

Paper 135-2007

Perl Regular Expressions 102

Kenneth W. Borowiak, PPD, Inc., Morrisville, NC

ABSTRACT

Perl regular expressions were made available in SAS® in Version 9 through the PRX family of functions and call routines. The SAS community has already generated some literature on getting started with these often-cryptic, but very powerful character functions. The goal of this paper is to build upon the existing literature by exposing some of the pitfalls and subtleties of writing regular expressions. Using a fictitious clinical trial adverse event data set, concepts such as zero-width assertions, anchors, non-capturing buffers and greedy quantifiers are explored. This paper is targeted at those who already have a basic knowledge of regular expressions.

Keywords: word boundary, negative lookaheads, positive lookbehinds, zero-width assertions, anchors, non-capturing buffers, greedy quantifiers

INTRODUCTION

Regular expressions enable you to generally characterize a pattern for subsequent matching and manipulation of text fields. If you have ever used a text editor's Find (-and Replace) capability of literal strings then you are already using regular expressions, albeit in the most strict sense. In SAS Version 9, Perl regular expressions were made available through the PRX family of functions and call routines. Though the Programming Extract and Reporting Language is itself a programming language, it is the regular expression capabilities of Perl that have been implemented in SAS.

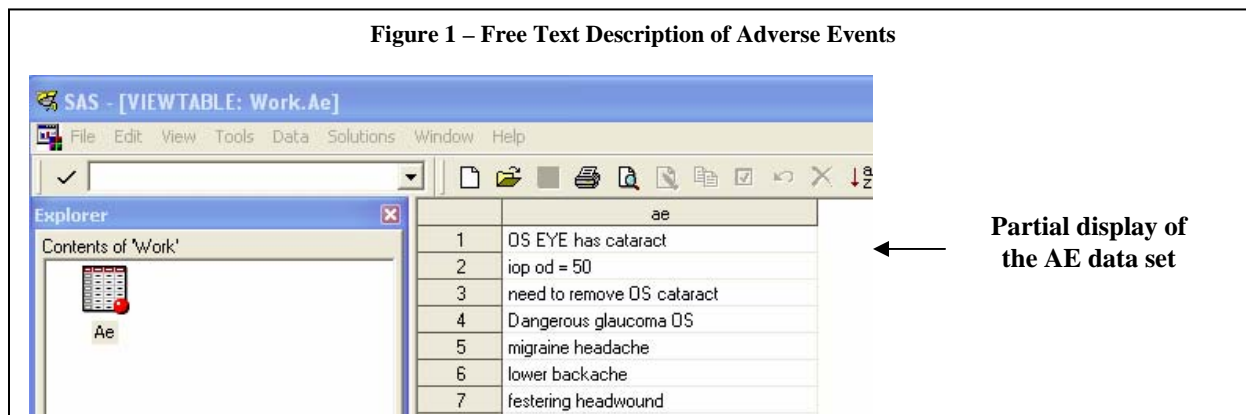
The SAS community of users and developers has already generated some literature on getting started with these often-cryptic, but very powerful character functions. Introductory papers by Cassell [2005], Cody [2006], Pless [2005] and others are referenced at the end of this paper. The goal of this paper is to build upon the existing literature by exposing some of the pitfalls and subtleties of writing and using regular expressions (aka regexen). This paper is targeted at those who already have a basic knowledge of regular expressions. If that is not you, then see the papers previously mentioned and (please) return here. Do not be discouraged if you have to read those papers two, five, or even twenty times before you start feeling comfortable with regular expressions.

In this paper, regular expression concepts such as zero-width assertions, anchors, non-capturing buffers and greedy quantifiers are explored using a fictitious clinical trial data set.

PRELIMARIES

Figure 1 below contains a partial display of an adverse event data set from a fictitious ocular (i.e. eye related) clinical trial. For those unfamiliar with clinical trials, an adverse event is any unfavorable or unintended sign temporarily associated with a medicinal product. It is not uncommon to capture a free-text description of the event before it is mapped using a standard classification system. The program that generated the entire data set can be found in the Appendix at the end of this paper.

