PharmaSUG China 2016 – Paper 98

Performing Pattern Matching by Using Perl Regular Expressions

Arthur Li, City of Hope National Medical Center, Duarte, CA

ABSTRACT

SAS® provides many DATA step functions to search and extract patterns from a character string, such as SUBSTR, SCAN, INDEX, TRANWRD, etc. Using these functions to perform pattern matching often requires utilizing many function calls to match a character position. However, using the Perl Regular Expression (PRX) functions or routines in the DATA step will improve pattern matching tasks by reducing the number of function calls and making the program easier to maintain. In this talk, in addition to learning the syntax of Perl Regular Expressions, many real-world applications will be demonstrated.

COMMON STEPS FOR USING PRX IN THE DATA STEP

A regular expression can be considered a separate type of language. It consists of a sequence of characters that is used to define a search pattern. In SAS, there are many functions and CALL routines that use a modified version of the Perl regular expression (PRX) to parse character strings. For the rest of the paper, PRX will be used instead of the full name.

The most common steps in using PRX in the DATA step consist of the following:

1. Defining a regular expression pattern and compile it by using the PRXPARSE function
2. Use a PRX function or CALL routine to search or substitute text strings by using the defined pattern
3. (optional) More character manipulations are performed based on the result from step 2

For example, Program 1 starts with defining a regular expression pattern (**`/Zack/`**) that is enclosed in the PRXPARSE function. In the second step, the PRXMATCH function is used to locate the defined pattern in the input address. If the pattern is found, the PRXMATCH function would return the position of the pattern. A more detailed discussion about both functions will be discussed later in this paper. In the third step, when the position is not equaling to 0 (the match is found), the address in which the pattern is found will be printed in the SAS log.

Program 1:

```
data _null_;
    /*step 1*/
    if _N_=1 then patternID=prxparse("/Zack/");
    retain patternID;
    input address $80.;
    /*step 2*/
    position = prxmatch(patternID, address);
    /*step 3*/
    if position ^= 0 then put address=;
    datalines;
Zack Johnson, 153 First Str, Chapel Hill, NC27514
Dan Zack, 67891 64th st, Brea, CA
Sally Johns, 4 Moritz Street, Duarte, CA 91010
;
```

SAS Log from Program1:

```
1376
1377  data _null_;
1378     if _N_=1 then patternID=prxparse("/Zack/");
1379     retain patternID;
1380     input address $80.;
1381     position = prxmatch(patternID, address);
1382     if position ^= 0 then put address=;
1383     datalines;

address=Zack Johnson, 153 First Str, Chapel Hill, NC27514
address=Dan Zack, 67891 64th st, Brea, CA
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds
```

DEFINING A PERL REGULAR EXPRESSION

In SAS, the PRX pattern is defined and enclosed with a pair of forward slashes (/). To compile a PRX that can be used for pattern matching, you need to use the PRXPARSE function:

regular-expression-id=**PRXPARSE**(*perl-regular-expression*)

The *regular-expression-id* is a numeric pattern identifier that is returned by the PRXPARSE function. This pattern identifier number is used by other PRX functions or CALL routines to match patterns. If an error occurs in parsing the regular expression, SAS returns a missing value.

COMPILING A REGULAR EXPRESSION

During the DATA step execution, the PRXPARSE function compiles the PRX and assigns a numeric pattern identifier to the *regular-expression-id* at each DATA step iteration. However, if we have a large number of observations to process, it will waste a lot of memory. A common programming practice is to generate the pattern identifier only once when processing the first observation. For example, Program 1 uses the IF statement to assign the pattern identifier to PATTERNID when _N_ equals 1.Since PATTERNID is used at every iteration of the DATA step, the RETAIN statement is then used to retain its value.

Alternatively, you can use the PRX *compile once* option by placing an "o" option immediately after the closing slash, just like in Program 2. Using the /o option will force SAS to compile the PRX pattern only once and create PATTERNID at the first iteration only. This approach will also imply your code by avoiding the use of the RETAIN statement.

