Paper 110-31

An Introduction to Perl Regular Expressions in SAS 9

Ron Cody, Robert Wood Johnson Medical School, Piscataway, NJ

Introduction

Perl regular expressions were added to SAS in Version 9. SAS regular expressions (similar to Perl regular expressions but using a different syntax to indicate text patterns) have actually been around since version 6.12, but many SAS users are unfamiliar with either SAS or Perl regular expressions. Both SAS regular expressions (the RX functions) and Perl regular expressions (the PRX functions) allow you to locate patterns in text strings. For example, you could write a regular expression to look for three digits, a dash, two digits, a dash, followed by four digits (the general form of a social security number). The syntax of both SAS and Perl regular expressions allows you to search for classes of characters (digits, letters, non-digits, etc.) as well as specific character values.

Since SAS already has such a powerful set of string functions, you may wonder why you need regular expressions. Many of the string processing tasks can be performed either with the traditional character functions or regular expressions. However, regular expressions can sometimes provide a much more compact solution to a complicated string manipulation task. Regular expressions are especially useful for reading highly unstructured data streams. For example, you may have text and numbers all jumbled up in a data file and you want to extract all of the numbers on each line that contains numbers.

Once a pattern is found, you can obtain the position of the pattern, extract a substring, or substitute a string.

A Brief tutorial on Perl regular expressions

I have heard it said that Perl regular expressions are "write only." That means, with some practice, you can become fairly accomplished at writing regular expressions, but reading them, even the ones you wrote yourself, is quite difficult. I strongly suggest that you **comment** any regular expressions you write so that you will be able to change or correct your program at a future time.

The PRXPARSE function is used to create a regular expression. Since this expression is compiled, it is usually placed in the DATA step following a statement such as IF _N_ = 1 then …. Since this statement is executed only once, you also need to retain the value returned by the PRXPARSE function. This combination of combining _N_ and RETAIN when used with the PRXPARSE function is good programming technique since you avoid executing the function for each iteration of the DATA step. So, to get started, let's take a look at the simplest type of regular expression, an exact text match. Note: each of these functions will be described in detail, later in the tutorial.

Program 1: Using a Perl regular expression to locate lines with an exact text match

```
***Primary functions: PRXPARSE, PRXMATCH;

DATA _NULL_;
  TITLE "Perl Regular Expression Tutorial – Program 1";

  IF _N_ = 1 THEN PATTERN_NUM = PRXPARSE("/cat/");
  *Exact match for the letters 'cat' anywhere in the string;
  RETAIN PATTERN_NUM;

  INPUT STRING $30.;
  POSITION = PRXMATCH(PATTERN_NUM,STRING);
  FILE PRINT;
  PUT PATTERN_NUM= STRING= POSITION=;
DATALINES;
There is a cat in this line.
Does not match CAT
cat in the beginning
At the end, a cat
cat
;
```

Explanation:
You write your Perl regular expression as the argument of the PRXPARSE function. The single or double quotes inside the parentheses are part of the SAS syntax. Everything else is a standard Perl regular expression. In this

example, we are using the forward slashes (/) as the default Perl delimiters. The letters 'cat' inside the slashes specify an exact match to the characters "cat". Each time you compile a regular expression, SAS assigns sequential numbers to the resulting expression. This number is needed to perform searches by the other PRX functions such as PRXMATCH, PRXCHANGE, PRXNEXT, PRXSUBSTR, PRXPAREN, or PRXPOSN. Thus, the value of PATTERN_NUM in this program is one. In this simple example, the PRXMATCH function is used to return the position of the word "cat" in each of the strings. The two arguments in the PRXMATCH function are the return code from the PRXPARSE function and the string to be searched. The result is the first position where the word "cat" is found in each string. If there is no match, the PRXMATCH function returns a zero. Let's look at the output from **Program 1**:

```
Perl Regular Expression Tutorial - Program 1
PATTERN_NUM=1 STRING=There is a cat in this line. POSITION=12
PATTERN_NUM=1 STRING=Does not match CAT POSITION=0
PATTERN_NUM=1 STRING=cat in the beginning POSITION=1
PATTERN_NUM=1 STRING=At the end, a cat POSITION=15
PATTERN_NUM=1 STRING=cat POSITION=1
```

Notice that the value of PATTERN_NUM is 1 in each observation and the value of POSITION is the location of the letter "c" in "cat" in each of the strings. In the second line of output, the value of POSITION is 0 since the word "cat" (lowercase) was not present in that string.

Be careful. Spaces count. For example, if you change the PRXPARSE line to read:

```
IF _N_ = 1 THEN PATTERN_NUM = PRXPARSE("/ cat /");
```

then the output will be:

```
PATTERN_NUM=1 STRING=There is a cat in this line. POSITION=11
PATTERN_NUM=1 STRING=Does not match CAT POSITION=0
PATTERN_NUM=1 STRING=cat in the beginning POSITION=0
PATTERN_NUM=1 STRING=At the end, a cat POSITION=14
PATTERN_NUM=1 STRING=cat POSITION=0
```

Notice that the strings in lines 3 and 5 no longer match because the regular expression has a space before and after the word "cat." (The reason there is a match in the fourth observation is that the length of STRING is 30 and there are trailing blanks after the word "cat.")