Figure 1 – Free Text Description of Adverse Events



	ae
1	OS EYE has cataract
2	iop od = 50
3	need to remove OS cataract
4	Dangerous glaucoma OS
5	migraine headache
6	lower backache
7	festering headwound

WORD BOUNDARY (\b)

The *word boundary* \b is one of the features of regular expressions that make them more flexible than traditional SAS text string functions. This metacharacter is the boundary between a word character (\w¹) and a non-word character. Not only could the non-word character be \W (which is the complement of \w), but also the beginning or the end of the variable. Here is one tempting misconception to avoid: *\b is not the boundary between non-whitespace and whitespace.*

Consider the task where we would like to flag observations from the data set AE that describe ocular adverse events using a regular expression. Specifically, ocular adverse events are the observations where the string OD, OS, or OU is found, which represent the right, left, and both eyes, respectively². We would like to flag the observation 'Dangerous glaucoma OS' for the OS and not for the ou part of 'Dangerous'. In addition, suppose the adverse events were recorded by a clinician with messy handwriting and were entered in a free text field by a data entry clerk fixating on her next cigarette break, so the data is not very clean. Therefore, a few additional requirements need to be incorporated into our regular expression. Characters in the strings of interest could be separated by a period (.) but nothing else; they may be in upper or lower case, and letter 'O' could be replaced by the number '0'. In Figure 2A below you can find a DATA step containing a regular expression to determine whether the variable AE contains the string indicating an ocular adverse event.

Figure 2A - Use of the Word Boundary \b

```
/* Flag observations indicating an ocular adverse event */
data Find_Ocular_AE(drop=:);
  set AE;

  retain _re;
  if _n_=1 then do;
    _re=prxparse('/\b[o0]\.?[uds]\.?/i'); /* Compile the regex once at the */
    if missing(_re) then do;              /* first iteration of DATA step */
      putlog 'ERROR: Regex _re is malformed';
      stop;
    end;
  end;

  match_pos= prxmatch(_re,ae) ; /* Position where the match begins */
  OcularYN=(match_pos > 0);    /* Boolean indicating a match */
run;
```

Comments on the Regex

/\b[o0]\.?[uds]\.?/i

Options: case insensitive

- Assert position at a word boundary
- Match a single character present in the list "o0"
- Match the character "." literally
 - Between zero and one times, as many times as possible, giving back as needed (greedy)
- Match a single character present in the list "uds"
- Match the character "." literally
 - Between zero and one times, as many times as possible, giving back as needed (greedy)

¹ To refresh your memory, \w represents the character class [a-zA-Z0-9_].

² These are abbreviations from their Latin translation, *ocular dexter*, *ocular sinister*, and *ocular unitas*.

Before describing the regular expression needed for the task, some general comments about the way the PRX functions are used in this example used are in order. In the code in Figure 2A, the regular expression is compiled only once at the first iteration of the DATA step using the PRXPARSE function. The value assigned to the variable `_re` by the compilation of the regular expression is retained and available for future use as an argument to the PRXMATCH function. Checking for errors in the regular expression is also done at this juncture. If the regular expression is not syntactically correct, a missing value will be assigned to variable `_re`. If this occurs, a polite reminder is written to the SAS log and the DATA step will then terminate. The compile once behavior can also be accomplished using the `/o` modifier under most conditions. Future examples will have the regex entered directly in the PRXMATCH function located in the WHERE clause of PROC SQL, where one-time compilation of the regex occurs automatically. Generally speaking, one-time compilation of the regex occurs when used in WHERE clauses of any PROC or DATA step. However, you lose the ability to do any error checking on the regex.

Comments on the regex defined in the PRXPARSE function are included in Figure 2A. The variable `match_pos` contains the starting position in the variable where the first regex match begins. If the regex does not match the variable in the second argument in the PRXMATCH function, then `match_pos` is assigned the value of 0. The field `OcularYN` is a Boolean field that has a value of 1 if the observation is flagged as an ocular adverse event (i.e. `match_pos` greater than zero). Displayed in Figure 2B below are some observations from the Find_Ocular_AE data set. Notice that we were indeed successful flagging observation #4 for the OS part of the string and not the ou part of 'Dangerous', which is confirmed by noting the match starts at position 20 and not 7.

