31     var5    = 'Something about var5'
32     var6    = 'Something about var6'
33     var7    = 'Something about var7'
34     var8    = 'Something about var8'
35     var9    = 'Something about var9'
36     var10   = 'Something about var10'
37
```

Conclusion

There are many applications of regular expressions some of which can be quite complicated at first to code and also to decipher when looking at someone else's code. The examples here are purposefully not those kind of regular expressions. The code you see in the paper is very simple and copy and paste ready.

Hopefully, you can try out these examples to see how the different regex functions work and begin to think of ways you can apply them in your own work. It may seem daunting at first, but it can actually be a lot of fun. Imagine that, having fun at work!

Acknowledgements

I would like to thank Rosie Grzebyk for her support in reviewing this paper.

Recommended Reading

SAS Perl Regular Expressions Tip Sheet
http://support.sas.com/rnd/base/datastep/perl_regexp/regexp-tip-sheet.pdf

SAS 9.2 Language Reference: Dictionary – Functions and Call Routines –
Using Perl Regular Expressions in the DATA Step
All PRXnnn and CALL PRXnnn functions

Mastering Regular Expressions – Jeffrey E.F. Friedl

Contact Information

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

Scott Davis
Experis
5220 Lovers Lane, Suite 200
Portage, MI 49002
Phone: 269-553-5122
E-mail: scott.davis@experis.com