Program 2:

```
data _null_;
    patternID=prxparse("/Zack/o");
    input address $80.;
    position = prxmatch(patternID, address);
    if position ^= 0 then put address=;
    datalines;
Zack Johnson, 153 First Str, Chapel Hill, NC27514
Dan Zack, 67891 64th st, Brea, CA
Sally Johns, 4 Moritz Street, Duarte, CA 91010
;
```

A TUTORIAL FOR CREATING THE PERL REGULAR EXPRESSION (PRX)

In Program 1, the PRX pattern is **/Zack/**, which consists of only letters. In this pattern, each character is going to match exactly one single character in the searched string. That is to say "Z" needs to match with "Z", "a" needs to match with "a", etc. Instead of matching one character in the string, you can also write a PRX to match many characters or even zero-width characters.

In addition to letters, a PRX can also contain digits, metacharacters, and special characters. Metacharacters are a set of characters that means something different from its literal meaning. For example, in PRX, a period (**.**) not only matches itself, but also matches any single character except for a newline character (**\n**).

Special characters are some ASCII characters that affect the structure and behavior of PRX. The most commonly used special characters are a forward slash **/**, parentheses **( )**, a vertical bar **|**, and a backslash ****.

- forward slash **/**: The forward slash is used to enclose the PRX
- Parentheses **( )**: Using parentheses creates logical groups of pattern characters and metacharacters.
- Vertical bar **|**: logical OR. For example, **/Zack|ZACK/** can match either "Zack" or "ZACK"
- Black slash ****:
  - A backslash is mostly used as part of metacharacters. For example, the metacharacter **\d** matches on numerical digits (0-9). It tells SAS that the "d" component is not a regular letter by placing a backslash in front of the "d"
  - Sometimes you may want to treat a special character as a regular text. In this situation, you need to place a backslash in front of this special character. We often call the backslash an "escape" character

You can add these special characters, such as **|** or parentheses, to make your matching more flexible. For example:

- **/(Z|z)ack/** will match either "Zack" or "zack"
- **/(Z|z)ack|(J/j)ohn/** will match either "Zack" or "zack", "John", or "john"
- **/\(Zack\)/** will match "(Zack)"

Both metacharacters and other special characters will be introduced gradually in the following subsection.

USING METACHARACTERS TO MATCH ONE CHARACTER IN THE SEARCHED STRING

The simple example of matching one character in the string is demonstrated in the previous program. Sometimes you might to want to use one character in the PRX to match different characters in the string. In this situation, you need to utilize metacharacters in the PDX. The most commonly used metacharacters are summarized in the Table 1.

Table 1: Examples of Meta characters

Metacharacter	Description	Example	Matches
Wild card: **.**	Matches any single character except for (\n)	**/D.g/**	"Dog", "D.g", "D8g", "D-g", …
Word: **\w**	Matches a-z, A-Z, 0-9 and underscore(_)	**/D\wg/**	"Dog", "Dag", "D8g", "D_g", …
Non-word: **\W**	Matches a value that **\w** doesn't match, except for \n.	**/D\Wg/**	"D-g", "D!g", "D.g", "D g",…
Whitespace: **\s**	Matches one single white space, tab, or newline (\n)	**/D\sg/**	"D g"
Non-whitespace: **\S**	Matches a value that **\s** doesn't match	**/D\Sg/**	"Dog", "D8g", "D-g", "D!g",…
Digit: **\d**	Matches on one numerical digit (0-9)	**/D\dg/**	"D1g", "D8g", "D0g",…
Non-digit: **\D**	Matches on one non-numerical character	**/D\Dg/**	"Dog", "D_g", "D-g", "D!g",…

USING A CHARACTER CLASS TO MATCH ONE CHARACTER IN THE SEARCHED STRING

We can create a character class by enclosing all the possible values within a pair of square brackets to match one single character in the searched string. Here are the characteristics of character class that one needs to know:

- You can include metacharacter(s) in the character class
- When you include a special character in the character class, this special character will only be treated as a regular text. No need to place a backslash in front. For example, (will match (, | will match |, etc
- You can use the range notation (-) with the character class. For example, **[a-d]** matches one character between letter "a" and "d". **[3-7]** matches one digit between 3 and 7.
- You can also exclude a list of characters by placing **^** at the beginning of the character class (except for \n)