Figure 2B - Partial Results from Ocular AE Matching

	<i>Obs</i>	<i>Adverse Event Text</i>	<i>match_pos</i>	<i>OcularYN</i>
The 'ou' from 'headwound' does not match the definition of an ocular adverse event	4	Dangerous glaucoma OS	20	1
	5	migraine headache	0	0
	6	lower backache	0	0
	7	festering headwound	0	0
	8	O.D. has issues	1	1

Now consider the regular expression in Figure 2C below where the metacharacter `\b` is replaced by the character class `\s`³. Is the regular expression below equivalent to the one in Figure 2A?

Figure 2C - \b is a Zero-Width Assertion

The character class \s consumes at least one byte of space, but \b does not.

```

_re=prxparse('/\s[o0]\.?[uds]\.?/i');

```

Obs	Adverse Event Text	match_pos	OcularYN
8	O.D. has issues	0	0

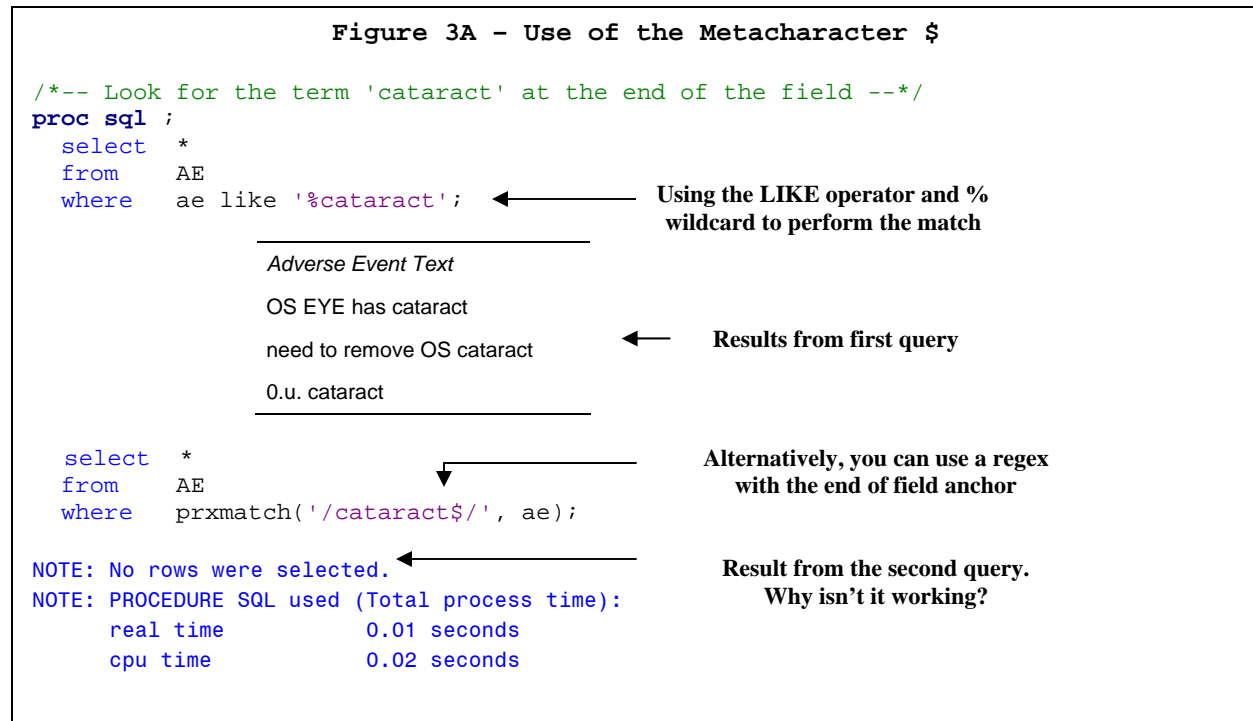
Inspecting the values of the fields in observation #8 we find that the two regexen are not equivalent. The reason is that `\s` consumes at least one byte of the character field. The string O.D. is in the first four bytes of the field, so the leading space character is not matched. The metacharacter `\b` does not consume any bytes, and in Perl this is referred to as a *zero-width assertion*. This supermodel-like phenomenon will show up again throughout this paper.

