Chapter 2
Perl Regular Expressions

Introduction

Perl regular expressions were added in SAS®9. SAS regular expressions (similar to Perl regular expressions but using a different syntax to indicate text patterns) have actually been around since SAS 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) enable 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 enables 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 a large text file and want to extract all the e-mail addresses. Another example would be extracting ZIP codes from an address file. Once a pattern is found, you can obtain the position of the pattern, extract a substring, or substitute a string.

The first edition of this book described both the Perl and SAS regular expressions, but for several reasons the older SAS regular expression functions (all beginning with RX) were not included in the second edition. First, more people may already be familiar with Perl regular expressions. Second, the implementation of the Perl regular expression in SAS is said to be more efficient. Finally, there is a more extensive set of Perl regular expression functions compared to the older SAS regular expressions. The syntax and usage of these two sets of functions is different and it is confusing to use both. If you are new to regular expressions, I suggest that you stick to the Perl regular expressions exclusively.

I have not attempted to provide a complete description of Perl regular expressions in this book. Hopefully, it will be a good start, but to become an expert, you will need to obtain a book on Perl or some other documentation on regular expressions.

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 in the future.

The PRXPARSE function is used to create a regular expression. Because 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. If the first argument to the PRXPARSE function is a constant, SAS will not recompile the regular expression for each iteration of the DATA step. However, it is still considered good programming practice to use the RETAIN statement with _N_ when you use the PRXPARSE function. 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 in the appropriate section of this chapter following this tutorial.

Program 2.1: Using a Perl regular expression to locate lines with an exact text match

```
title "Perl Regular Expression Tutorial - Program 1";
data _null_;

   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 dog
cat in the beginning
at the end, a cat
cat
;
```

Explanation

You write your Perl regular expression as the argument of the PRXPARSE function. In this example, the regular expression is a constant so you need to place it in single or double quotation marks. You could, alternatively, assign the regular expression to a character variable and use that character variable as the argument of the PRXPARSE function.

The argument of the PRXPARSE function is a standard Perl regular expression. In this example, you are using the forward slashes (/) as delimiters for the regular expression. You can choose other characters as delimiters, as long as you start and end the expression with the same delimiter. The forward slash is most commonly used as a delimiter for regular expressions.

Each time you compile a regular expression, SAS assigns sequential numbers to the resulting expression. This pattern identifier returned by the PRXPARSE function is needed to identify a pattern when you perform searches using the other "PRX" functions such as PRXMATCH, PRXCHANGE, PRXNEXT, PRXSUBSTR, PRXPAREN, or PRXPOSN. Thus, the value of

PATTERN_NUM in this program is 1. If you executed another PRXPARSE function in the same DATA step, the function would return a 2.

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 pattern identifier from the PRXPARSE function and the string to be searched. You can think of the PRXMATCH function much as the FIND function; instead of looking for an exact match, you can also search for a text pattern. The PRXMATCH function returns the first position in String where "cat" is found. If there is no match, the PRXMATCH function returns a 0. Let's look at the following output.

```
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 dog 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. (If you invoked PRXPARSE a second time in this DATA step, the returned value would be 2.) In the second line of output, the value of POSITION is 0, since the word "cat" 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:

```
Perl Regular Expression Tutorial - Program 1
Pattern_num=1 String=there is a cat in this line. Position=11
Pattern_num=1 String=does not match dog 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 repetition characters, `*`, +, 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`) creates 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 starting location of the two word characters or the single digit can be obtained using the PRXPOSN function.

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 that contains characters such as \, (, or)? You do this by preceding any of these special characters with a \ 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 following table describes several of the wild cards and metacharacters used with regular expressions:

Metacharacter	Description	Examples
. (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 one of the letters in the range 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"
[^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, digits, 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 \	a\\b matches a\b
\.	Matches the character .	Mr\. matches Mr.

This is not a complete list of Perl metacharacters, but it's enough to get you started. The SAS Product Documentation at http://support.sas.com/documentation or any book on Perl programming will provide you with more details. Examples of each of the "PRX" functions in this chapter will also help you understand how to write these expressions.

Note: The wording of arguments in this book might differ from the wording of arguments in the SAS Product Documentation.