Table 2: Examples of User-Defined Character Classes

Character class	Description	Example	Matches
[io23]	Matches "i", "o", "2", or "3"	**/D[io23]g/**	"Dig", "Dog", "D2g", "D3g"
[1-5]	Matches 1 to 5	**/A[1-5]/**	"A1", "A2", "A3", "A4", "A5"
[\da-c]	Matches all digits and letters "a", "b", and "c"	**/D[\da-c]g/**	"D0g"… "D9g", "Dag", "Dbg", "Dcg"
[^aeiou]	Matches anything other than lower case vowels	**/D[^aeiou]g/**	"D0g", "Dbg", "Dcg", "Ddg", "D!g", …

MATCHING ZERO, ONE, OR MORE CHARACTERS IN THE SEARCHED STRING

You can use a repetition modifier, which is also a type of special character, to match characters in the strings, and the number of characters in the searched strings can be varied. The repetition modifiers only change the behavior of the metacharacters or characters immediately preceding them. If you want to repeat the entire group of characters, you can place them in parentheses. Examples of using repetition modifiers are shown in Table 3.

Table 3: Examples of Using Repetition Modifiers

Repetition Modifier	Description	Example	Matches
*	Requests preceding character to match 0 or more times.	/Art*/	"Art", "Arthur", "Artie", …
		/U\W*S\W*/	"US", "U.S", "U.S.-", ..
+	Requests preceding character to match 1 or more times.	/1st +St/	"1st St", "1st St", …
		/(Hi)+/	"Hi", "HiHiHi", …
?	Requests preceding character to match 0 or 1 time.	/1\D?800\D?123\D?4567/	"18001234567", "1-800-123-4567"
{n}	Requests preceding character to match exactly n times.	/1-800-\d{3}-\d{4}/	"1800-123-4567","1-800-345-8797", …
{n,}	Requests preceding character to match at least n times.	/1-800-\d{1,}-\d{2,}/	"1800-123-4567" "1-800-12-345", …
{n,m}	Requests preceding character to match n to m times	/A\w{2,5}\d{1,2}/	"Art0", "Arthur12", …

MATCHING ZERO-WIDTH CHARACTERS IN THE SEARCHED STRING

In some situations, one might want to create a PRX that matches characters at the beginning or end of the string, or at the beginning or end of a word. Table 4 summarizes the metacharacters that you can use to achieve such a task.

Table 4: Methacharacters to Locate Boundaries.

Metacharacter	Description	Example	Matches
^	Matches beginning of a line or string	/^Zack/	Matches "Zack Johnson", *not* "A Zack"
$	Matches end of a line or string	/\d{5}$/	CA 91768
\b	Matches a word boundary; it separates a \w character and a \W character	/Ave/b/	Matches "Ave" from "Ave," *not* "Avenue"
		/\b9\d\d\b/	Matches "900" from "(900)", *not* "19001"
\B	Matches a non-word boundary; the opposite of **\b**	/run\B/	Matches "run" from "running", *not* "run,"

PRX FUNCTIONS AND CALL ROUTINES

In this section, the four PRX functions PRXMATCH, PRXCHANGE, PRXPOSN, and PRXPAREN) and four PRX call routines (CALL PRXCHANGE, CALL PRXPOSN, CALL PRXSUBSTR, and CALL PRXNEXT) will be reviewed.

THE PRXMATCH FUNCTION

The PRXMATCH function is used to search for a pattern match. If the pattern is found, the function will return the position at which the pattern is found. When there are multiple matches that are found, only the position of the first match is returned. If there is no match, PRXMATCH returns a zero.

PRXMATCH(*regular-expression-id* | *perl-regular-expression*, *source*)

You can either use the *regular-expression-id*, which is a pattern identifier that is returned from the PRXPARSE function, or a *perl-regular-expression* for the function. The *source* argument is used to specify a character constant, variable, or expression that you want to search.

For example, Program 3 uses the PRXMATCH function to identify whether the input addresses contain zip codes. If a zip code is found, then the address will be printed in the log.

