

Paper 138-30

PRX Functions and Call Routines

David L. Cassell, Design Pathways, Corvallis, OR

ABSTRACT

With the advent of SAS® 9, the power of Perl's regular expressions is now available in the DATA step. The PRX functions and call routines enable you to use the pattern matching features of Perl 5.6.1 to search for patterns in text, to save information about matching patterns, and to substitute new forms for the matched patterns. The PRX functions are invaluable for all manner of text manipulation, including validation, text replacement, and string testing.

In this workshop, you will learn:

- how to use Perl regular expressions in the DATA step
- how to write SAS code to access the PRX functions and CALL routines
- how to write some simple Perl regular expressions
- how to use these Perl regular expressions to search for text and replace text

Familiarity with the DATA step is required. No prior knowledge of Perl is needed, but it won't hurt.

WHAT'S A REGULAR EXPRESSION?

Regular expressions are simply a way of characterizing and parsing text. Regular expressions achieve this by describing how the sections of text ought to appear. This means that we have to use a sort of miniature programming language just to this description of patterns. Each regular expression is just a string of characters designed to tell the program what sorts of patterns you want to find.

The simplest form of a regular expression is just a word or phrase for which to search. For example, the phrase

```
Groucho Marx
```

could be a regular expression. We would use it by putting the phrase inside a pair of slashes, like this:

```
/Groucho Marx/
```

And this would tell our program to search for any string which contained the exact words 'Groucho Marx' anywhere inside it. The phrase 'I saw Groucho Marx' would match, as would the phrase 'Groucho Marx was funny' or the phrase 'Groucho Marx' with no other letters. But the phrase 'Groucho and Me' would not have the exact string inside it, and hence would not match.

If this was all that regular expressions could do, then they would not be worth very much. The substr() function can do this much. But, as we will see, regular expressions can do a lot more.

Regular expressions are more common than you realize. If you have ever used wildcards to look for files in a directory, then you have use a form of regular expressions. And many of these forms look a lot like Perl regular expressions. If you have ever typed something like:

```
dir DA*.sas
```

then you have used regular expressions to look for patterns. But, if you have done this, then be forewarned that these are not truly Perl regular expressions, and the meaning of the asterisk and period are different for Perl (and hence the PRX functions). This is one of the easiest traps to fall into when working with Perl regular expressions. The asterisk and period are 'metacharacters', characters with special meanings in regular expressions, so pay careful attention when we cover the meaning of the asterisk and the period, and remember that they won't work the way you expect them to when using MS-DOS wildcards at a DOS prompt.

HOW DO THE PRX... FUNCTIONS WORK IN A SAS 9 DATA STEP?

As of SAS 9.0, Perl regular expressions are available in the SAS DATA step. The simplest form will look like the example above, a text string within slashes. Since text strings need to be quoted properly in SAS functions, we will need to remember to put quotes around those slashes as well.

Suppose we have a simple database of names and phone numbers for our company to check. The database will have only first name, last name, and phone number (including area code). A small database like this might look like the following:

| Obs | lastname | firstname | phonenum |
|-----|--------------|-----------|----------------|
| 1 | Marx | Chico | 412-555-4242 |
| 2 | Marx | Harpo | 541 555-3775 |
| 3 | Marx | Groucho | (909) 555-3389 |
| 4 | Marx | Karl | (404)555-9977 |
| 5 | Ma rx | Zeppo | (664) 555-8574 |
| 6 | Matrix | Jon | (703) 555-6732 |
| 7 | von Trapp | Maria | 928-555-6060 |
| 8 | van den Hoff | Friedrich | 870-555-3311 |
| 9 | MacDonald | Ole | 677-555-5687 |
| 10 | MacDuff | Killegan | (854)555-8493 |
| 11 | McMurphy | Randall | 422-555-4738 |
| 12 | Mac Heath | Mack | 956-555-4141 |
| 13 | Potter | Lily | (646) 555-3324 |
| 14 | Potter | James | (646) 555-3324 |

It is not that hard to look at the above table and see a few potential problems. With only fourteen records, simple inspection is possible. But with fourteen thousand records, or fourteen million records, examining the records by hand becomes unreasonable. Let's use this database to explore the basics of the PRX functions.