Perl regular expressions use special characters (called metacharacters) to represent classes of characters. (Named in honor of Will Rogers: "I never meta character I didn't like.) Before we present a table of Perl regular expression metacharacters, it is instructive to introduce a few of the more useful ones. The expression \d refers to any digit (0 - 9), \D to any non-digit, and \w to any word character (A-Z, a-z, 0-9, and _). The three metacharacters, *, +, and ? are particularly useful because they add quantity to a regular expression. For example, the * matches the preceding subexpression zero or more times; the + matches the previous subexpression one or more times, and the ? matches the previous expression zero or one times. So, here are a few examples using these characters:

PRXPARSE("/\d\d\d/") matches any three digits in a row

PRXPARSE("/\d+/") matches one or more digits

PRXPARSE("/\w\w\w* /") matches any word with two or more characters followed by a space

PRXPARSE("/\w\w? +/") matches one or two word characters such as x, xy, or _X followed by one or more spaces

PRXPARSE("/(\w\w) +(\d) +/") matches two word characters, followed by one or more spaces, followed by a single digit, followed by one or more spaces. Note that the expression for the two word characters (\w\w) is placed in parentheses. Using the parentheses in this way creates what is called a capture buffer. The second set of parentheses (around the \d) represent the second capture buffer. Several of the Perl regular expression functions can make use of these capture buffers to extract and/or replace specific portions of a string. For example, the location of the two word characters or the single digit can be obtained using the PRXPOSN function.

Remember that the quotes are needed by the PRXPARSE function and the outer slashes are used to delimit the regular expression. Since the backslash, forward slash, parentheses and several other characters have special meaning in a regular expression, you may wonder, how do you search a string for a \ character or a left or right parenthesis? You do this by preceding any of these special characters with a backslash character (in Perl jargon called an escape character). So, to match a \ in a string, you code two backslashes like this: \\. To match an open parenthesis, you use \(.

The table below describes several of the wild cards and metecharacters used with regular expressions:

Metacharacter	Description	Examples
*	Matches the previous subexpression zero or more times	cat* matches "cat", "cats", "catanddog" c(at)* matches "c", "cat", and "catatat"
+	Matches the previous subexpression one or more times	\d+ matches one or more digits
?	Matches the previous subexpression zero or one times	hello? matches "hell" and "hello"
. (period)	Matches exactly one character	r.n matches "ron", "run", and "ran"
\d	Matches a digit 0 to 9	\d\d\d matches any three digit number
\D	Matches a non-digit	\D\D matches "xx", "ab" and "%%"
^	Matches the beginning of the string	^cat matches "cat" and "cats" but not "the cat"
$	Matches the end of a string	cat$ matches "the cat" but not "cat in the hat"
[xyz]	Matches any one of the characters in the square brackets	ca[tr] matches "cat" and "car"
[a-e]	Matches the letters a to e	[a-e]\D+ matches "adam", "edam" and "car"
[a-eA-E]	Matches the letter a to e or A to E	[a-eA-E]\w+ matches "Adam", "edam" and "B13"
{n}	Matches the previous subexpression n times	\d{5} matches any 5-digit number and is equivalent to \d\d\d\d\d
{n,}	Matches the previous subexpression n or more times	\w{3,} matches "cat" "_NULL_" and is equivalent to \w\w\w+
{n,m}	Matches the previous subexpression n or more times, but no more than m	\w{3,5} matches "abc" "abcd" and "abcde"
[^abcxyz]	Matches any characters except abcxyz	[^8]\d\d matches "123" and "999" but not "800"
x|y	Matches x or y	c(a|o)t matches "cat" and "cot"
\s	Matches a white space character, including a space or a tab,	\d+\s+\d+ matches one or more digits followed by one or more spaces, followed by one or more digits such as "123•••4" Note: •=space
\w	Matches any word character (upper- and lowercase letters, blank and underscore)	\w\w\w matches any three word characters

\(	Matches the character (	\(\d\d\d\) matches three digits in parentheses such as "(123)"
\)	Matches the character)	\(\d\d\d\) matches three digits in parentheses such as "(123)"
\\	Matches the character \	\D•\\•|D matches "the \ character" Note: •=space
\1	Matches the previous capture buffer and is called a back reference.	(\d\D\d)\1 matches "9a99a9" but not "9a97b7" (.)\1 matches any two repeated characters

This is not a complete list of Perl metacharacters, but it's enough to get you started. The Version 9 Online Doc or any book on Perl programming will provide you with more details. Examples of each of the PRX functions in this tutorial will also help you understand how to write these expressions.

Function used to define a regular expression

Function: PRXPARSE

Purpose: To define a Perl regular expression to be used later by the other Perl regular expression functions.

Syntax: **PRXPARSE(Perl-regular-expression)**

Perl-regular-expression is a Perl regular expression. Please see examples in the tutorial and in the sample programs in this chapter. The PRXPARSE function is usually executed only once in a DATA step and the return value is retained.