Program 3:

```
data _null_;
    patternID=prxparse("/\w\w\s?\d{5}/o");
    input address $80.;
    position = prxmatch(patternID, address);
    if position ^= 0 then put address= position=;
    datalines;
Zack Johnson, 153 First Str, Chapel Hill, NC27514
Dan Zack, 67891 64th st, Brea, CA
Sally Johns, 4 Moritz Street, Duarte, CA 91010
;
```

There are many ways to write the PRX pattern to locate the zip code. However, there are a couple of issues one needs to consider when creating this PRX. First of all, the address number can also be a five digit number. Secondly, there is no space between the state abbreviation and the zip code in the first input address. Thus, the PRX pattern in Program 3 is written as **`/\w\w\s?\d{5}/:`**

- **`\w\w`**: Two words elements
- **`\s?`**: One or zero space
- **`\d{5}`**: 5 digits

THE CALL PRXSUBSTR ROUTINE

Similar to the PRXMATCH function, the CALL PRXSUBSTR routine not only can return the position of a matched pattern, it can also return the length of the searched pattern.

CALL PRXSUBSTR(*regular-expression-id*, *source*, *position* <, *length*>);

The *position* argument is a numeric variable with a returned value that is the position in *source* where the pattern begins. If no match is found, CALL PRXSUBSTR = returns zero. The optional *length* variable is a numeric variable with a returned value that is the length of the substring that is matched by the pattern. If no match is found, CALL PRXSUBSTR returns zero.

Program 4 illustrates an example to extract the zip code from the input address. The PRX pattern that is defined in Program 3 is to match both the two-letter state abbreviation and the zip code. The PRXSUBSTR routine in Program 4 identifies the position and length of the pattern and stores them in the POSITION and LENGTH variable. If a pattern is matched, both the state and the zip code will be extracted by using the SUBSTR function and the resulting values will be stored in the STATE_ZIP variable. The second CALL PRXSUBSTR is also needed to identify the location and length of the zip code from the newly-created STATE_ZIP variable by using the **`/\d{5}/`** pattern. Finally, another SUBSTR function is used to extract the zip code.

Program 4:

```
data ex4 (keep=address zip);
     patternID=prxparse("/\w\w\s?\d{5}/o");
     patternID2=prxparse("/\d{5}/o");
     input address $80.;
     call prxsubstr(patternID, address, position, length);
     if position ^= 0 then do;
         state_zip = substr(address, position, length);
         call prxsubstr(patternID2, state_zip, position_zip, length_zip);
        zip = substr(state_zip, position_zip, length_zip);
     end;
     else zip = " ";
     datalines;
Zack Johnson, 153 First Str, Chapel Hill, NC27514
Dan Zack, 67891 64th st, Brea, CA
Sally Johns, 4 Moritz Street, Duarte, CA 91010
;

proc print data=ex4;
run;
```

Listing Output from Program 4:

```
                          The SAS System
              Obs    address                                          zip

               1     Zack Johnson, 153 First Str, Chapel Hill, NC27514    27514
               2     Dan Zack, 67891 64th st, Brea, CA
               3     Sally Johns, 4 Moritz Street, Duarte, CA 91010       91010
```

THE CALL PRXPOSN ROUTINE

The method of extracting the zip code in Program 4 used two CALL PRXSUBSTR and two SUBSTR functions to complete the task. Alternatively, you can also us the PROXPOSN routine to accomplish such a task.

The PROXPOSN routine requires an understanding of the *capture buffer* concept. Remember that we can use parentheses to create a logical grouping within PRX. When a pair of parentheses are used within the PRX, a slot in the memory buffer is created. Each slot can be referenced accordingly in the sequential order of the parenthesized pair in the PRX. For example, if we place two parenthesis pairs in the pattern **`/(\w\w)\s?(\d{5})/`**, then the first capture buffer would refer to **`\w\w`**, and the second one would refer to **`\d{5}.`** The CALL PRXPOSN is used to return the starting position and length of a capture buffer.

CALL PRXPOSN(*regular-expression-id*, *capture-buffer*, *start* <, *length*>);

The first argument is *regular-expression-id* that is returned by the PRXPARSE function. The *capture-buffer* argument is a numeric constant, variable, or expression with a value that locates the capture buffer from which to retrieve the starting position and length. The *start* argument is a numeric variable with a returned value that is the position at which the capture buffer is found. If the value of *capture-buffer* is not found, CALL PRXPOSN returns a zero value. The optional *length* argument is a numeric variable with a returned value that is the pattern length of the previous pattern match. If the pattern match is not found, CALL PRXPOSN returns a zero as well.