If we want to search this database for all records containing the string 'arx', we can use the following program:

```
data check_arx;
  set prx.checkfile;
  if _n_=1 then do;                               /* 1 */
    retain re;                                     /* 2 */
    re = prxparse('/arx/');                         /* 3 */
    if missing(re) then do;                         /* 4 */
      putlog 'ERROR: regex is malformed';          /* 5 */
      stop;                                         /* 6 */
    end;                                           /* 7 */
  end;

  if prxmatch(re,lastname);                         /* 8 */
run;

proc print data=check_arx;
  var lastname firstname;
  title 'Last name matches "arx" ';
run;
```

This program takes our database above and generates the lines:

| Obs | lastname | firstname |
|-----|----------|-----------|
| 1 | Marx | Chico |
| 2 | Marx | Harpo |
| 3 | Marx | Groucho |
| 4 | Marx | Karl |

Now how does this program work? In the line marked [1], we begin a DO-group which runs only when we read in the first record of the database. This is the right time to perform tasks which need only be done once, but must be done before any other processing of the file. The subsequent lines [2] through [6] are executed once, before this processing is done.

In the line marked [2], we use the RETAIN statement so that the regular expression RE will be available as every record of the database is processed.

In the line marked [3], we actually use the PRXPARSE function to build our regular expression RE. As with all DATA step functions, we need parentheses around the parameters of the function. Within the parentheses, we need quotes around the string which represents our Perl function. The slashes are in fact the Perl matching function. Within the slashes is the regular expression, in this case only a text string. But this leads to the complicated-looking form

```
prxparse('/arx/')
```

which we will see repeated through much of this paper. Remember: parentheses, around quotes, around the Perl function. The two types of Perl functions we will see are the above matching function above and the Perl substitution function, which in a PRXPARSE function will look like:

```
prxparse('s/pattern to find/text to substitute/')
```

In the lines marked [4] through [7], we introduce error-handling in the DATA step. If the pattern for the regular expression is poorly written, or we ask for some component of Perl regular expressions which is not available in SAS 9, then the PRXPARSE function will fail. In this case, the value of RE will be missing. We check for this once, before beginning our processing of the file. If missing(re) is true, then the do-group will cause the DATA step do two things. It will use the PUTLOG statement to write an error message to the log [the line marked 5] and it will stop the execution of the DATA step before processing [the line marked 6]. This error-checking is optional - in the sense that never checking your rearview mirrors while driving a car is optional. It works fine, until you crash horribly. Until you really feel comfortable developing Perl regular expressions and using them in the PRX functions, it is strongly recommended that you maintain this sort of error code. If you are going to be using functions like this in production code, good error-checking is essential. And, if you are going to be writing self-modifying code – code which can change as inputs vary – then error-checking is critical.

Finally, in the line marked [8], we use the PRXMATCH function. PRXMATCH requires two input parameters. The first must be the name of the regular expression variable, and the second must be the name of the character expression you wish to search. PRXMATCH returns the numeric position in the character string at which the regular expression pattern begins. If no match is found, then PRXMATCH returns a zero. If a match is found, then PRXMATCH returns the starting position of the matching string, which will be a whole number greater than 0. Here, we use the Boolean nature of the IF statement to select any record for which PRXMATCH returns a non-zero number, that is, any record for which a match is found in the LASTNAME variable.

PERL REGULAR EXPRESSION FORMS

Now that we have seen a character string, the simplest form of the Perl regular expression, we are ready to take a look at some of the basics of Perl regular expressions so we can perform more complicated pattern searches. But we have seen the first of the rules: concatenation. If you want to follow 'r' with 'x', just write 'rx'. The five basics are:

- concatenation
- wildcards
- iterators
- alternation
- grouping

Now that we know what 'concatenation' really means, let's look at the next step, wildcards.

