

# PRX Functions and Call Routines: There Is Hardly Anything Regular About Them

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

## ABSTRACT

The PRX functions and call routines are a family of powerful tools to work with text strings. Introduced in Version 9, the PRX functions give you access to the Perl-style regular expressions to match, validate, and replace patterns of text. While many of the traditional character string functions (INDEX, TRANWRD, COMPRESS, etc.) and the LIKE operator can handle many of the trivial text matching and replacement tasks that come your way, the PRX functions shine in the face of the most complicated cases. Though regular expressions offer an immense amount pattern matching capabilities, this paper will cover only the basics of PRX.

**Keywords:** regular expression, metacharacters, character class, alternation, anchors, word boundaries.

## INTRODUCTION

A *regular expression* is a string that characterizes 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 incorporated into SAS.

Figure 1 below contains an example of a regular expression to match the two byte string Mr. You can use the PRXMATCH function to match text, which takes two arguments. The first argument is the regex (short for regular expression), which is a character string nested between a pair of quotes. Then there is a pair of forward slashes, which are the delimiters of the regex. Mr is the two bytes of literal text to be matched. The second argument is the character field to search for the text, which is NAME in this example. If the pattern is found in the source field, then an integer indicating the location of the byte where the match begins is returned. If the result is greater than zero, then a match was found. The structure to PRXMATCH is similar to the INDEX function.

---

**Figure 1 - Regular Expression to Match Literal Text**

---

```
proc print data=characters label obs ;
  where prxmatch( '/^Mr/' , name )>0 ;
run ;
```

---

As an alert reader, you may have noticed the ^ in the regex. The caret in this example is not a character to be matched, but denotes that the match must start at the beginning of the string. The ^ is one of many *metacharacters* that will be encountered throughout the paper. A metacharacter is a reserved character with a special meaning that adds an extra element of control.

## WHAT CAN YOU DO WITH PRX?

The PRX functions and call routines enable you to perform a variety of text manipulation tasks, which include:

- Matching (PRXMATCH)
  - Subsetting - Restrict inputting and outputting observations with a WHERE clause or IF statement
  - Conditional logic - Process observations matching a pattern in a DATA step (IF-THEN-ELSE statements and DO and SELECT blocks) and PROC SQL (CASE statements). You can also use PRX to match a pattern of text in a macro variable in the Macro facility via %sysfunc().
  - Validate - Confirm the data or input matches an acceptable pattern, such as phone numbers, social security numbers, and e-mail addresses

- Extracting (PRXPOSN and PRXSUBSTR) - Create a new field that contains some or all of the matched text
- Substitution (PRXCHANGE)
  - Find text that matches a pattern and then replace it with different text
  - Find text that matches a pattern and then remove it (compression)

### WHY WOULD YOU WANT TO LEARN PRX?

If you are new to PRX, one of the first things that you may discover is that they look rather cryptic. The syntax for regular expressions in PRX could be considered a language in themselves. Outside of matching literal text, writing even a simple regular expression can be difficult in the beginning. Reading a regular expression is even harder! If you have text string functions such as INDEX, TRANWRD, and COMPRESS, the LIKE and CONTAINS operators, and the colon modifier in your arsenal and are facing such a steep learning curve, why would you bother learning PRX?

- Flexibility - As we delve into the functionality of regular expressions, you will see the tremendous amount of flexibility offered by using concepts such as *character classes* and *quantifiers*.
- Ubiquity - In addition to being available in SAS, many other programming languages (e.g. Perl, Java, PL/SQL), text editors, and applications have regex capabilities. As regular expressions continue to grow in popularity, chances are that you will encounter them at some point.
- Portable Syntax - Continuing from the previous point, the syntax of the many flavors of regex is often interchangeable. And where the syntax does not translate exactly, an analogous concept often exists.
- Support - There is a growing collection of documentation on PRX and regular expressions in the SAS community. However, there is an enormous amount of documentation on regex outside of the SAS community which you can take advantage of because of the portable syntax. There are a number of websites that contain user contributed regular expressions to match both common and arcane patterns.

This paper does not attempt to cover all aspects of the PRX functions and call routines and regular expressions. This paper serves as a complement to other introductory PRX papers written by Cassell (3) and Cody (4). Borowiak (1) examines some of the intermediate to advanced regex topics. Friedl (5) provides a comprehensive overview of regular expressions.

All of examples presented in the paper have been tested with SAS Version 9.1.3 Service Pack 4. Enhancements to PRX introduced in V9.2 will not be discussed and the interested reader should consult the relevant documentation (6).