The forward slash "/" is the default delimiter. However, you may use any non-alphanumeric character instead of "/". Matching brackets can also be used as delimiters. Look at the last few examples below to see how other delimiters may be used.

If you want the search to be case-insensitive, you can follow the final delimiter with an "i". For example, PRXPARSE("/cat/I") will match Cat, CAT, or cat (see example 4 below).

Examples:

Function	Matches	Does not Match
PRXPARSE("/cat/")	"The cat is black"	"cots"
PRXPARSE("/^cat/")	"cat on the roof"	"The cat"
PRXPARSE("/cat$/")	"There is a cat"	"cat in the house"
PRXPARSE("/cat/i")	"The CaT"	"no dogs allowed"
PRXPARSE("/r[aeiou]t/")	"rat", "rot, "rut	"rt" and "rxt"
PRXPARSE("/\d\d\d /")	"345" and "999" (three digits followed by a space)	"1234" and "99"
PRXPARSE("/\d\d\d?/")	"123" and "12" (any two or three digits)	"1", "1AB", "1 9"
PRXPARSE("/\d\d\d+/")	"123" and "12345" (three or more digits)	"12X"
PRXPARSE("/\d\d\d*/")	"123", "12", "12345" (two or more digits)	"1" and "xyz"
PRXPARSE("/r.n/")	"ron", "ronny", "r9n", "r n"	"rn"
PRXPARSE("/[1-5]\d[6-9]/")	"299", "106", "337"	"666", "919", "11"
PRXPARSE("/(\d|x)\d/")	"56" and "x9"	"9x" and "xx"

PRXPARSE("/[^a-e]\D/")	"fX", "9 ", "AA"	"aa", "99", "b%"
PRXPARSE("/^\/\//")	"//sysin dd *"	"the // is here"
PRXPARSE("/^\/(\/|*)/")	a "//" or "/*" in cols 1 and 2	"123 /*"
PRXPARSE("#//#")	"//"	"/*"
PRXPARSE("/\/\//")	"//" (equivalent to previous expression)	"/*"
PRXPARSE("[\d\d]")	any two digits	"ab"
PRXPARSE("<cat>")	"the cat is black"	"cots"

Functions to locate text patterns

- PRXMATCH
- PRXSUBSTR (call routine)
- PRXPOSN (call routine)
- PRXNEXT (call routine)
- PRXPAREN

Function: PRXMATCH

Purpose: To locate the position in a string, where a regular expression match is found. This function returns the first position in a string expression of the pattern described by the regular expression. If this pattern is not found, the function returns a zero.

Syntax: **PRXMATCH(pattern-id *or* regular-expression, string)**

pattern-id is the value returned from the PRXPARSE function

regular-expression is a Perl regular expression, placed in quotation marks (version 9.1 and higher).

string is a character variable or a string literal.

Examples:

Regular Expression	String	Returns	Does not match (returns 0)
/cat/	"The cat is black"	5	"cots"
/^cat/	"cat on the roof"	1	"The cat"
/cat$/	"There is a cat"	12	"cat in the house"
/cat/I	"The CaT"	5	"no dogs allowed"
/r[aeiou]t/	"rat", "rot, "rut	1	"rt" and "rxt"
/\d\d\d /	"345" and "999"	1	"1234" and "99"
/\d\d\d?/	"123" and "12"	1	"1", "1AB", "1 9"
/\d\d\d+/	"123" and "12345"	1	"12"
/\d\d\d*/	"123", "12", "12345"	1	"1" and "xyz"
/r.n/	"ron", "ronny", "r9n", "r n"	1	"rn"
/[1-5]\d[6-9]/	"299", "106", "337"	1	"666", "919", "11"

/(\d|x)\d/	"56" and "x9"	1	"9x" and "xx"
/[^a-e]\D/	"fX", "9 ", "AA"	1	"aa", "99", "b%"
/^\/\//	"//sysin dd *"	1	"the // is here"
/^\/(\/|*)/	a "//" or "/*" in cols 1 and 2	1	"123 /*"

Examples of PRXMATCH without using PRXPARSE: STRING = "The cat in the hat"

Function	Result
PRXMATCH("/cat/",STRING)	4
PRXMATCH("/\d\d+/","AB123")	3

Program 2: Using a regular expression to search for phone numbers in a string

***Primary functions: PRXPARSE, PRXMATCH;

```
DATA PHONE;
  IF _N_ = 1 THEN PATTERN = PRXPARSE("/\(\d\d\d\) ?\d\d\d-\d{4}/");
  ***Regular expression will match any phone number in the form:
    (nnn)nnn-nnnn or (nnn) nnn-nnnn.;
  /*
    \(      matches a left parenthesis
    \d\d\d  matches any three digits
    (blank)? matches zero or one blank
    \d\d\d  matches any three digits
    -       matches a dash
    \d{4}   matches any four digits
  */
  RETAIN PATTERN;

  INPUT STRING $CHAR40.;
  IF PRXMATCH(PATTERN,STRING) GT 0 THEN OUTPUT;
DATALINES;
One number (123)333-4444
Two here:(800)234-2222 and (908) 444-2344
None here
;
PROC PRINT DATA=PHONE NOOBS;
  TITLE "Listing of Data Set Phone";
RUN;
```

Explanation:

To search for an open parenthesis, you use a '\('. The three \d's specify any three digits. The closed parenthesis is written as '\)'. The space followed by the '?' means zero or one space. This is followed by any three digits and a dash. Following the dash are any four digits. The notation: \d{4} is a short way of writing \d\d\d\d. The number in the braces indicates how may times to repeat the previous subexpression. Since you only execute the PRXPARSE function once, remember to RETAIN the value returned by the function.