Perl uses simple text strings, as we have already seen. But Perl also uses wildcards, special characters (called metacharacters in Perl) which stand for more than one single text character. A few of the common ones are:

- . the period matches exactly one character, regardless of what that character is

`\w` a 'word'-like character, `\w` matches any of the characters a-z, A-Z, 0-9, or the underscore
`\d` a 'digit' character, `\d` matches the numbers 0 to 9 only
`\s` a 'space'-like character, `\s` matches any whitespace character, including the space and the tab
`\t` matches a tab character only
`\W` a 'non-word' character, that is, anything not matched by `\w`
`\D` a 'non-digit' character, that is, anything not matches by `\d`
`\S` a 'non-whitespace' character, that is, anything not matched by `\s`

Note that the period is a wildcard. When using MS-DOS regular expressions, the period is a real period marking the boundary between the file name and the file extension. Don't let that trip you up when working with a Perl regex (shorthand for 'regular expression').

Now let us look at a few more examples.

```
/a.x/
```

This would match any string which contained an 'a', followed by any character, followed by an 'x'. This would match the 'arx' in 'Marx', as well as the 'anx' in 'Manx' or 'phalanx', and the 'aux' in 'auxiliary'. It would not match 'Matrix', as the period will only match one character, and there are three characters between the 'a' and 'x' in Matrix.

```
/M\w\wx/
```

This would match any string which contained 'M', two 'word' characters, and an 'x'. It would match 'Marx'. But it would also match 'M96x' and 'M_1x', since `\w` matches numbers and underscores as well. Be sure that your use of wildcards doesn't lead to too many false positives!

Perl also provides 'iterators', ways of indicating that you want to control the number of times a character or wildcard matches. Some of the iterators are:

- * matches 0 or more occurrences of the preceding pattern
- + matches 1 or more occurrences of the preceding pattern
- ? matches exactly 0 or 1 occurrences of the preceding pattern
- {k} matches exactly k occurrences of the preceding pattern
- {n,m} matches at least n and at most m occurrences of the preceding pattern

Now let us look at using these too.

```
/a*x/
```

Those who are used to Win32-style filename patterns will slip up here. Instead of matching 'a' followed by an arbitrary string of characters (as in MS-DOS and win32 naming conventions), or 'a' followed by an arbitrary string of characters, followed by an 'x' (as in unix shells), the Perl regular expression uses '*' as an iterator. In Perl, this asks to match any string which contains 0 or more occurrences of the letter 'a' immediately followed by an 'x'. So this would match 'ax', 'aax', 'aaax', and 'aaaaaaaaaax'. It would also match 'x' alone or 'lox', since zero occurrences of 'a' will match the 'a*' part of the regular expression. This pattern would therefore match 'Marx' and 'Manx' and 'phalanx' and 'Matrix', all of which have an 'x' in them. If you wish to match an 'a', then a string of arbitrary characters, then an 'x', you must write your regular expression as `/a.*x/`, using `.*` to mean 'any number of arbitrary characters'. Use care when using the '*' iterator!

```
/r+x/
```

This would match one or more occurrences of 'r', followed immediately by an 'x'. So it would match 'rx' and 'rrrx' and 'rrrrrrrx'. It would match the 'rx' in 'Marx', but not the 'rix' in 'Matrix'.

```
/ri?x/
```

This would match 'r', followed by an optional 'i', followed by 'x'. So this would match both the 'rx' in 'Marx' and the 'rix' in 'Matrix'.

```
/k\w{0,7}/
```

This would match a 'k', followed by 0 up to 7 'word-like' characters. In fact, this would match any SAS version 6 data set name starting with 'k'. Remember that the \w wildcard matches a 'word' in the sense of legal characters in SAS version 6 dataset and variable names.

```
/Ma\w{1,3}x/
```

This would match a capital 'M', followed by 'a', followed by one to three 'word' characters, followed by 'x'. So this would match 'Matrix' and 'Marx'. It would also match 'Manx' and 'Maalox' and 'M1_9x', though.

Perl also provides ways of grouping expressions and providing alternate choices. The parentheses are the usual choice for grouping, and the 'or' operator | provides alternation. Perl also uses the parentheses to 'capture' chunks of the pattern for later use, so we will want to remember that later.

In particular, when we put anything inside parentheses, the regex engine 'captures' the matching string and places it into a 'capture buffer'. That buffer gets used if we make a substitution with the CALL PRXCHANGE() function. Also, the information about that buffer gets passed back for the use of the CALL PRX... functions, so that we can know the starting position and length of the matching text. We'll see in a while how to use that information.