### SAMPLE DATA

The PRX functions and call routines work on character data. Since they offer quite a bit of flexibility, they are ideal for working with messy and unstructured data. Throughout the paper, examples will use the CHARACTERS data set defined in Figure 2, which is a free-text captured list of movie characters.

**Figure 2 - Sample Data**


---

```

data characters ;
  input name $40. ;
  datalines ;
MR Bigglesworth
Mini-mr biggggleswerth
Mr. Austin D. Powers
dr evil
MINIME(1/8th size of dr evil)
mr bIgglesWorTH
Ml$$ foxy cleopatra
Scott Evil
MRS. KENSINGTON
;
run ;

```

---

## MATCHING

The majority of the discussion of regular expressions in this paper will be presented in the context of matching. However, the concepts and syntax can also be applied to text extraction and substitution. Any syntax pertaining specifically to extraction and substitution will be presented later on in those sections.

## CASE SENSITIVITY

The results of the query from Figure 1 in the Introduction are presented in Figure 3.

**Figure 3 - Case Sensitivity of Regex**


---

```

/* Match literal string 'Mr' at the beginning of the field */
prxmatch( '/^Mr/', name )>0 ;

RESULT
Obs      Name
3        Mr. Austin D. Powers

```

---

Note that only the third observation matched the regex. The first ( MR Bigglesworth ), sixth ( mr bIgglesWorTH ), and ninth ( MRS. KENSINGTON ) observations were not matched because of the one of the first two bytes did not match the case of the character specified in the regex. By default, regular expressions are case-sensitive. However, you can override the default setting by using a *modifier*. In particular, specifying the *i* modifier after the closing delimiter of the regex makes the match case insensitive, as shown in Figure 4.

**Figure 4 - i Modifier to Make the Match Case-Insensitive**


---

```

/* A case-insensitive search for 'Mr' at the beginning of the field */
prxmatch( '/^Mr/i', name )>0 ;

RESULT
Obs      Name
1        MR Bigglesworth
3        Mr. Austin D. Powers
6        mr bIgglesWorTH
9        MRS. KENSINGTON

```

---

## CHARACTER CLASSES

**User-Defined** A character class is one of the variety of ways to build flexibility into the pattern search. You specify a user-defined character class with a pair of brackets (i.e. [ ]) and it allows any single character specified within to be matched. One way to deal with case-sensitivity in pattern matching is to use a character class. Suppose you are interested in displaying observations that begin with `M` followed by either a lower or uppercase `r`. You could then write the regex as displayed in Figure 5.

---

**Figure 5** -Using a Character Class to Address Partial Case-Sensitivity

---

```
/* Match 'M' followed by 'R' or 'r' at the beginning of the field */
prxmatch( '/^M[Rr]/', name )>0 ;

RESULT
Obs      Name
1        MR Bigglesworth
3        Mr. Austin D. Powers
9        MRS. KENSINGTON
```

---

When crafting a regex that needs to be case sensitive except at few locations with limited surrogates, character classes are a viable way to proceed. As a counter-example, consider a case insensitive search for the text `bigglesworth` in Figure 6. The regex is unnecessarily lengthy and would be more easily accomplished by using the `i` modifier.

---

**Figure 6** -An Unwise Application of Character Classes

---

```
/* A case insensitive search for the text 'bigglesworth' */
prxmatch( '/[Bb][Ii][Gg][Gg][lL][eE][Ss][Ww][oO][Rr][tT][Hh]/', name )>0 ;

RESULT
Obs      Name
1        MR Bigglesworth
```

---

User-defined character classes have their own set of metacharacters. The metacharacters that can be used within a character class do not necessarily have the same meaning when used outside of them. For example, `[^M]` is a *negated character class* that allows any character except `M` to be matched. The metacharacter `^` in the first position inside the character class invokes the negation. This differs from the use of `^` in the first position of a regex, as in Figures 1 and 2, which acts as an *anchor* that requires the match to begin at the beginning of the field. Another character class metacharacter is the dash (i.e. `-`), which serves to indicate a range when specified between two characters. The character class `[a-e]` means any lower case character between `a` and `e`. In order to make the dash character part of the character class and not to be used as a metacharacter then make it either the first or last character within the pair of brackets<sup>1</sup>.

**Predefined** There are many predefined character classes at your disposal, many of which of displayed in Table 1.

---

<sup>1</sup>These is another way to include the dash a character to be matched in a character class that will be discussed in the More Metacharacters section.