Since the PRXMATCH function returns the first position of a match, any line containing one or more valid phone numbers will return a value greater than zero. Output from PROC PRINT is shown next:

```
Listing of Data Set Phone

RETURN    STRING

   1      One number (123)333-4444
   1      Two here:(800)234-2222 and (908) 444-234
```

Program 3: Modifying Program 2 to search for toll-free phone numbers

***Primary functions: PRXPARSE, PRXMATCH

***Other function: MISSING;

```
DATA TOLL_FREE;
  IF _N_ = 1 THEN DO
    RE = PRXPARSE("/\(8(00|77|87)\) ?\d\d\d-\d{4}\b/");
    ***Regular expression looks for phone numbers of the form:
      (nnn)nnn-nnnn or (nnn) nnn-nnnn.  In addition the first
      digit of the area code must be an 8 and the next two
      digits must be either a 00, 77, or 87.;
    IF MISSING(RE) THEN DO;
      PUT "ERROR IN COMPILING REGULAR EXPRESSION";
      STOP;
    END;
  END;
  RETAIN RE;
  INPUT STRING $CHAR80.;
  POSITION = PRXMATCH(RE,STRING);
  IF POSITION GT 0 THEN OUTPUT;
DATALINES;
One number on this line (877)234-8765
No numbers here
One toll free, one not:(908)782-6354 and (800)876-3333 xxx
Two toll free:(800)282-3454 and (887) 858-1234
No toll free here (609)848-9999 and (908) 345-2222
;
```

```
PROC PRINT DATA=TOLL_FREE NOOBS;
  TITLE "Listing of Data Set TOLL_FREE";
RUN;
```

Explanation:

Several things have been added to this program compared to the previous one. First, the regular expression now searches for numbers that begin with either (800), (877), or (887). This is accomplished by placing an "8" in the first position and then using the or operator (the |) to select either 00, 77, or 87 as the next two digits. One other difference between this expression and the one used in the previous program is that the number is followed by a word boundary (a space or end-of-line: \b). Hopefully, you're starting to see the impressive power of regular expressions by now. The MISSING function tests if its argument is missing or not. If you have an invalid regular expression, the value of RE will be missing and the MISSING function will return a true value. See the listing of data set TOLL_FREE below:

```
Listing of Data Set TOLL_FREE

RE  STRING                                                    POSITION

 1  One number on this line (877)234-8765                        25
 1  One toll free, one not:(908)782-6354 and (800)876-3333 xxx     42
 1  Two toll free:(800)282-3454 and (887) 858-1234
15
```

Program 4: Using PRXMATCH without PRXPARSE (entering the regular expression directly in the function)

```
***Primary functions: PRXMATCH;

DATA MATCH_IT;
  INPUT @1 STRING $20.;
  POSITION = PRXMATCH("/\d\d\d/",STRING);
DATALINES;
LINE 345 IS HERE
NONE HERE
ABC1234567
;
PROC PRINT DATA=MATCH_IT NOOBS;
  TITLE "Listing of Data Set MATCH_IT";
RUN;
```

Explanation:

In this program, the regular expression to search for three digits is placed directly in the PRXMATCH function instead of a return code from PRXPARSE. The output is:

```
Listing of Data Set MATCH_IT

STRING              POSITION
```

```
LINE 345 IS HERE        6
NONE HERE               0
ABC1234567              4
```

Function: CALL PRXSUBSTR

Purpose: Used with the PRXPARSE function to locate the starting position and length of a pattern within a string. The PRXSUBSTR call routine serves much the same purpose as the PRXMATCH function plus it returns the length of the match as well as the starting position.

Syntax: **CALL PRXSUBSTR(pattern-id, string, start, <length>)**

pattern-id is the return code from the PRXPARSE function

string is the string to be searched

start is the name of the variable that is assigned the starting position of the pattern

length is the name of a variable, if specified, that is assigned the length of the substring. If no substring is found, the value of length is zero.

Note: It is especially useful to use the SUBSTRN function (instead of the SUBSTR function) following the call to PRXSUBSTR, since a zero length as an argument to the SUBSTRN function results in a character missing value.

Examples: For these examples, PATTERN = PRXPARSE("/\d+/");

Function	STRING	START	LENGTH
CALL PRXSUBSTR(PATTERN,STRING,START,LENGTH)	"ABC 1234 XYZ"	5	4
CALL PRXSUBSTR(PATTERN,STRING,START,LENGTH)	"NO NUMBERS HERE"	0	0
CALL PRXSUBSTR(PATTERN,STRING,START,LENGTH)	"123 456 789"	1	3

Program 5: Locating all 5- or 9-digit zip codes in a list of addresses

Here is an interesting problem that shows the power of regular expressions. You have a list of mailing addresses. Some addresses have three lines, some four, others, possibly more or less. Some of the addresses have zip codes, some do not. The zip codes are either 5- or 9-digits in length. Go through the file and extract all the valid zip codes.