When we use the '|' symbol for alternation, we can match either of the component pieces in the regex.

```
/r|n/
```

The '|' operator tells us to choose either 'r' or 'n'. So any string which has either an 'r' or an 'n' will match here. Perl also provides another way of making such a pattern – the character class – which we will discuss later.

```
/Ma(r|n)x/
```

The parentheses group the two parts of the 'alternation' pattern. The whole pattern now will only match a capital 'M', then an 'a', then either 'r' or 'n', then 'x'. So only strings containing 'Marx' or 'Manx' will match. Still, strings like 'Marxist' or 'MinxManxMunx' will match this pattern.

```
/Ma(tri|r|n)x/
```

Again, the parentheses group the patterns to be alternated. But now we see that we do not have to have the same length patterns for alternation, and we do not have to stick with only two choices. 'Matrix' or 'Marx' or 'Manx' would all match, as would any string containing one of them.

ANOTHER EXAMPLE WITH PRXMATCH

If we now want to search for only those names starting with 'Mc' or 'Mac', followed by a space, then one or more letters, we can modify our above code. We can use our newly-learned techniques and write the pattern as /(Mc|Mac) \w+/. We use parentheses to group the only part we wish to have alternation, the choice of 'Mc' or 'Mac'. We then have a literal character, the space, followed by the wildcard \w with the iterator '+', meaning 'one or more'. Then our program would look like:

```
data check_mac;
  set prx.checkfile;
  if _n_=1 then do;
    retain re;
    re = prxparse('/(Mc|Mac) \w+/');
    if missing(re) then do;
      putlog 'ERROR: regex is malformed';
      stop;
    end;
  end;

  if prxmatch(re,lastname) > 0;
run;
```

```
proc print data=check_mac;
  var lastname firstname;
  title 'Last name matches Mc or Mac, blank, then "word" ';
run;
```

This produces the single record of output:

| Obs | lastname | firstname |
|-----|-----------|-----------|
| 1 | Mac Heath | Mack |

Note in passing that the regular expression could also have been written as `/Ma?c \w+/` with the '?' iterator handling the difference in beginnings for the last names. 'Ma?c' looks for 'M', then an optional 'a', then a 'c'. This would find both 'Mc' and 'Mac', without accidentally matching any undesired strings. There is usually more than one way to work out a regular expression, and sometimes there can be a great number of alternatives.

YET MORE METACHARACTERS

Perl also has special characters (more of those metacharacters) which don't match a character at all, but represent a particular place in a string.

^ represents the beginning of the string, before the first character
 \$ represents the end of the string, after the last character
 \b represents a word boundary, the position between a 'word' character and a 'non-word' character
 \B matches when we are NOT at a word boundary

The metacharacters \b and \B are somewhat subtle. They are zero-width features (called 'assertions' in Perl, if you want to throw around some jargon and impress your colleagues) which take some getting used to.

```
/^M\w+x$/
```

The '^' and '\$' indicate that the pattern must match at the beginning of the string and finish up at the end of the string. The \w+ will match any sequence of 'word' characters'. So this would match 'Marx' and 'Matrix' and 'MinxManxMunx'. But it would not match 'minManx' (does not start with a 'M'). It would not match 'Marxist' (does not end with an 'x'). It would match 'M39x', since digits and underscores are also matched by \w.

```
/^M\w{2,6}x\b/
```

Now this matches a string which begins with 'M', has two to six additional 'word' characters, followed by an 'x', and then the word must end. It could be followed by a space, or a tab, or a percent sign, or a dash. But it can not be followed one of the characters which are represented by the \w metacharacter. 'b' will only match at the boundary of a 'word', either immediately before or immediately after a string of characters that would be matched by \w.

```
/^M\w+x\B/
```

This matches a string which begins with 'M', has one or more 'word' characters, then an 'x', and then something other than a 'non-word' character. The word cannot end at the 'x' and still match, because of the \B metacharacter. This would match 'Marxist' or 'MinxManxMunx', because both of these strings could match starting with the capital 'M' and then match an 'x' in the middle of the 'word'. This would not match 'Matrix' or 'Marx', because there is no way to get a match without having the 'x' as the last letter of the word, and \B disallows that.