³ To refresh your memory, `\s` is the Whitespace character class which includes space, tab, carriage return character and some other peculiar beasts.

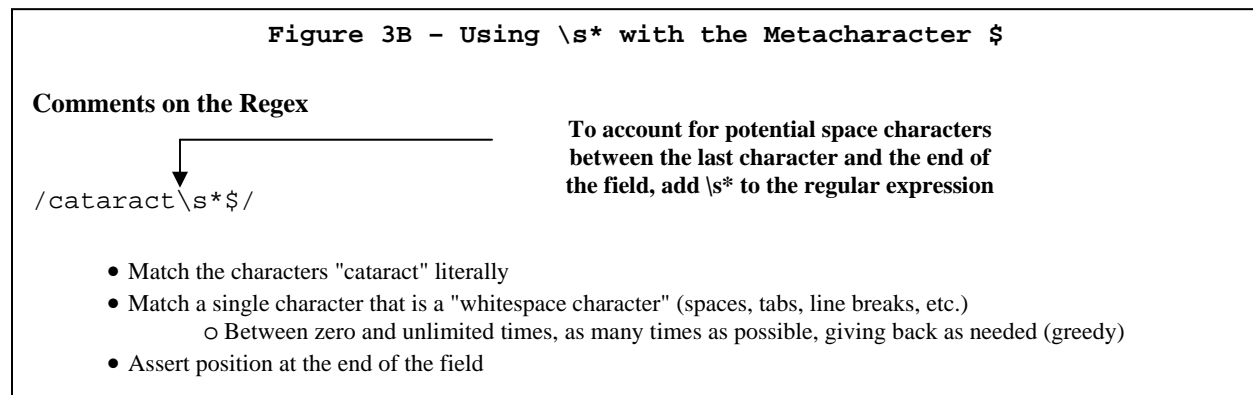
END OF FIELD BOUNDARY (\$)

The previous section noted that the word boundary `\b` assertion can be met by matching the end of a string. The metacharacter `$` is also a zero-width assertion that specifically matches the end of a character field. Both `\b` and `$` are sometimes referred to as *anchors*, as they enable you to match parts of the pattern surrounded by certain conditions.

Consider the queries below in Figure 3A where we like to select observations that have the term 'cataract' at the end of the field. Prior to being an aspiring PRX guru, you might have attacked this problem using the LIKE operator and the % wildcard available in the WHERE clause⁴. Now you can make use of the `$` metacharacter of Perl regular expressions to handle the job.



Why isn't the `$` metacharacter working as advertised? Well, when using the `$` metacharacter with the PRX function you need to account for all bytes in the character field. The AE field has length 27 and none of the three potential matches have 'cataract' in position 20 through 27. You have a couple of options to remedy this problem. One way would be to use the TRIM or STRIP functions on the second argument in the PRXMATCH function. Another way would be to account for the possibility of trailing spaces at the end of the fields, as shown below in Figure 3B.



⁴ Another way to perform the task is to use `substr(left(reverse(ae)),1,8)='tcaratac'`. People joke that regular expressions are 'write-only' because of the difficulty in reading them, but this solution may be as awkward, if not more.

Please keep in mind that \$ is a metacharacter, so it has a special meaning. If you need to match on the literal character '\$', you need to mask its special powers by preceding it with \, which is sometimes referred to a 'backwhacking'. To borrow a quote from frequent SAS-L poster David Cassell, "You backwhack special characters to make them normal, and you backwhack normal characters to make them special".

NON-CAPTURING BUFFERS

When writing regular expressions, you may find it necessary to use parentheses. One of the main functions of parentheses is to capture parts of a regex to be extracted using the PRXPOSN function. Another use is to group parts of the regex, particularly when you apply a quantifier on them, as demonstrated below in Figure 4A. In that example, we would like to match any observations that have two consecutive strings of 'very' separated by one or more space characters.