***Primary functions: PRXPARSE and PRXSUBSTR

***Other functions: SUBSTRN;

```
DATA ZIPCODE;
  IF _N_ = 1 THEN RE = PRXPARSE("/ \d{5}(-\d{4})?/");
  RETAIN RE;
```

```
  /*
    Match a blank followed by 5 digits followed by
    either nothing or a dash and 4 digits

    \d{5}     matches 5 digits
    -         matches a dash
    \d{4}     matches 4 digits
    ?         matches zero of one of the preceding subexpression

  */

  INPUT STRING $80.;
  LENGTH ZIP_CODE $ 10;
  CALL PRXSUBSTR(RE,STRING,START,LENGTH);
  IF START GT 0 THEN DO;
    ZIP_CODE = SUBSTRN(STRING,START + 1,LENGTH - 1);
    OUTPUT;
  END;
  KEEP ZIP_CODE;
DATALINES;
John Smith
12 Broad Street
Flemington, NJ 08822
Philip Judson
Apt #1, Building 7
777 Route 730
Kerrville, TX 78028
Dr. Roger Alan
44 Commonwealth Ave.
Boston, MA 02116-7364
;
PROC PRINT DATA=ZIPCODE NOOBS;
  TITLE "Listing of Data Set ZIPCODE";
RUN;
```

Explanation:

The regular expression is looking for a blank, followed by five digits, followed by zero or one occurrences of a dash and four digits. The PRXSUBSTR call routine checks if this pattern is found in the string. If it is found (START greater than zero), the SUBSTRN function extracts the zip code from the string. Note that the starting position in this function is START + 1 since the pattern starts with a blank. Also, since the SUBSTRN function is conditionally executed only when START is greater than zero, the SUBSTR function would be equivalent. A listing of the zip codes is shown below:

```
Listing of Data Set ZIPCODE
```

```
ZIP_CODE

08822
78028
02116-7364
```

Program 6: Extracting a phone number from a text string

***Primary functions: PRXPARSE, PRXSUBSTR

***Other functions: SUBSTR, COMPRESS, and MISSING;

```
DATA EXTRACT;
  IF _N_ = 1 THEN DO;
    PATTERN = PRXPARSE("/\(\d\d\d\) ?\d\d\d-\d{4}/");
    IF MISSING(PATTERN) THEN DO;
      PUT "ERROR IN COMPILING REGULAR EXPRESSION";
      STOP;
    END;
  END;
  RETAIN PATTERN;

  LENGTH NUMBER $ 15;
  INPUT STRING $CHAR80.;
  CALL PRXSUBSTR(PATTERN,STRING,START,LENGTH);
    IF START GT 0 THEN DO;
    NUMBER = SUBSTRTRING,START,LENGTH);
(S     NUMBER = COMPRESS(NUMBER," ");
    OUTPUT;
  END;
  KEEP NUMBER;
DATALINES;
THIS LINE DOES NOT HAVE ANY PHONE NUMBERS ON IT
THIS LINE DOES: (123)345-4567 LA DI LA DI LA
ALSO VALID (123) 999-9999
TWO NUMBERS HERE (333)444-5555 AND (800)123-4567
;
PROC PRINT DATA=EXTRACT NOOBS;
  TITLE "Extracted Phone Numbers";
RUN;
```

Explanation:

This program is similar to Program 2. You use the same regular expression to test if a phone number has been found, using the PRXMATCH function. The call to PRXSUBSTR gives you the starting position of the phone number and its length (remember, the length can vary because of the possibility of a space following the area code). The values obtained from the PRXSUBSTR function are then used in the SUBSTR function to extract the actual number from the text string. Finally, the COMPRESS function removes any blanks from the string. See the listing below for the results.

```
Extracted Phone Numbers

   NUMBER


(123)345-4567
(123)999-9999
(333)444-5555
```

Function: CALL PRXPOSN

Purpose: To return the position and length for a capture buffer (a subexpression defined in the regular expression). Used in conjunction with the PRXPARSE and one of the PRX search functions (such as PRXMATCH).

Syntax: **CALL PRXPOSN(pattern-id, capture-buffer-number, start, <length>)**

pattern-id is the return value from the PRXPARSE function

capture-buffer-number is a number indicating which capture buffer is to be evaluated

start is the name of the variable that is assigned the value of the first position in the string where the pattern from the nth capture buffer is found

length is the name of the variable, if specified, that is assigned the length of the found pattern

Before we get to the examples and sample programs, let's spend a moment discussing capture-buffers. When you write a Perl regular expression, you can make pattern groupings using parentheses. For example, the pattern: /(\d+) *([a-zA-Z]+)/ has two capture buffers. The first matches one or more digits; the second matches one or more upper- or lowercase letters. These two patterns are separated by zero or more blanks.