Table 1:

Predefined Character Classes	
<code>\d</code>	Any digit character, equivalent to <code>[0-9]</code>
<code>\D</code>	Any non-digit character, equivalent to <code>[^0-9]</code>
<code>\s</code>	Any whitespace character, including space and tab characters
<code>\S</code>	Any non-whitespace character, equivalent to <code>[^\s]</code>
<code>\w</code>	Any 'word' character, equivalent to <code>[a-zA-Z0-9_]</code>
<code>\W</code>	Any non-word character, equivalent to <code>[^a-zA-Z0-9_]</code>
<code>[:alpha:]</code>	Any alphabetic character, equivalent to <code>[a-zA-Z]</code>
<code>[:alphanum:]</code>	Any alphanumeric character, equivalent to <code>[a-zA-Z0-9]</code>
<code>[:digit:]</code>	Any digit character, equivalent to <code>\d</code>
<code>[:lower:]</code>	Any lowercase alphabetic character, equivalent to <code>[a-z]</code>
<code>[:punct:]</code>	Any punctuation character
<code>[:upper:]</code>	Any uppercase alphabetic character, equivalent to <code>[A-Z]</code>

The last six entries in the table are POSIX bracketed expressions. To use them, they must be enclosed within a second a pair of brackets, as displayed in Figure 7. The other predefined character classes that begin with a `\` do not need to be enclosed within a pair of brackets.

---

**Figure 7** -Using a Predefined Character Class
 

---

```

/* A search for any digit character in NAME */
prxmatch( '/[[:digit:]]/', name )>0 ;

/* or */

prxmatch( '/\d/', name )>0 ;

RESULT
Obs      Name
5        MINIME(1/8th size of dr evil)
7        M1$$ foxy cleopatra

```

---

## ALTERNATION

Another method that can be used to build in flexibility with regex is *alternation*, which is specified with a `|` and is analogous to the OR logical operator. Suppose we continue with the task presented in Figure 5, you could then write the regex as displayed in Figure 8.

---

**Figure 8** -Using Alternation to Address Partial Case-Sensitivity
 

---

```

/* Match 'M' followed by 'R' or 'r' at the beginning of the field */
prxmatch( '/^M(R|r)/', name )>0 ;

RESULT
Obs      Name
1        MR Bigglesworth
3        Mr. Austin D. Powers
9        MRS. KENSINGTON

```

---

Note that the alternatives lower and upper case `R` are grouped within parenthesis. Without the grouping parenthe-

sis, you would have created two subexpressions, namely `^MR/` and `/r/`. However, using parenthesis `()` to group the alternatives causes a side effect in this simple pattern matching exercise. The parenthesis cause the characters matched within to be 'captured' and held in a temporary variable, which plays an important role when we discuss text extraction later on in the paper. Since we are not interested in pulling out the matched characters in this example, there is no need to expend the resources needed to maintain the temporary variable. For grouping only, we can use non-capturing parenthesis specified with `(?:)`, as demonstrated below in Figure 9.

---

**Figure 9** -Using Alternation with Non-Capturing Parenthesis to Address Partial Case-Sensitivity

---

```
/* Match 'M' followed by 'R' or 'r' at the beginning of the field */
prxmatch( '^M(?:R|r)', name )>0 ;
```

RESULT

Obs	Name
1	MR Bigglesworth
3	Mr. Austin D. Powers
9	MRS. KENSINGTON

---

Contrary to the examples provided showing how to address partial case-sensitivity, in general, character classes and alternation are not the same thing. Character classes are a collection of characters that require any one of them to be matched. With alternation, the lengths of the alternatives do not have to be the same. A character class may even be part of one of the alternatives. In Figure 10, a regex is crafted to find observations with the known members of the `Evil` family.

---

**Figure 10** -Alternatives Do Not Need to Be of the Same Length

---

```
/* Ignoring case, match 'DR' or 'SCOTT' at the beginning
of the field followed by a space and then 'EVIL' */
prxmatch( '^(?:DR|SCOTT)\sEVIL/i', name )>0 ;
```

RESULT

Obs	Name
4	dr evil
8	Scott Evil

---

## MORE METACHARACTERS

**Period** One of the most powerful metacharacters in regular expressions is the period or dot, which matches any character that consumes a byte. The period in regex is analogous to the underscore used in conjunction with the `LIKE` operator. Consider the regex in Figure 11 that attempts a case-insensitive match for the string `MR.` at the beginning of the field.