Figure 4A - Grouping Part of a Regex With Parentheses

```

/*- Look for 'very' followed by one or more spaces characters, twice -*/
proc sql ;
  select *
  from   AE
  where  prxmatch('/(very\s+){2}/', ae);
quit;

```

The parentheses in the regex create a backreference

Adverse Event Text
ou iop very very serious

Query result ↑

Comments on the Regex

`/(very\s+){2}/`

- Match the regular expression below and capture its match into backreference number 1
 - Exactly 2 times
 - Note: You repeated the backreference itself. The backreference will capture only the last iteration. Put the backreference inside a group and repeat that group to capture all iterations.
 - Match the characters "very" literally
 - Match a single character that is a "whitespace character" (spaces, tabs, line breaks, etc.)
 - Between one and unlimited times, as many times as possible, giving back as needed (greedy)

By using parentheses you are creating a *backreference*, which is sort of like a temporary internal variable. This is perfectly fine when you need to extract information captured in the parentheses. In the case above, it is not necessary to create a backreference since you only need to know that the matched occurred, but do not need to extract information. You can prevent creating a backreference by inserting '?' directly after of the open parens to make a *non-capturing buffer*, as shown below in Figure 4B. The regex will return the same result as the one above, but it does so making a more judicious use of computing resources. This plays a more important role as the number of observations and fields you are searching in your data set grows.

Figure 4B - Using a Non-Capturing Buffer

Comments on the Regex

`/(?:very\s+){2}/` ← Add ?: to make the parentheses non-capturing

- Match the regular expression below
 - Exactly 2 times
 - Match the characters "very" literally
 - Match a single character that is a "whitespace character" (spaces, tabs, line breaks, etc.)
 - Between one and unlimited times, as many times as possible, giving back as needed (greedy)

GREEDY and LAZY QUANTIFIERS

You may have noticed in the *Comments on the Regex* section of Figure 4B the following note: “match ... between one and unlimited times, as many times as possible, giving back as needed (greedy)”. The quantifiers {}, +, * and ? are said to be *greedy* because they allow the preceding expression to be matched as many times as possible while still allowing the entire regular expression to find a match. Let us take a peek under the hood of the regex engine to see how the match is performed using the previous example. There is only one match from the AE data set that the regular expression matches (“ou iop very very serious”), so let’s concentrate on that observation. The regex engine scans the beginning of the field looking for the character ‘v’. The first 10 bytes of the field are sequentially searched without finding a ‘v’, but at the eleventh byte the first character in the expression is found. The engine then looks for ‘e’ (successfully), then ‘r’ (successfully), and then ‘y’ (successfully). The next part of the pattern \s+ asks to match one or more space characters. This observation has three spaces in between ‘y’ and the ‘v’ of the next ‘very’. The engine then finds a space, but it does not start searching for the ‘v’ as required by the quantifier {2} just yet. Since more space characters follow the first space then the engine will continue to match them until it encounters a non-space character. This demonstrates the + quantifier acting greedy. Fortunately for us, the next non-space character found is a ‘v’, so the engine then continues the search for the next ‘very’.

The greediness of the + quantifier on the \s character class in the previous example may seem rather innocuous. However, when variable length quantifiers are applied to wildcards (e.g. .) and character classes with multiple elements (e.g. \w), greediness can significantly increase the time it takes to find a match. Since greedy quantifiers match as many characters as possible, it may match too much and prevent the entire expression from matching. In this case, the regex engine starts giving back one character at a time to see if the remaining part of the expression can be matched. The more backtracking the engine needs to do, then the longer the matching process takes.

To combat greediness we can add a ? after a quantifier makes it *lazy*, as demonstrated below in Figure 5. By adding the ? quantifier after another quantifier, we are asking the previous expression to match as few times as possible before continuing on to find an overall match. If the regex does not match in its entirety, then the expression that the lazy quantifier is operating on is allowed to match an additional character one at a time until it enables the entire expression to match.

Figure 5 - Making a Quantifier Lazy

Comments on the RegEx

`/(:?very\s+?) {2}/`

← Adding ? after the + makes the
quantifier lazy

- Match the regular expression below
 - Exactly 2 times
 - Match the characters "very" literally
 - Match a single character that is a "whitespace character" (spaces, tabs, line breaks, etc.)
 - Between one and unlimited times, as few times as possible, expanding as needed (lazy)

Are greedy quantifiers less desirable than lazy quantifiers? Well, it depends on the situation. The regex engine’s priority is to find a match. And if you extracting parts of a regular expression, you may need to code your regular expression specifically using a greedy or lazy quantifier. How much extra time it takes your lazy quantifier to expand or your greedy quantifier to give back in order to find a match may be data dependent. You should write your regular expression according to whatever knowledge you have about the data. Trial and error is not out of the realm of possibility.