To use the CALL PRXPOSN, you need to use the PRXMATCH, PRXSUBSTR, PRXCHANGE, or PRXNEXT functions (or routines) to return capture buffers. Program 5 uses PRXMATCH to locate the pattern in the input address. If the pattern is found, then CALL PRXPOSN is used to identify the starting position and the length of the second specified buffer, which is the zip code. Then the extracted starting position and the length of the zip code is used in the SUBSTR function to create the ZIP variable.

Program 5:

```
data ex5 (keep=address zip);
    patternID=prxparse("/(\w\w)\s?(\d{5})/o");
    input address $80.;
    position = prxmatch(patternID, address);
    if position ^= 0 then do;
        call prxposn(patternID, 2, start, length);
        zip = substr(address, start, length);
    end;
    else zip = " ";
    datalines;
Zack Johnson, 153 First Str, Chapel Hill, NC27514
Dan Zack, 67891 64th st, Brea, CA
Sally Johns, 4 Moritz Street, Duarte, CA 91010
;

proc print data=ex5;
run;
```

Listing Output from Program 5

```
                          The SAS System
              Obs    address                                        zip

               1     Zack Johnson, 153 First Str, Chapel Hill, NC27514    27514
               2     Dan Zack, 67891 64th st, Brea, CA
               3     Sally Johns, 4 Moritz Street, Duarte, CA 91010       91010
```

THE PRXPOSN FUNCTION

The PRXPOSN function is similar to the CALL PRXPOSN routine except that the PROXPOSN function returns the capture buffer itself rather than the position and the length of the capture buffer.

PRXPOSN(*regular-expression-id*, *capture-buffer*, *source*)

The *source* argument is used to specify the text from which to extract capture buffers. Similar to CALL PRXPOSN, the PRXPOSN function uses PRXMATCH, PRXSUBSTR, PRXCHANGE, or PRXNEXT to return capture buffers. Program 6 is a modified version of Program 5 by using the PRXPOSN function.

Program 6:

```
data ex6 (keep=address zip);
    patternID=prxparse("/(\w\w)\s?(\d{5})/o");
    input address $80.;
    position = prxmatch(patternID, address);
    if position ^= 0 then do;
        zip = prxposn(patternID, 2, address);
    end;
    else zip = " ";
    datalines;
Zack Johnson, 153 First Str, Chapel Hill, NC27514
Dan Zack, 67891 64th st, Brea, CA
Sally Johns, 4 Moritz Street, Duarte, CA 91010
;
```

THE PRXCHANGE FUNCTION

The PRXCHANGE function can be used to perform a replacement for a matched pattern.

PRXCHANGE(*perl-regular-expression* | *regular-expression-id*, *times*, *source*)

You can use either a PRX or a *regular-expression-id* that is returned from the PRXPARSE function as the first argument. The *times* argument is a numeric constant, variable, or expression that specifies the number of times to search for a match and replace a matching pattern. If you use -1 for the *times* argument, then matching patterns continue to be replaced until the end of *source* is reached. The *source* argument is used to specify a character constant, variable, or expression that you would like to search.

The situation for using the PRXCHANGE can apply to applications of data standardization. For example, the addresses that we entered in the previous program have different ways of writing the word “Street.” Both first and second records use the abbreviations (Str and st). If we want to replace the abbreviation with the word “Street”, we can use the PRXCHANGE function.

When writing a PRX for the PRXCHANGE function, you need to write two components: the first component is the pattern that you would like to search, and the second component is the pattern that you will use to replace. These two components need to be separated by a slash (**/**). Furthermore, a substitution operator (s) needs to be placed in front of the expression. Program 7 illustrates how to replace the street abbreviation with the full word “Street.”

