Paper 046-29

Double Your Pleasure, Double Your Words

David L. Cassell, Design Pathways, Corvallis, OR

ABSTRACT

Now that SAS® 9 has Perl regular expressions in the DATA step, there are new capabilities in it. A common editing problem, finding doubled words in a text, can be tackled using Perl regular expressions. But the structure of the DATA step when using the PRX... functions and CALL routines can be simplified by using the Whitlock DO-loop in conjunction with the PRX functions. We illustrate this, showing how to build a simple Perl regular expression, and how to use the Whitlock DO-loop with the PRX... functions to find doubled words in lines of text.

INTRODUCTION

The traditional form of the DATA step uses an implicit loop to read in lines from a file for processing. However, this can be changed, to make the DATA step look more like code from a traditional computer language. Instead of writing:

```
data mynewdata;
  set myolddata end=eof;

  if _n_=1 then do;
    /* initializing lines of code */
    end;

  /* the body of the DATA step here */

  if eof then do;
    /* lines of code to clean up afterward */
    end;
  run;
```

We can opt for the style which has become known on SAS-L as the Whitlock DO-loop, or the DOW-loop. This structure provides the same utility as the traditional form of the DATA step, but it explicitly puts the file-reading features within a do until() loop, like this:

```
data mynewdata;
  /* initializing lines of code */

  do until(eof);
    set myolddata end=eof;
    /* the body of the DATA step here */
    end;

  /* lines of code to clean up afterward */
  run;
```

This is but one form of the Whitlock DO-loop: other versions use other forms of the clause inside the do until() statement. We will see the utility of the above form in just one moment...

THE PRX... FUNCTIONS AND CALL ROUTINES

As of SAS 9, many of the Perl regular expression features of Perl version 5.6.1 are available within the DATA step. A typical piece of code using these regular expression - or 'regex', as the cognoscenti say - features might look like this:

```
data _null_;
   set MyData;

   if _n_ = 1 then do;
      retain re;
      re = prxparse('/Perl pattern match/');
      end;

   if prxmatch(re,MyVariable) then do;
      /* more lines of code */
      end;
      run;
```

As an aside, let us point out that we have deliberately left the regex-checking code out of this paper. That forces us to assume that the regex is well-formed. If the regex in the PRXPARSE() function is not well-formed, then the value of RE will be missing, and any subsequent code using the regular expression will fail to work properly. Adding the checking code will make the program longer, and will make the Whitlock DO-loop look even more desirable.

We can use the PRX functions and/or CALL routines in conjunction with the Whitlock DO-loop, to make our code look a little cleaner. So let's look first at our problem...

THE DOUBLED WORD AND THE EDITOR

A common problem in editing text is finding occurrence of doubled words: phrases such as "you you" in text. We would need to be able to find words which differ in capitalization, as in "You you...". We would need to find words which might have one space in between them, or several spaces, or even a tab. We would need to make sure that we do not accidentally match parts of words, like "...the theropod..." which has the seven-character string "the the" contained inside it, or "...blithe the...", which has the same feature.

This can be done with a SAS macro that would scan each line, parse out the words one by one, match them, and use only uppercase for the comparisons. However, that might be a few lines of less-than-obvious code. We can achieve the same goal using the PRX... functions and the Whitlock DO-loop, in a fairly clear way. Let's begin with eight lines of simple example text. (We will refrain from numbering the lines, to make sure the reader knows the numbers are not part of the lines themselves.)

```
Around the rugged rock the
The ragged rascal ran.
Around the rugged rock the
ragged rascal ran.  Around
the the rugged rock the
ragged rascal ran.  Around
the rugged rock rock the
ragged rascal ran ran.
```

It is clear from inspection that there are doubled words on the fifth, seventh, and eighth lines. However, in the real world, we would have 80,000 lines to examine, or 80,000,000 lines, not just 8. So visual checks are simply not enough. And we can’t forget that visual checks fail some of the time: did you note that there is a doubled word spanning lines 1 and 2 ? Let's use the Whitlock DO-loop and the new functions PRXPARSE() and PRXMATCH() to address the problem.