LOOKAROUNDS

You may find it necessary to write a regular expression to find a certain string that is *not* followed by something else. Take for example where we would like to select observations with the string 'head' that is not followed by the string 'wound'. Consider the use of a negated character class in the query below in Figure 6A.

Figure 6A – Using Negated Character Classes to Perform a Non-Match

```
/*-- Find word 'head' not followed by 'wound' --*/
data AE2(drop=_) ;
  set AE ;
  if _n_=1 then _re=prxparse('/\bhead[^w][^o][^u][^n][^d]/i');
  retain _re;
  Head_No_Wound=(prxmatch(_re,ae)>0);
run;
```

Obs	Adverse Event Text	Head_No_Wound
5	migraine headache	1
7	festering headwound	0

← Partial display of
AE2 data set

Comments on the Regex

`/\bhead[^w][^o][^u][^n][^d]/`

- Assert position at a word boundary
- Match the characters "head" literally
- Match any character that is not a "w"
- Match any character that is not a "o"
- Match any character that is not a "u"
- Match any character that is not a "n"
- Match any character that is not a "d"

}

← Each of the negated character
classes consume one byte

We were successful in not matching 'festering headwound' with our regex as intended. However, writing the regex with negative character classes has a big drawback in that they require bytes to be matched. Suppose 'head' was located in at the end of the field in position 24 through 27 in the AE field. Though 'wound' does not follow 'head', the regex would not match because it still needs to match five more characters (i.e., not 'w', not 'o', etc.). The pitfalls of using negated character classes can be avoided by using a *negative lookahead assertion*. This zero-width assertion is formed by entering an expression that you do not want to match within (?!), as demonstrated below in Figure 6B.

Figure 6B – Using a Negative Lookahead Assertion

Comments on the Regex

`/\bhead(?!wound)/` ← Negative lookaheads (!) are zero-width assertions

- Assert position at a word boundary
- Match the characters "head" literally
- Assert that it is impossible to match the regex below starting at this position (negative lookahead)
 - Match the characters "wound" literally

The previous example demonstrated how you can look for something that is not followed by something else. Similarly, you can also write a regex to look for something not preceded by something else by using a *negative lookbehind assertion*. Suppose you wanted to flag observations with 'ache' that are not preceded by 'back'. The syntax for a negative lookbehind is the expression enclosed by (?<!), as demonstrated below in Figure 6C.

Figure 6C – Using a Negative Lookbehind Assertion

```
/*-- 'ache' not preceded by 'back' --*/
data AE3(drop=_:) ;
  set AE ;
  if _n_=1 then _re=prxparse('/(?<!back)ache/');
  retain _re;
  NoBack_Ache=(prxmatch(_re,ae)>0);
run;
```

Obs	Adverse Event Text	NoBack_Ache
5	migraine headache	1
6	lower backache	0
15	aches & pains all over body	1

Did not want
to flag this
observation

Comments on the Regex

/ (?<!back)ache /

Partial display of
AE3 data set

- Assert that it is impossible to match the regex below with the match ending at this position (negative lookbehind)
 - Match the characters "back" literally
- Match the characters "ache" literally

There is one important difference between negative lookaheads and lookbehinds. A wiseguy might answer “Yeah, they work in opposite directions!”, but that is exactly at the heart of the matter. Perl regex work on a variable from left to right. Without going into the gory details, they can only operate backwards in a limited manner. So if you are going to use a negative lookbehind assertion the expression must be of fixed length. Using quantifiers that can yield variable length, such as ?, * and +, is not allowed. Alternation is allowed but only if the alternatives have the same length. For example, a positive lookbehind for asserting ‘os’ or “OD” occurs before something written as (?<=(os|OD)) is permissible. Negative lookaheads are not binded by the fixed length constraint.