Program 7:

```
data _null_;
    input address $80.;
    new_address=prxchange("s/\s+[sS]t(reet|r)?/ Street/o", -1, address);
    put new_address=;
    datalines;
Zack Johnson, 153 First Str, Chapel Hill, NC27514
Dan Zack, 67891 64th st, Brea, CA
Sally Johns, 4 Moritz Street, Duarte, CA 91010
;
```

SAS Log from Program7:

```
2047  data _null_;
2048      input address $80.;
2049      new_address=prxchange("s/\s+[sS]t(reet|r)?/ Street/o", -1, address);
2050      put new_address=;
2051      datalines;

new_address=Zack Johnson, 153 First Street, Chapel Hill, NC27514
new_address=Dan Zack, 67891 64th Street, Brea, CA
new_address=Sally Johns, 4 Moritz Street, Duarte, CA 91010
```

```
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds
```

You can reference a capture buffer by using its reference number after a dollar sign ($) in the PDX. Program 8 shows how to switch the first and last names and separate them by a comma. The first component of the PRX **^(\w\w+)\s+(\w\w+)**, is used to identify the names of the address. The second component, **$2, $1**, is to arrange the order of the first and second buffer and separate them by a comma.

Program 8:

```
data _null_;
    input address $80.;
    new_address=prxchange("s/^(\w\w+)\s+(\w\w+)/$2, $1/o", -1, address);
    new_address1=prxchange("s/\s+[sS]t(reet|r)?/ Street/o", -1, new_address);
    put new_address1=;
    datalines;
Zack Johnson, 153 First Str, Chapel Hill, NC27514
Dan Zack, 67891 64th st, Brea, CA
Sally Johns, 4 Moritz Street, Duarte, CA 91010
;
```

SAS Log from Program8:

```
2056  data _null_;
2057      input address $80.;
2058      new_address=prxchange("s/^(\w\w+)\s+(\w\w+)/$2, $1/o", -1, address);
2059      new_address1=prxchange("s/\s+[sS]t(reet|r)?/ Street/o", -1, new_address);
2060      put new_address1=;
2061      datalines;

new_address1=Johnson, Zack, 153 First Street, Chapel Hill, NC27514
new_address1=Zack, Dan, 67891 64th Street, Brea, CA
new_address1=Johns, Sally, 4 Moritz Street, Duarte, CA 91010
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds
```

THE CALL PRXCHANGE ROUTINE

Similar to the PRXCHANGE function, the CALL PRXCHANGE routine can also perform a replacement for matched patterns.

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

Unlike the PRXCHANGE function, the first argument can only be a *regular-expression-id* that is returned from the PRXPARSE function. The *times* argument is a numeric constant, variable, or expression that specifies the number of times to search for a match and replace a matching pattern. If you use -1 as the value *times*, then matching patterns continue to be replaced until the end of *source* is reached. The *old-string* argument is used to specify the character expression on which to perform a search and replace. The optional *new-string* argument is used to specify a variable in which to store the replacement result. If the *new-string* argument is not specified, the replacement changes will be made to the *old-string.* The optional *result-length* is a numeric variable that contains the value of the number of characters that are copied to the result. The optional *truncation-value* is also a numeric variable with a value equaling 1 when the entire replacement result is longer than the length of the *new-string;* otherwise, the value will be set to 0. The last optional argument *number-of-changes* is a numeric variable that records the number of replacements that were made.

Similar to Program 8, Program 9 switches the order of first and last names by using CALL PRXCHANGE. After the replacement, the modified values are stored in the new variable NEW_ADDRESS.

Program 9:

```
data _null_;
    input address $80.;
    length new_address $80;
    patternID=prxparse("s/^(\w\w+)\s+(\w\w+)/$2, $1/o");
    call prxchange(patternID, -1, address, new_address);
    put new_address=;
    datalines;
Zack Johnson, 153 First Str, Chapel Hill, NC27514
Dan Zack, 67891 64th st, Brea, CA
Sally Johns, 4 Moritz Street, Duarte, CA 91010
;
```

SAS Log from Program9:

```
2067  data _null_;
2068      input address $80.;
2069      length new_address $80;
2070      patternID=prxparse("s/^(\w\w+)\s+(\w\w+)/$2, $1/o");
2071      call prxchange(patternID, -1, address, new_address);
2072      put new_address=;
2073      datalines;

new_address=Johnson, Zack, 153 First Str, Chapel Hill, NC27514
new_address=Zack, Dan, 67891 64th st, Brea, CA
new_address=Johns, Sally, 4 Moritz Street, Duarte, CA 91010
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds
```

THE PRXPAREN FUNCTION

The PRXPAREN function is used to return a value of the largest capture buffer that contains the data of the first match. You need to use the PRXSUBSTR, PRXMATCH, PRXNEXT, or PRXCHANGE functions (routines) with the PRXPAREN together to obtain the accurate result.