---

**Figure 11** -Using the Dot Metacharacter

---

```
/* A case-insensitive match for 'MR.' at the beginning of the field */
prxmatch( '^MR./i', name )>0 ;
```

RESULT

Obs	Name
1	MR Bigglesworth
3	Mr. Austin D. Powers
6	mr bIgglesWorTH
9	MRS. KENSINGTON

---

You may have noticed that there is not a period in the third byte of the first, third, and ninth observations. Why were these returned as matches? Recall that the dot is a metacharacter and it matched a space character in the third byte for the first and third observations and S for the ninth. To match the period as literal text rather than use it as a metacharacter it must be preceded by a backslash. This is true when trying to match any character that happens to be a metacharacter and is akin to masking percent signs and ampersands in the Macro facility. Masking metacharacters in regular expressions is sometimes referred to as *backwhacking*. Figure 12 demonstrates backwhacking the period in the third character matched by the regex.

**Figure 12** -Matching a Period Literally and Not as a Metacharacter

```
/* A case-insensitive match for 'MR.' at the beginning of the field */
prxmatch('/^MR\./i', name)>0;

RESULT
Obs      Name
3        Mr. Austin D. Powers
```

**Quantifiers** A *quantifier* or *repetition factor* specifies how often the preceding subexpression regular should match. Table 2 displays the predefined and user-defined forms of quantifiers.

Table 2:

Quantifiers	
+	Match the previous subexpression one or more times
*	Match the previous subexpression one or more times or not at all
?	Match previous subexpression one time or not at all
{m,n}	Match previous subexpression at least m times but no more than n times, where m and n are positive integers. If n is omitted then the subexpression is allowed to match as many times as possible.

Consider the two functionally equivalent regular expressions below in Figure 13 that aim to match some form of the string `bigglesworth`. A preliminary look at the data suggests that the `g` could appear more than twice, so this is built into in the regexen.

**Figure 13** -Using Quantifiers

```
/* Find misspellings of 'bigglesworth' */
prxmatch('/bigg+lesw(o|e|i)rth/i', name )>0 ;

/* or */

prxmatch('/big{2,}lesw[oei]rth/i', name )>0 ;

RESULT
Obs      Name
1        MR Bigglesworth
2        Mini-mr biggggleswerth
6        mr bIgglesWorTH
```

In the first regex, the literal text `big` is searched for. Then the `+` quantifier is applied to the second `g`, meaning it must be found at least one more time before checking the remainder of the pattern. In the second regex, only one

$\mathcal{G}$  is needed because the user-defined quantifier  $\{2, \}$  requires it to be matched at least twice before checking the remainder of the pattern.

**End of Field Anchor** You have already seen the beginning of field anchor  $\wedge$ . There is also an end of field anchor, which is denoted by a dollar sign (i.e.  $\$$ ). Consider the regex in Figure 14 that searches for the string `Evil` at the end of the field.

---

**Figure 14** -Using the End of Field Anchor

---

```
/* A case-insensitive search for 'Evil' at the end of the field */
prxmatch( '/Evil\s*$/i', name )>0 ;

RESULT
Obs          Name
4            dr evil
8            Scott Evil
```

---

The regex is written to allow whitespace between `Evil` and the end of the field by specifying `\s*`. If the regex had been written as `/Evil$/i`, then the string `Evil` would have needed to start at position 37 in the field `NAME`.

**Word Boundary** The anchors  $\wedge$  and  $\$$  are assertions that require the regex to match at the beginning and end of the field, respectively. Another anchor is a *word boundary* and is specified with `\b`. A word boundary asserts the location between a word character (i.e. `\w`) and a non-word character (i.e. `\W`). A word boundary is used in Figure 15 where a regex is used for a case-insensitive search for the word `MR` at the beginning of the field.

---

**Figure 15** -Using a Word Boundary

---

```
/* A case-insensitive match for the word 'MR' at the beginning of the field */
prxmatch( '/^MR\b/i', name )>0 ;

RESULT
Obs          Name
1            MR Bigglesworth
3            Mr. Austin D. Powers
6            mr bIgglesWorTH
```

---

In the first and sixth observations, the second byte `r` is a word character that is followed by a space. The space is a non-word character, so it satisfies the word boundary condition. The third byte in the third observation is a period, so that too satisfies the word boundary condition. The ninth observation (`MRS . KENSINGTON`) does not meet the word boundary condition because the third byte `S` is a word character.

## EXTRACTION

Another useful aspect of PRX is the ability extract the text matched by PRXMATCH. This section will explore two ways to use PRX to extract matched patterns.