Regular expression syntax includes repetition operators. These operators are extremely powerful in that they enable you to specify that an expression is to be repeated a given number of times. For example, `\d` is any digit. If you follow this expression with an asterisk (`*`), you now have a specification for zero or more digits. Likewise, `\d+` is a specification for one or more digits. The following table describes these repetition operators:

Repetition Operators	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"
{n}	Matches the previous subexpression *n* times	\d{4} matches four digits
{n,m}	Matches the previous subexpression *n* or more times, but no more than *m*	\w{3,5} matches "abc" "abcd" and "abcde"

Function That Defines a Regular Expression

Function: PRXPARSE

Purpose: This function returns a pattern identifier that is used later by the other Perl regular expression functions or CALL routines. The identifier numbers start at 1, and each time you invoke this function, the identifier number is incremented by 1.

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

Perl-regular-expression is a Perl regular expression. See examples in the brief tutorial and in the sample programs in this chapter. If the *Perl-regular-expression* is a constant, the Perl regular expression is compiled only once. Successive calls to PRXPARSE will not cause a recompile, but will return the regular-expression-id for the regular expression that was already compiled. You may choose to execute the PRXPARSE function only once by using the code `if _n_ = 1 then . . .`, in which case you will need to RETAIN the returned value. (See the following examples.)

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

If you want the search to be case insensitive, you can follow the final delimiter with the letter `i`. For example, `PRXPARSE("/cat/i")` will match Cat, CAT, or cat (see the fourth example).

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", "rxt", and "riot"
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("/(\d|x)\d/")	"56" and "x9"	"9x" and "xx"
PRXPARSE("/[^a-e]\D/")	"fX", "9 ", "AA"	"aa", "99", "b%", "d9"
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"

See examples of the PRXPARSE function in all of the examples in this chapter.

Functions That Locate Text Patterns

Function: PRXMATCH

Purpose: To locate the position in a string where a regular expression match is found. PRXMATCH returns the starting position at which the pattern is found. If this pattern is not found, the function returns a 0. (This function is similar to the FIND function, but instead of matching a string of characters, it is matching a pattern described by a regular expression.)

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 (SAS 9.1 and later). Note: If you use *pattern-id*, you must use the PRXPARSE function first. If you place the regular expression (in quotation marks) directly in the PRXMATCH function, you do not use the PRXPARSE function.

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", "rxt", "root"
/\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"

(*continued*)

(*continued*)

Regular Expression	String	Returns	Does Not Match (Returns 0)
/[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	Returns
PRXMATCH("/cat/",STRING)	4
PRXMATCH("/\d\d+/","AB123")	3

Program 2.2: Using a regular expression to search for phone numbers in a string

```
data phone;
   input String $char40.;
   if _n_ = 1 then Pattern = prxparse("/\(\d\d\d\) ?\d\d\d-\d{4}/");
   retain Pattern;
   if prxmatch(Pattern,String) then output;
   ***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
   */
datalines;
One number (123)333-4444
Two here:(800)234-2222 and (908) 444-2344
None here
;
title "Listing of Data Set Phone";
proc print data=phone noobs;
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 many times to repeat the previous subexpression. Since you execute the PRXPARSE function only once, remember to use the RETAIN statement 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

String                                    Pattern

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

Program 2.3: Modifying Program 2.2 to search for toll-free phone numbers

```
***Primary functions: PRXPARSE, PRXMATCH;
data toll_free;
   if _n_ = 1 then
      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.;
   retain RE;
   input String $char80.;
   Position = prxmatch(RE,String);
   if Position 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
;
title "Listing of Data Set TOLL_FREE";
proc print data=toll_free noobs;
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. Here is the listing of data set TOLL_FREE:

```
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 2.4: Using PRXMATCH without PRXPARSE (entering the regular expression directly in the function)

```
***Primary function: PRXMATCH;

data match_it;
   input @1 String $20.;
   Position = prxmatch("/\d\d\d/",String);
datalines;
LINE 345 IS HERE
NONE HERE
ABC1234567
;
title "Listing of Data Set MATCH_IT";
proc print data=match_it noobs;
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. (Note that many of the other PRX functions and CALL routines do not allow the regular expression as an argument.) The output is:

```
                        Listing of Data Set MATCH_IT

String              Position

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

The last line of this output is a good illustration of what Jason Secosky (the SAS guru on regular expressions) calls "false positives." You are looking for three digits and not specifying what comes after the three digits. So, even though there are more digits following the string "123", the pattern still matches. If you did not want this string to be selected by the regular expression, you could add a word boundary (`\b`) at the end of the regular expression (`/\d\d\d\b/`). A word boundary is either a space or the beginning or end of a line. If you did this, the value of POSITION in the last observation would be 0.

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 using a zero length as an argument of the SUBSTRN function results in a character missing value instead of an error.

Examples

For these examples, `RE = PRXPARSE("/\d+/")`.

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

Program 2.5: 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 mailing list. Some addresses have three lines, some have four, and others have possibly more or less. Some of the addresses have ZIP codes, some do not. The ZIP codes are either five digits or nine digits, a dash, followed by four digits (ZIP + 4). Go through the file and extract all the valid ZIP codes.