PRXPAREN(*regular-expression-id*)

The only requirement for the PRXPAREN function is the *regular-expression-id*, which needs to be returned by the PRXPARSE function.

Program 10 illustrates the use of the PRXPAREN function. Notice that there are four groupings in the PRX. The outer parentheses that enclosed `((one)|(two)|(three))` is referenced as 1, then `(one)` is referenced as 2, `(two)` referenced as 3, and`(three)`is referenced as 4. Once a match is found, the PRXMATCH function returns the value of the largest capture buffer. For example, the first line of the input data contains “one”, “two”, and “three,” and “one” is the first matched word in the pattern. Although “one” is in both buffer 1 and buffer 2, only 2 is returned by the PRXPAREN function.

Program 10:

```
data ex10 (keep=numbers which);
    input numbers $15.;
    patternID=prxparse("/((one)|(two)|(three))/o");
    position = prxmatch (patternID, numbers);
    if position then do;
        which=prxparen(patternID);
        output;
    end;
    datalines;
one two three
two XXX YYY
XXX YYY three
four five six
;
```

```
proc print data=ex10;
run;
```

Listing Output from Program 10:

```
                          The SAS System

                Obs       numbers       which

                 1     one two three      2
                 2     two XXX YYY        3
                 3     XXX YYY three      4
```

A useful application is to use the PRXPAREN function to find the largest capture-buffer number first. Then pass this number to the CALL PRXPOSN routine to extract the position and length of the desired match.

For example, suppose that a telephone number is entered by either using parentheses to enclose the area code and using a dash to separate the rest of the seven digits or all the numbers are separated by two dashes. Thus, there are two ways of writing the matched patterns. At the beginning of Program 11, both patterns were entered as strings, INPUT1 and INPUT2. Then the final pattern is created by combining INPUT1 and INPUT2 by using the concatenation operator (||) along with the OR (|) operator. Notice that the final pattern will contain two groupings with one corresponding to each input pattern. The purpose of this program is to identify whether the input address with the phone numbers contains the area code from Orange County (Area codes: 949 or 714). Once the pattern is matched (from the PRXMATCH function), the PRXPAREN function returns the buffer number that identifies the match. Then this number is passed to the CALL PRXPOSN routine to extract the position and length of the area code. In the final step, if an OC area code is matched with the extracted area code, the address will be printed in the log.

Program 11:

```
data _null_;
    input1 = "\(([2-9]\d\d)\) ?[2-9]\d\d-\d{4}";
    input2 = "([2-9]\d\d)-[2-9]\d\d-\d{4}";
    both_input = "/(" || input1 || ")|(" || input2 || ")/o";
    patternID = prxparse(both_input);
    OC_ID = prxparse("/714|949/o");

    length areacode $ 3;
    input address $80.;

    if prxmatch(patternID, address) then do;
         which_format = prxparen(patternID);
         call prxposn(patternID, which_format, pos, len);
         areacode = substr(address, pos, len);
         if prxmatch(OC_ID, areacode) then  put "In OC:" address=;
    end;
    datalines;
Zack Johnson, 153 First Str, Chapel Hill, NC27514 (828) 345-2345
Dan Zack, 67891 64th st, Brea, CA 714-320-1000
Sally Johns, 4 Moritz Street, Duarte, CA 91010 310-232-0001
Eric Johnson, 112 Dublin Ln, Irvine, CA (949) 230-3209
;
```