### PRXPOSN

In the section on Alternation, it was called to your attention that text matched within parenthesis is held in internal temporary variables. These temporary held fields are referred to as *capture buffers*. To move the text from a capture buffer to a SAS variable you can use the PRXPOSN function. In Figure 16, a search is performed for abbreviated titles (i.e. Mr, Mrs, Dr) of the movie characters at the beginning of the field within a DATA step. If a match is found, then a new variable called TITLE is created with the captured text.

**Figure 16** -Extracting Text from a Capture Using PRXPOSN

```

data _null_ ;
  set characters ;

  * Compile the regex ;
  re=prxparse( '/^([MD]rs?\.\.?) /i' ) ;

  * Search for a match ;
  start_pos=prxmatch( re, name ) ;

  * Extract the first (and only) capture buffer ;
  * Must use PRXMATCH before PRXPOSN ;
  length title $5 ;
  title=prxposn( re, 1, name ) ;

  put name=    @38 start_pos=    @52 title= ;
run ;

```

**Result**

name=MR Bigglesworth	start_pos=1	title=MR
name=Mini-mr biggggleswerth	start_pos=0	title=
name=Mr. Austin D. Powers	start_pos=1	title=Mr.
name=dr evil	start_pos=1	title=dr
name=MINIME(1/8th size of dr evil)	start_pos=0	title=
name=mr bIgglesWorTH	start_pos=1	title=mr
name=Ml\$\$ foxy cleopatra	start_pos=0	title=
name=Scott Evil	start_pos=0	title=
name=MRS. KENSINGTON	start_pos=1	title=MRS.

You may have noticed a number of differences in this example from the others where we were only selecting observations matching the pattern. The first is the inclusion of the PRXPARSE function. You can directly enter the regex as the first argument to PRXMATCH when all you interested in doing is matching. For extracting, you will need a compiled version of the regex in the first argument of PRXPOSN, which is what PRXPARSE does. Since you will need to use PRXMATCH to create the capture buffers that PRXPOSN will extract, you can pass the compiled version of the regex to PRXMATCH as well. The second argument to PRXPOSN is a positive integer that indicates which capture buffer you would like to extract, which is 1 in this example. The third argument is the field to which the pattern was applied. From the results, you can see that when a match was found (i.e. `start_pos>0`) that field TITLE is populated with the extracted text.

**PRXSUBSTR**

Another way to extract text matched by a regular expression is through the CALL PRXSUBSTR routine. Similar to using the PRXPOSN function, extracting text via CALL PRXSUBSTR requires multiple steps. The DATA step in Figure 17 replicates the same task presented in Figure 16 with CALL PRXSUBSTR.

**Figure 17** -Extracting Text Via CALL PRXSUBSTR

---

```

data _null_ ;
  set characters ;

  * Compile the regex ;
  re=prxparse( '/^([MD]rs?\.\.?) /i' ) ;

  * Identify starting position and length of the matched pattern ;
  call prxsubstr( re , name, start_pos, match_length ) ;

  * Extract text only if a match is found ;
  length title $5 ;
  if start_pos>0 then title=substr( name, start_pos, match_length ) ;

  put Name= @38 Start_Pos= @55 Match_length= @72 Title= ;
run ;

```

name=MR Bigglesworth	start_pos=1	match_length=2	title=MR
name=Mini-mr biggggleswerth	start_pos=0	match_length=0	title=
name=Mr. Austin D. Powers	start_pos=1	match_length=3	title=Mr.
name=dr evil	start_pos=1	match_length=2	title=dr
name=MINIME(1/8th size of dr evil)	start_pos=0	match_length=0	title=
name=mr bIgglesWorTH	start_pos=1	match_length=2	title=mr
name=Ml\$\$ foxy cleopatra	start_pos=0	match_length=0	title=
name=Scott Evil	start_pos=0	match_length=0	title=
name=MRS. KENSINGTON	start_pos=1	match_length=4	title=MRS.

---

The first argument to CALL PRXSUBSTR is the compiled version of the regular expression and the second argument is the field to which the pattern was applied. The call routine implicitly performs the pattern match. The call routine populates two fields containing the starting position and length of the substring that matches a pattern, which are START\_POS and MATCH\_LENGTH, respectively. The values of these fields are used in the second and third arguments of a subsequent call to the SUBSTR function in order to extract the text. A check for a pattern match (i.e. start\_pos>0) is done to avoid notes in the Log regarding invalid argument values in the call to the SUBSTR function.