The lookarounds discussed so far have been of the negative sort. Similarly, your regex can contain *positive lookaheads* and *lookbehinds* and the syntax for these assertions can be found below in Figure 6D. These assertions have zero-width and their buffers are non-capturing. Positive lookbehinds are faced with the fixed length constraint. All of the lookarounds discussed are another form of anchors, like \b and \$.

Figure 6D – Syntax for Positive Lookarounds

Positive Lookahead: (?=)

Positive Lookbehind: (?<=)

Like their negative lookahead counterparts, positive lookarounds are zero-width assertions.

Let us examine an example where positive lookaheads are useful. Suppose we need to create a new variable where any instances of 'headwound' are displayed as 'head wound'. Using the PRXCHANGE function, we replace any instances where the pattern matches and insert a space character in the appropriate spot, as demonstrated below in Figure 6E.

Figure 6E – Using a Positive Lookahead Assertion

```
/*-- Replace instances of 'headwound' with 'head wound' --*/
proc sql ;
  select  ae,
         prxchange('s/(head)(?=wound)/$1 /i',-1, ae) label='Revised AE'
  from    Ae;
quit ;
```

<i>Adverse Event Text</i>	<i>Revised AE</i>
festering headwound	festering head wound

Comments on the Regex

`s/(head)(?=wound)/$1 /`

- Match the regular expression below and capture its match into backreference number 1
 - Match the characters "head" literally
- Assert that the regex below can be matched, starting at this position (positive lookahead)
 - Match the characters "wound" literally
- Perform the substitution (create a new variable)
 - At the starting position of a match, insert
 - The first (and only) backreference
 - A space

Partial result from query

Could the substitution regular expression have been written as `'s/headwound/head wound/'`? Indeed it could have, but it would have required replacing 9 bytes with 10 bytes for each match. Using the positive lookahead, only 4 bytes are captured in the first backreference ('head') and are replaced with 5 bytes ('head '). This example demonstrates the potential gains in efficiency by using a positive lookahead.

CONCLUSION

With the introduction of SAS Version 9, Perl regular expressions are now at the disposal of SAS users. The goal of this paper was to survey some of the intermediate topics of regular expressions. Having a solid understanding of both the fundamental and more difficult concepts are essential to writing a correct and efficient regular expression. After reading this paper, hopefully you are feeling confident to use the PRX functions and call routines to tackle the toughest text manipulation problems that come your way.

REFERENCES

Cassell, David L. (2005), “PRX Functions and Call Routines”, *Proceedings of the 30th Annual SAS Users Group International*

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 Expressions with Examples from Clinical Data”, *PharmaSUG 2005 Proceedings*

SAS OnLineDoc®

Schwartz, R., Phoenix, T., D Foy, B. , *Learning Perl 4th Edition*

Tabladillo, Mark (2005), “Mapping an Exclusive Regular Expression Strategy”, *Proceedings of the 30th Annual SAS Users Group International*

Wall, L., Christiansen, T., Schwartz, R., *Programming Perl 2nd Edition*

ACKNOWLEDGEMENTS

The author would like to thank David Cassell and Mike Dupin for their insightful comments in reviewing this paper.

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.

Comments of the regular expressions were generated with the help of RegexBuddy software (<http://www.regexbuddy.com>).

CONTACT INFORMATION

Your comments and questions are valued and encouraged.

Contact the author at:

Kenneth W. Borowiak
PPD, Inc.
3900N Paramount Parkway
Morrisville NC 27560

E-mail: Ken.Borowiak@rtp.ppd.com
EvilPettingZoo97@aol.com

APPENDIX

```
/*- An adverse event (AE) data set from a fictitious ocular (i.e. eye
    related) clinical trial -*/

data AE;
  attrib ae length=$27 label="Adverse Event Text";
  input ;
  ae=_infile_ ;
datalines;
OS EYE has cataract
iop od = 50
need to remove OS cataract
Dangerous glaucoma OS
migraine headache
lower backache
festering headwound
O.D. has issues
O.u. cataract
GSW selling hot ipods
gun shot wound to back
Ou dry eye
ou iop very    very serious
put food colouring in OS
aches & pains all over body
;
run;
```