SAS Log from Program11:

```
2797  data _null_;
2798       input1 = "\(([2-9]\d\d)\) ?[2-9]\d\d-\d{4}";
2799       input2 = "([2-9]\d\d)-[2-9]\d\d-\d{4}";
2800       both_input = "/(" || input1 || ")|(" || input2 || ")/o";
2801       patternID = prxparse(both_input);
2802       OC_ID = prxparse("/714|949/o");
2803
2804       length areacode $ 3;
2805       input address $80.;
2806
```

```
2807      if prxmatch(patternID, address) then do;
2808           which_format = prxparen(patternID);
2809           call prxposn(patternID, which_format, pos, len);
2810           areacode = substr(address, pos, len);
2811           if prxmatch(OC_ID, areacode) then  put "In OC:" address=;
2812      end;
2813      datalines;

In OC:address=Dan Zack, 67891 64th st, Brea, CA 714-320-1000
In OC:address=Eric Johnson, 112 Dublin Ln, Irvine, CA (949) 230-3209
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds
```

THE CALL PRXNEXT ROUTINE

The CALL PRXNEXT routine is used to return the position and length of a substring that matches a PRX pattern, and iterates over multiple matches within one string.

CALL PRXNEXT(*regular-expression-id*, *start*, *stop*, *source*, *position*, *length*);

The *start* argument is a numeric variable that specifies the position at which to start the pattern matching in the *source* argument. The *stop* is a numeric constant, variable, or expression that specifies the last character to use in *source*. The *source* argument is to specify a character constant, variable, or expression that you want to search. The *position* argument is a numeric variable with a returned value that is the position in *source* at which the pattern begins. Lastly, the *length* argument is a numeric variable with a returned value that is the length of the string that is matched by the pattern.

Program 12 illustrates an example of using the PRXNEXT routine to extract all the numeric values from the input addresses. The START and STOP arguments were initialized to 1 and the length of the input address respectively. The first PRXNEXT routine is used to extract the starting position and length of the first match. Then an iterative DO WHILE loop is used to continue extracting the match. Within the loop, the SUBSTR function is used to extract the numbers based on the extracted position and length.

Program 12:

```
data ex12 (keep=address num1-num5);
    patternID=prxparse("/\d+/o");
    input address $80.;
    start = 1;
    stop = length(address);
    call prxnext(patternID, start, stop, address, position, length);
    array num[5] $;
     do i = 1 to 5 while (position > 0);
         num[i]=  substr(address, position, length);
        call prxnext(patternID, start, stop, address, position, length);
    end;
    datalines;
Zack Johnson, 153 First Str, Chapel Hill, NC27514 (828) 345-2345
Dan Zack, 67891 64th st, Brea, CA 714-320-1000
Sally Johns, 4 Moritz Street, Duarte, CA 91010 310-232-0001
Eric Johnson, 112 Dublin Ln, Irvine, CA (949) 230-3209
;
proc print data=ex12;
run;
```

Listing Output from Program 12

Listing of Data Set FIND_NUM

Obs	address	num1	num2	num3	num4	num5
1	Zack Johnson, 153 First Str, Chapel Hill, NC27514 (828) 345-2345	153	27514	828	345	2345
2	Dan Zack, 67891 64th st, Brea, CA 714-320-1000	67891	64	714	320	1000
3	Sally Johns, 4 Moritz Street, Duarte, CA 91010 310-232-0001	4	91010	310	232	0001
4	Eric Johnson, 112 Dublin Ln, Irvine, CA (949) 230-3209	112	949	230	3209	

CONCLUSION

In this paper, only a small proportion of the syntax for writing PRX is introduced. However, based on the examples in this paper, you have probably started to realize how powerful the PRX can be, especially when you start to work on large and un-cleaned data. Thoroughly knowing PRX might become an essential skill for being a successful programmer.

REFERENCES

Cody, Ron. (2004). “An Introduction to Perl Regular Expressions in SAS 9,” in Proceedingsof SUGI29, Montréal, Canana

SAS Institute. (2010). *SAS® 9.4 Functions and CALL Routines Reference.* Cary, NC: SAS Institute.

Windham, K. Matthew. (2014). *Introduction to Regular Expressions in SAS ®.* Cary, NC: SAS Institute.

CONTACT INFORMATION

Arthur X. Li
City of Hope National Medical Center
Division of Information Science
1500 East Duarte Road
Duarte, CA 91010 - 3000
Work Phone: (626) 256-4673 ext. 65121
Fax: (626) 471-7106
E-mail: arthurli@coh.org