## SUBSTITUTION

PRX is also a powerful tool for substituting patterns of text. In Figure 18, the PRXCHANGE function is used to replace abbreviations for Mister with the literal text.

**Figure 18** -Replacing a Matched Pattern Via PRXCHANGE

```

proc sql number ;
  select  Name
         , prxchange( 's/mr\b\.?/Mister/i', -1, name ) as Name2
  from    Characters
  ;
quit ;

```

Row	Name	Name2
1	MR Bigglesworth	Mister Bigglesworth
2	Mini-mr biggggleswerth	Mini-Mister biggggleswerth
3	Mr. Austin D. Powers	Mister Austin D. Powers
4	dr evil	dr evil
5	MINIME(1/8th size of dr evil)	MINIME(1/8th size of dr evil)
6	mr bIgglesWorTH	Mister bIgglesWorTH
7	Ml\$\$ foxy cleopatra	Ml\$\$ foxy cleopatra
8	Scott Evil	Scott Evil
9	MRS. KENSINGTON	MRS. KENSINGTON

The first argument to the PRXCHANGE function in this example is the explicit version of the regular expression. You can also pass a variable that is a compiled version of the regex as the first argument. A substitution regex has some requirements that make it different from a regex used to match or extract. The first requirement is the inclusion of an `s` before the first delimiter. Between the first and second delimiter is the regex with the pattern to match. If a match is found, the text is replaced with contents between the second and third delimiters. The second argument to the PRXCHANGE function is an integer indicating how many occurrences of the matched pattern should be replaced in the source field. Specifying `-1` in the second argument means to replace all occurrences. The third argument is the field to which the pattern was applied, which is `NAME` in this example.

Some other important points regarding substitution regular expressions:

- There exists a PRXCHANGE call routine which capable of populating a field with the number of times the substitution was performed. `CALL PRXCHANGE` is not permitted in PROC SQL.
- While your substitution regex requires three delimiters, you are not required to specify anything between the second and third delimiters. This would result in the matched pattern being removed or COMPRESSED.
- The contents between the second and third delimiters does not necessarily need to be literal text. If you create any capture buffers with the first part of the regex then you can reference them with a dollar sign. For example, `%2` allows you to reference the second capture buffer created in the matching portion of the substitution regex.

## CONCLUSION

A considerable amount of material has been discussed that should enable you to begin pattern matching using regular expressions through the PRX functions and call routines. However, there are still many important concepts that are important in order to master regex that are beyond the scope of this paper. Once you gain some comfort with the rudimentary concepts discussed here, you will want to research the following regex topics:

- The internal workings of the Perl regex engine and the optimization techniques it employs to efficiently find a match. You can use the PRXDEBUG call routine to do this.
- Ways to write a regex so the optimization techniques of the engine can be invoked.
- Using positive and negative lookahead assertions
- How to make quantifiers greedy or lazy and how this effects the text that is matched, captured or replaced
- Using mode modifiers and mode modifying spans to alter behavior for parts of the regex

## REFERENCES

- [1] Borowiak, Kenneth W., "Perl Regular Expressions 102" SGF 2007  
<http://www2.sas.com/proceedings/forum2007/135-2007.pdf>
- [2] Borowiak, Kenneth W., "Sensitivity Training for PRXers" NESUG 2007  
<http://www.nesug.info/Proceedings/nesug07/cc/cc05.pdf>
- [3] Cassell, David L., "The Basics of the PRX Functions" SGF 2007  
<http://www2.sas.com/proceedings/forum2007/223-2007.pdf>
- [4] Cody, Ronald P., "An Introduction to Perl Regular Expressions in V9" SUGI29  
<http://www2.sas.com/proceedings/sugi29/265-29.pdf>
- [5] Friedl, Jeffrey E.F., *Mastering Regular Expressions 3rd Edition*
- [6] SAS OnlineDoc®  
Pattern Matching Using Perl Regular Expressions (PRX)  
PRXPARSE Function  
Tables of Perl Regular Expression (PRX) Metacharacters

## ACKNOWLEDGEMENTS

The author would like to thank Jenni Borowiak, Bill Deese, Toby Dunn and Jim Worley 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 Perl Copyleft.

This document was typeset with L<sup>A</sup>T<sub>E</sub>X.

## CONTACT INFORMATION

Your comments and questions are valued and encouraged.

Ken Borowiak  
3900 Paramount Parkway  
Morrisville, NC 27560

Ken.Borowiak@rtp.ppd.com  
EvilPettingZoo97@aol.com  
(919) 462-5373