```
***Primary functions: PRXPARSE and CALL PRXSUBSTR
***Other function: 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 or one of the preceding subexpression

   */

   input String $80.;
   length Zip_code $ 10;
   call prxsubstr(RE,String,Start,Length);
   if Start 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
;
title "Listing of Data Set ZIPCODE";
proc print data=zipcode noobs;
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. Here is a listing of the ZIP codes:

```
Listing of Data Set ZIPCODE

Zip_code

08822
78028
02116-7364
```

Program 2.6: Extracting a phone number from a text string

```
***Primary functions: PRXPARSE, CALL PRXSUBSTR
***Other functions: SUBSTR, COMPRESS;

data extract;
   if _n_ = 1 then
      Pattern = prxparse("/\(\d\d\d\) ?\d\d\d-\d{4}/");
   retain Pattern;

   length Number $ 15;
   input String $char80.;
   call prxsubstr(Pattern,String,Start,Length);
      if Start then do;
      Number = substr(String,Start,Length);
      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
;
title "Extracted Phone Numbers";
proc print data=extract noobs;
run;
```

Explanation

This program is similar to Program 2.2. 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 following listing for the results (notice that this program extracts only the first phone number on a line if there is more than one number on a line).

```
                        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 *n*th 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.

Example

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 2.7: Using the PRXPOSN CALL routine to extract the area code and exchange from a phone number

```
***Primary functions: PRXPARSE, PRXMATCH, CALL PRXPOSN
***Other function: SUBSTR;


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
      Blank?   matches zero or more blanks
      \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 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
;
title "Listing of Data Set PIECES";
proc print data=pieces noobs heading=h;
run;
```

Explanation

The regular expression in this program looks similar to the one in Program 2.2. 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 the starting position if the match is successful, 0 otherwise. The first call to PRXPOSN requests the position of the start of the first capture buffer. Since we know that the length is 3, 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

exchange (the first three digits following the area code or the blank after the area code). Note that we don't need the length of the exchange either, but it was included to illustrate how lengths are obtained. The SUBSTR function then extracts the substrings. Here is a listing of the resulting data set:

```
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 2.8: Using regular expressions to read very unstructured data

```
***Primary functions: PRSPARSE, PRXMATCH, CALL 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
      blank+  matches one or more blanks
      \D*     matches zero or more non-digits
      \d+     matches one or more digits
   */
   retain ret;
```

```
   input String $char40.;
   Pos = prxmatch(ret,String);
   if Pos then do;
      call prxposn(ret,1,Start1,Length1);
      if Start1 then X = input(substr(String,Start1,Length1),9.);
      call prxposn(ret,2,Start2,Length2);
      if Start2 then Y = input(substr(String,Start2,Length2),9.);
      output;
   end;
   keep String X Y;
datalines;
XXXXXXXXXXXXXXXXX 9 XXXXXXX         123
This line has a 6 and a 123 in it
456 789
None on this line
Only one here: 77
;
title "Listing of Data Set READ_NUM";
proc print data=read_num noobs;
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, and then to 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 INPUT function is used to perform a character-to-numeric conversion (see Chapter 10, "Special Functions," for more details on this function).

The following listing of READ_NUM 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

XXXXXXXXXXXXXXXXX 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: To locate the *n*th 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 begin 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 the string is used.

position is the name of the variable that is assigned the starting position of the *n*th 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 STOP = 14 POS = 1 LENGTH = 2	START = 7 STOP = 14 POS = 4 LENGTH = 3	START = 12 STOP = 14 POS = 11 LENGTH = 1

Program 2.9: Finding digits in random positions in an input string using CALL PRXNEXT

```
***Primary functions: PRXPARSE, CALL PRXNEXT
***Other functions: LENGTHN, INPUT;