Examples: For these examples the following lines of SAS code were submitted:

```
PATTERN = PRXPARSE("/(\d+) *([a-zA-Z]+)/");
MATCH = PRXMATCH(PATTERN,STRING);
```

Function	STRING	START	LENGTH
CALL PRXPOSN(PATTERN,1,START,LENGTH)	"abc123 xyz4567"	4	3
CALL PRXPOSN(PATTERN,2,START,LENGTH)	"abc123 xyz4567"	8	3
CALL PRXPOSN(PATTERN,1,START,LENGTH)	"XXXYYYZZZ"	0	0

Program 7: Using the PRXPOSN function to extract the area code and exchange from a phone number

```
***Primary functions: PRXPARSE, PRXMATCH, PRXPOSN
***Other functions: SUBSTR;
RUN;
DATA PIECES;
  IF _N_ THEN RE = PRXPARSE("/\((\d\d\d)\) ?(\d\d\d)-\d{4}/");
  /*
    \(      matches an open parenthesis
    \d\d\d  matches three digits
    \)      matches a closed parenthesis
    b?      matches zero or more blanks (b = blank)
    \d\d\d  matches three digits
    -       matches a dash
    \d{4}   matches four digits
  */
  RETAIN RE;

  INPUT NUMBER $CHAR80.;
  MATCH = PRXMATCH(RE,NUMBER);
  IF MATCH GT 0 THEN DO;
    CALL PRXPOSN(RE,1,AREA_START);
    CALL PRXPOSN(RE,2,EX_START,EX_LENGTH);
    AREA_CODE = SUBSTR(NUMBER,AREA_START,3);
    EXCHANGE = SUBSTR(NUMBER,EX_START,EX_LENGTH);
  END;
  DROP RE;
DATALINES;
THIS LINE DOES NOT HAVE ANY PHONE NUMBERS ON IT
THIS LINE DOES: (123)345-4567 LA DI LA DI LA
ALSO VALID (609) 999-9999
TWO NUMBERS HERE (333)444-5555 AND (800)123-4567
;
PROC PRINT DATA=PIECES NOOBS HEADING=H;
  TITLE "Listing of Data Set PIECES";
RUN;
```

Explanation:

The regular expression in this program looks similar to the one in Program 2. Here we have added a set of parentheses around the expression that identifies the area code '\d\d\d' and the expression that identifies the exchange '\d\d\d'. The PRXMATCH function returns a one if the match is successful, zero otherwise. The first call to PRXPOSN requests the position of the start of the first capture buffer. Since we know that the length is three, we do not need to include a length argument in the calling sequence. The next function call asks for the starting position and length (although we don't need the length) of the position of the exchange (the first three digits following the area code or the blank after the area code. The SUBSTR function then extracts the substrings. A listing of the resulting data set is shown below:

```
Listing of Data Set PIECES
```

```
NUMBER

THIS LINE DOES NOT HAVE ANY PHONE NUMBERS ON IT
THIS LINE DOES: (123)345-4567 LA DI LA DI LA
ALSO VALID (609) 999-9999
TWO NUMBERS HERE (333)444-5555 AND (800)123-4567

          AREA_                                AREA_
MATCH     START     EX_START     EX_LENGTH     CODE      EXCHANGE

    0         .          .            .
   17        18         22            3          123        345
   12        13         18            3          609        999
   18        19         23            3          333        444
```

Program 8: Using regular expressions to read very unstructured data

***Primary functions: PRSPARSE, PRXMATCH, PRXPOSN
***Other functions: SUBSTR, INPUT;

***This program will read every line of data and, for any line
that contains two or more numbers, will assign the first
number to X and the second number to Y;

```
DATA READ_NUM;
***Read the first number and second numbers on line;
  IF _N_ = 1 THEN RET = PRXPARSE("/(\d+) +\D*(\d+)/");
  /*
    \d+    matches one or more digits
    b+     matches one or more blanks (b = blank)
    \D*    matches zero or more non-digits
    \d+    matches one or more digits
  */
  RETAIN RET;

  INPUT STRING $CHAR40.;
  POS = PRXMATCH(RET,STRING);
  IF POS GT 0 THEN DO;
    CALL PRXPOSN(RET,1,START1,LENGTH1);
    IF START1 GT 0 THEN X = INPUT(SUBSTR(STRING,START1,LENGTH1),9.);
    CALL PRXPOSN(RET,2,START2,LENGTH2);
    IF START2 GT 0 THEN Y = INPUT(SUBSTR(STRING,START2,LENGTH2),9.);
    OUTPUT;
```

```
  END;
  KEEP STRING X Y;
DATALINES;
XXXXXXXXXXXXXXXXXX 9 XXXXXXX       123
This line has a 6 and a 123 in it
456 789
None on this line
Only one here: 77
;
PROC PRINT DATA=READ_NUM NOOBS;
  TITLE "Listing of Data Set READ_NUM";
RUN;
```

Explanation:

This example shows how powerful regular expressions can be used to read very unstructured data. Here the task was to read every line of data and to locate any line with two or more numbers on it, then, assign the first value to X and the second value to Y. (See the program example under the PRXNEXT function for a more general solution to this problem.) The listing of READ_NUM, below, shows that this program worked as desired. The variable STRING was kept in the data set so you could see the original data and the extracted numbers in the listing.

```
Listing of Data Set READ_NUM

STRING                                        X     Y

XXXXXXXXXXXXXXXXXX 9 XXXXXXX         123      9    123
This line has a 6 and a 123 in it              6    123
456 789                                      456    789
```

Function: CALL PRXNEXT

Purpose: Locates the nth occurrence of a pattern defined by the PRXPARSE function in a string. Each time you call the PRXNEXT routine, the next occurrence of the pattern will be identified.