First, what do we need our regex to do? It must match as follows:

[1] start with the beginning of a word,
[2] match the entire word,
[3] skip over an arbitrary amount of whitespace,
[4] match the second word as being identical to the first word, up to capitalization,
[5] and make sure that the entire second word is used in the match.

So let's build a Perl regular expression which does exactly that. The special character '\b' matches the border between a 'word' and a 'non-word'. This is a subtle issue. \b doesn't match the first letter of a word, or the last character before the word. It matches the zero-width dividing line between the word and the space (or other non-word character, like a tab or a line-drawing character). So we start our regex with a '\b'.

Next, we want to capture an entire word. The special character \w matches any 'word' character, including digits and underscores. The iterator '+' will match one or more of whatever is immediately before it. '\w+' will therefore match one or more consecutive word characters, i.e., a word. Note in passing that the numbers and underscore mean that \w will match any legal character in a V6 SAS data set name, which is convenient for SAS programmers.

Then we want to match some whitespace. '\s' will match one whitespace character: a space, a tab, a linefeed, etc. So '\s+' will match one or more consecutive whitespace characters.

Next, we want to match the previous word. If we capture that word somehow, we might be able to check against it. Perl provides the ability to capture a match using parentheses. Once we have captured that match, we can refer to it in our regex using '\1' (or, if we have multiple matches, then a backslash followed by the correct number of the match).

Then we need to make sure that we include all the second word as well, so another '\b' at the end of the second word will make sure that we match against the whole word and not the first portion of the word.

Finally, we need to ignore differences in capitalization. Adding the option 'i' at the end of the regex after the closing slash will automagically give us this feature. So the final regex looks like:

```
       /\b(\w+)\s+\1\b/i
```

THE WHITLOCK DO-LOOP IN PRACTICE

Now let's put that into a DATA step. Instead of having to worry about that _n_=1 stuff, and RETAINing the output of the PRXPARSE() function, we can make our code look a little nicer.

```
data _null_;
  re = prxparse('/\b(\w+)\s+\1\b/i');

  do until(eof);
    set MyData end=eof;
if prxmatch(re, line) then put line= ;
end;
  run;
```

And there we have it. The RETAIN is done implicitly, since the DATA step is not doing its usual iteration. RE is built at the beginning of the DATA step, before we start running through the lines of the file. And we get the following output in our log:

```
line=the the rugged rock the
line=the rugged rock rock the
line=ragged rascal ran ran.
```

But we still haven't solved all of our problem. This does everything except read across lines. And a quick look at our original text tells us that there is such a case, which we missed. So we need to remember the previous line's words as well. This is simple enough to fix, using LAG() to grab the previous line. We build a longer variable consisting of the previous and current lines, and execute the very same regex comparison on our longer line.

```
data _null_;
  re = prxparse('/\b(\w+)\s+\1\b/i');

  do until(eof);
    set temp1 end=eof;
longline = trim(lag(line)) || ' ' || line;
if prxmatch(re, longline) then put longline= ;
end;
  run;
```

Now we get the desired output in the log:

```
   longline=Around the rugged rock the The ragged rascal ran.
   longline=ragged rascal ran. the the rugged rock the
   longline=the the rugged rock the ragged rascal ran.
   longline=ragged rascal ran. the rugged rock rock the
   longline=the rugged rock rock the ragged rascal ran ran.
```

Note that some of the doubled words appear in more than one value of LONGLINE, and hence are written more than once.

As a secondary note, Perl has better ways of dealing with matching across lines, but the PRX functions in SAS cannot do everything that Perl 5.6.1 regular expressions can do. The SAS 9 documentation spells out these limitations in great detail.

CONCLUSION

The simplicity of the Whitlock DO-loop combines well with the functionality of the new PRX... functions and CALL routines to add even more capability to the SAS DATA step. Now all we have to do is learn all the ins nad outs of Perl regular expression syntax...

ACKNOWLEDGMENTS

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. ® indicates USA registration.

The Perl language was designed by Larry Wall, and is available for public use under both the GNU GPL and the Perl Copyleft.

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

CONTACT INFORMATION

Your comments and questions are valued and encouraged. The author may be contacted by mail at:

David L. Cassell
Design Pathways
3115 NW Norwood Pl.
Corvallis, OR 97330

or by e-mail at:
Cassell.David@epamail.epa.gov