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 = lengthn(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
;
title "Listing of Data Set FIND_NUM";
proc print data=find_num noobs;
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 1 and STOP to the length of the string (not counting trailing blanks). The PRXNEXT function is called, the value of START is set to the position of the blank after the first number, and the value of STOP is set 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 following listing 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: To return a value indicating the largest capture buffer number that found a match. Use PRXPAREN when a Perl regular expression contains several alternative matches. 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 2.10: Demonstrating the PRXPAREN function

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

/* Orders are identified by a numeric type: 1= Retail 2=Catalog
3=Internet
   Use the sale description to identify the type of sale and set
OrderType */

data paren;
   if _n_ = 1 then
   Pattern  = prxparse("/(Retail|Store)|(Catalog)|(Internet|Web)/i");
   ***look for order type in Description field;
   retain Pattern;
```

```
      input Description $char30.;
      Position = prxmatch(Pattern,Description);
      if Position then OrderType = prxparen(Pattern);
   datalines;
   Order placed on Internet
   Retail order
   Store 123: Retail purchase
   Spring catalog order
   Order from specialty catalog
   internet order
   Web order
   San Francisco store purchase
   ;
   title "Listing of Data Set PAREN";
   proc print data=paren noobs;
   run;
```

Explanation

Thanks to Jason Secosky at SAS for this example. Here you are inspecting a description to determine an order type. To make it more interesting, either Retail or Store orders are type 1, Catalog orders are type 2, and Internet or Web orders are type 3. Since each of these descriptions are placed in parentheses (without a \ in front of it) each of these descriptions defines a capture buffer—Retail or Store = 1, Catalog = 2, and Internet or Web = 3. The PRXPAREN function returns the value (1, 2, or 3) of the capture buffer that was matched.

Here is the output:

```
Listing of Data Set PAREN
                                                        Order
Pattern    Description                      Position     Type

   1       Order placed on Internet            17         3
   1       Retail order                         1         1
   1       Store 123: Retail purchase           1         1
   1       Spring catalog order                 8         2
   1       Order from specialty catalog        22         2
   1       internet order                       1         3
   1       Web order                            1         3
   1       San Francisco store purchase        15         1
```

Function That Substitutes One String for Another

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 substitution (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.

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

Program 2.11: Demonstrating the CALL PRXCHANGE function

```
***Primary functions: PRXPARSE, CALL 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
;
title "Listing of CAT_AND_MOUSE";
proc print data=cat_and_mouse noobs;
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. Therefore, 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. Here is the output from PROC PRINT:

```
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 2.12: Demonstrating the use of capture buffers with PRXCHANGE

```
***Primary functions: PRXPARSE, CALL 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
;
title "Listing of Data Set CAPTURE";
proc print data=capture noobs;
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 here.

```
Listing of Data Set CAPTURE

Return    String

   1      Cody Ron
   1      Lynn Russell
```

Function That Releases Memory Used by a Regular Expression

Function: CALL PRXFREE

Purpose: To free resources that were allocated to a Perl regular expression (usually used with a test for end-of-file). If you do not call PRXFREE, the resources used will be freed when the DATA step ends.

Syntax: **CALL PRXFREE(*pattern-id*)**

pattern-id is the value returned from the PRXPARSE function.

Examples

For this example, the statement `PATTERN = PRXPARSE("/\d/")` preceded the call to PRXFREE.

Function	Returns
`PRXFREE(PATTERN)`	Sets the value returned by PRXPARSE to missing

Program 2.13: Data cleaning example using PRXPARSE and CALL PRXFREE

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

data invalid;
   ***Valid ID's are 1 to 3 digits with possible leading blanks;
   infile 'c:\books\functions\idnums.dat' end=last;
   if _n_ = 1 then Valid = prxparse("/\d\d\d| \d\d|  \d/");
   /*
      \d\d\d              matches three digits
      (single blank)\d\d  matches a blank followed by two digits
      (two blanks)\d      matches two blanks followed by one digit
   */
```

```
    retain Valid;
    input @1 ID $char3.;
    Pos = prxmatch(Valid,ID);
    if Pos eq 0 then output invalid;
    if last then call prxfree(Valid);
    drop Valid;
run;
```

Explanation

In this example, valid IDs are character values of either three digits, a blank followed by two digits, or two blanks followed by one digit. The INFILE statement option END=LAST creates the logical variable (LAST), which is true when the last record is being read from the file. So, when LAST is true, a call is made to PRXFREE to release resources used by the regular expression. Note that the $CHAR3. informat is used to read the ID variable, since you want to maintain any leading blanks.