Syntax: **CALL PRXNEXT(pattern-id, start, stop, position, length)**

pattern-id is the value returned by the PRXPARSE function

start is the starting position to being the search

stop is the last position in the string for the search. If stop is set to –1, the position of the last non-blank character in string is used.

position is the name of the variable that is assigned the starting position of the nth occurrence of the pattern or the first occurrence after start

length is the name of the variable that is assigned the length of the pattern

Examples: For these examples the following statements were issued:

```
RE = PRXPARSE("/\d+/");
***Look for 1 or more digits;
STRING = "12 345 ab 6 cd";
START = 1;
STOP = LENGTH(STRING);
```

Function	Returns (1st call)	Returns (2nd call)	Returns (3rd call)
CALL PRXNEXT(RE, START, STOP, POS, LENGTH)	START = 3	START = 7	START = 12
	STOP = 14	STOP = 14	STOP = 14
	POS = 1	POS = 4	POS = 11
	LENGTH = 2	LENGTH = 3	LENGTH = 1

Program 9: Finding digits in random positions in an input string using CALL PRXNEXT

```
***Primary functions: PRXPARSE, PRXNEXT;

DATA FIND_NUM;
  IF _N_ = 1 THEN RET = PRXPARSE("/\d+/");
  *Look for one or more digits in a row;
  RETAIN RET;

  INPUT STRING $40.;
  START = 1;
  STOP = LENGTH(STRING);
  CALL PRXNEXT(RET,START,STOP,STRING,POSITION,LENGTH);
  ARRAY X[5];
  DO I = 1 TO 5 WHILE (POSITION GT 0);
    X[I] = INPUT(SUBSTR(STRING,POSITION,LENGTH),9.);
    CALL PRXNEXT(RET,START,STOP,STRING,POSITION,LENGTH);
  END;
  KEEP X1-X5 STRING;
DATALINES;
THIS 45 LINE 98 HAS 3 NUMBERS
NONE HERE
12 34 78 90
;
PROC PRINT DATA=FIND_NUM NOOBS;
```

```
  TITLE "Listing of Data Set FIND_NUM";
RUN;
```

Explanation:

The regular expression /\d+/ says to look for one or more digits in a string. The initial value of START is set to one and STOP to the length of the string (not counting trailing blanks). The PRXNEXT function is called and the value of START is set to the position of the blank after the first number and STOP to the length of the string. POSITION is the starting position of the first digit and LENGTH is the number of digits in the number. The SUBSTR function extracts the digits and the INPUT function does the character to numeric conversion. This continues until no more digits are found (POSITION = 0). See the listing below to confirm that the program worked as expected.

```
Listing of Data Set FIND_NUM

STRING                          X1    X2    X3    X4    X5

THIS 45 LINE 98 HAS 3 NUMBERS   45    98     3     .     .
NONE HERE                        .     .     .     .     .
12 34 78 90                                  12    34    78    90     .
```

Function: PRXPAREN

Purpose: When a Perl regular expression contains several alternative matches, this function returns a value indicating the largest capture-buffer number that found a match. You may want to use this function with the PRXPOSN function.

This function is used in conjunction with PRXPARSE and PRXMATCH.

Syntax: **PRXPAREN(pattern-id)**

pattern-id is the value returned by the PRXPARSE function

Examples: For this example, RETURN = PRXPARSE(/"(one)|(two)|(three)/") and POSITION = PRXMATCH(RETURN,STRING);

Function	STRING	Returns
PRXPAREN(RETURN)	"three two one"	3
PRXPAREN(RETURN)	"only one here:	1
PRXPAREN(RETURN)	"two one three"	2

Program 10: Demonstrating the PRXPAREN function

***Primary functions: PRXPARSE, PRXMATCH, PRXPAREN;

```
DATA PAREN;
```

```
  IF _N_ = 1 THEN PATTERN = PRXPARSE("/(\d )|(\d\d )|(\d\d\d )/");
  ***One or two or three digit number followed by a blank;
  RETAIN PATTERN;

  INPUT STRING $CHAR30.;
  POSITION = PRXMATCH(PATTERN,STRING);
  IF POSITION GT 0 THEN WHICH_PAREN = PRXPAREN(PATTERN);
DATALINES;
one single digit 8 here
two 888 77
12345 1234 123 12 1
;
PROC PRINT DATA=PAREN NOOBS;
  TITLE "Listing of Data Set PAREN";
RUN;
```

Explanation:

The Perl regular expression in the PRXPARSE function matches a one-, two-, or three-digit number followed by a blank. In the first observation, the single digit matches the first capture buffer so the PRXPAREN function returns a 1. In the second observation, the 888 is matched by the third capture buffer, and a 3 is returned. Finally, the first match in the third string is the three digit number which, again, is matched by capture buffer 3.

```
Listing of Data Set PAREN

                                                  WHICH_
PATTERN   STRING                       POSITION    PAREN

   1      one single digit 8 here         18         1
   1      two 888 77                       5         3
    1       12345 1234 123 12 1                 3            3
```

Function to substitute one string for another

- PRXCHANGE (call routine)

Function: CALL PRXCHANGE

Purpose: To substitute one string for another. One advantage of using PRXCHANGE over TRANWRD is that you can search for strings using wild cards. Note that you need to use the "s" operator in the regular expression to specify the search and replacement expression (see the explanation following the program).

Syntax: **CALL PRXCHANGE(pattern-id *or* regular-expression, times, old-string <, new-string <, result-length <, truncation-value <, number-of-changes>>>>);**

pattern-id is the value returned from the PRXPARSE function

regular-expression is a Perl regular expression, placed in quotation marks (version 9.1 and higher).

times is the number of times to search for and replace a string. A value of –1 will replace all matching patterns

old-string is the string that you want to replace. If you do not specify a new-string, the replacement will take place in old-string.

new-string if specified, names the variable to hold the text after replacement. If new-string is not specified, the changes are made to old-string.

result-length the name of the variable that, if specified, that is assigned a value representing the length of the string after replacement. Note that trailing blanks in old-string are not copied to new-string.

truncation-value the name of the variable, if specified, that is assigned a value of 0 or 1. If the resulting string is longer than the length of new-string, the value is 1, otherwise it is a 0. This value is useful to test if your string was truncated because the replacements resulted in a length longer than the original specified length.

number-of-changes the name of the variable, if specified, that is assigned a value representing the total number of replacements that were made.

'

Program 11: Demonstrating the PRXCHANGE function

```
***Primary functions: PRXPARSE, PRXCHANGE;

DATA CAT_AND_MOUSE;
  INPUT TEXT $CHAR40.;
  LENGTH NEW_TEXT $ 80;

  IF _N_ = 1 THEN MATCH = PRXPARSE("s/[Cc]at/Mouse/");
  *Replace "Cat" or "cat" with Mouse;
  RETAIN MATCH;

  CALL PRXCHANGE(MATCH,-
1,TEXT,NEW_TEXT,R_LENGTH,TRUNC,N_OF_CHANGES);
  IF TRUNC THEN PUT "Note: NEW_TEXT was truncated";
DATALINES;
The Cat in the hat
There are two cat cats in this line
;
PROC PRINT DATA=CAT_AND_MOUSE NOOBS;
  TITLE "Listing of CAT_AND_MOUSE";
RUN;
```

Explanation:

The regular expression and the replacement string is specified in the PRXPARSE function, using the substitution operator (the "s" before the first /. In this example, the regular expression to be searched for is /[Cc]at/. This matches either "Cat" or "cat" anywhere in the string. The replacement string is "Mouse." Since the length of NEW_TEXT was set to 80, even though the replacement of cat (or Cat) with Mouse results in a longer string, the new length does not exceed 80 so no truncation occurs. The –1 indicates that you want to replace every occurrence of "Cat" or "cat" with "Mouse." If you did not supply a NEW_TEXT variable, the replacement would be made to the original string. The output from PROC PRINT is shown below:

```
Listing of CAT_AND_MOUSE
TEXT


The Cat in the hat
There are two cat cats in this line

                                                                N_OF_
NEW_TEXT                                  MATCH   R_LENGTH   TRUNC   CHANGES

The Mouse in the hat                        1        42        0        1
There are two Mouse Mouses in this line     1        44        0        2
```

Program 12: Demonstrating the use of capture buffers with PRXCHANGE

```
***Primary functions: PRXPARSE, PRXCHANGE;

DATA CAPTURE;
  IF _N_ = 1 THEN RETURN = PRXPARSE("S/(\w+ +)(\w+)/$2 $1/");
  RETAIN RETURN;

  INPUT STRING $20.;
  CALL PRXCHANGE(RETURN,-1,STRING);
DATALINES;
Ron Cody
Russell Lynn
;
PROC PRINT DATA=CAPTURE NOOBS;
  TITLE "Listing of Data Set CAPTURE");
RUN;
```

Explanation:

The regular expression specifies one or more word characters, followed by one or more blanks (the first capture buffer), followed by one or more word characters (the second capture buffer). In the substitute portion of the regular expression, the $1 and $2 expressions refer to the first and second capture buffer respectively. So, by placing the $2 before the $1, the two words are reversed, as shown below.

```
Listing of Data Set CAPTURE)


RETURN     STRING
```

```
1      Cody Ron
 1       Lynn Russell
```

Acknowledgements

Contact Information

Ronald P. Cody, Ed.D.
89 Lazy Brook Road
Flemington, NJ 08822
cody@umdnj.edu