HANDLING PERL METACHARACTERS IN SEARCHES

These Perl regular expression features seem nice, but so far we have left a gaping hole in our text-searching capabilities. If metacharacters like '(' and '+' and '.' are always special, then how can we search for a real period? How can we search for a phone number which has an area code in parentheses, if the open parenthesis mark and the close parenthesis mark are special? Perl lets us use the backslash '\' to mark these metacharacters as regular characters. In Perl this is often called 'quoting' the special character.

```
/(\d\d\d) \d\d\d-\d{4}/
```

This matches three digits, a space, three more digits, a dash, and four more digits. But no real parentheses can be in the matched string. The parentheses are still metacharacters here. They mark the first three numbers as a group to be 'captured' for later processing.

```
/\(\d\d\d\) \d\d\d-\d{4}/
```

Now the opening and closing parentheses are to be treated as real parentheses, not special regular expression features. This matches three digits enclosed in parentheses, a space, three more digits, a dash, and four more digits.

```
/M\.+x/
```

This matches an 'M', followed by one or more real periods, followed by an 'x'. So a string like 'asdfM...xcvb' would match. The string has a capital 'M', then a sequence of periods, then an 'x' in it.

PRXMATCH WHEN QUOTING METACHARACTERS

If we want to search our database for phone numbers of the form (xxx) xxx-xxxx, where there may or may not be a space between the area code and the 7-digit number, then we need to be able to treat the parentheses marks as normal characters, rather than special characters. So the form above, with an optional space in the middle, will do what we need. Our regular expression will look like `/\(\d{3}\) ?\d{3}-\d{4}/` where the '?' character lets us search for zero or one spaces in the middle.

```
data check_num;
  set prx.checkfile;
  if _n_=1 then do;
    retain re;
    re = prxparse('/\(\d{3}\) ?\d{3}-\d{4}/');
    if missing(re) then do;
      putlog 'ERROR: regex is malformed';
      stop;
    end;
  end;

  if prxmatch(re,phonenum);
run;

proc print data=check_num;
  var phonenum lastname firstname;
  title 'Phone number matches (xxx) ?xxx-xxxx ';
run;
```

The output from this program looks like:

| Obs | phonenum | lastname | firstname |
|-----|----------------|----------|-----------|
| 1 | (909) 555-3389 | Marx | Groucho |
| 2 | (404) 555-9977 | Marx | Karl |
| 3 | (664) 555-8574 | Ma rx | Zeppo |
| 4 | (703) 555-6732 | Matrix | Jon |
| 5 | (854) 555-8493 | MacDuff | Killegan |
| 6 | (646) 555-3324 | Potter | Lily |
| 7 | (646) 555-3324 | Potter | James |

CHARACTER CLASSES

One more important feature of Perl regular expressions is the 'character class'. Perl will let us list inside square

brackets a whole series of possible choices for a single character.

```
/[cbr]at/
```

The character class here has the three letters 'c', 'b', and 'r'. So only those three letters are feasible matches immediately before the 'at'. Thus, this will match 'cat', 'bat', and 'rat'. But it will not match 'mat' or 'gat' or '_at'. Only the characters listed inside the square brackets can match that single character in the pattern.

```
/[a-z]at/
```

The character class allows a dash in the same way that SAS variables can be listed in sequence using a dash between the first and last names in the sequence. The 'a-z' inside a character class means all letters between 'a' and 'z', but nothing else. So this represents all lower-case letters. This pattern will match 'bat' and 'gat' and 'pat', but not 'Mat' or 'Pat' or '3at'.

```
/[A-R1-3-]at/
```

Now the character class contains all capital letters from 'A' to 'R', the numbers from '1' to '3', and a dash. Since the dash has a special meaning inside the character class (and not anywhere else in Perl regular expressions), we cannot use a dash as a normal character when it is between characters. The way around this is to make the dash have its normal meaning if it is at the beginning or end of the string of characters inside the brackets. This pattern will match 'Bat' and 'Rat' and '2at' and even '-at', but not 'sat' or 'Sat'.

```
/[^A-R1-3-]at/
```

The caret '^' as the first character in a character class has a special meaning as well. It means 'everything *except* what is inside the brackets'. So this now matches 'sat' and 'Sat' ('s' and 'S' are not in the list of characters), and will not match 'Bat' or 'Rat' or '2at' or '-at' (those characters that are in the list).

THE PERL SUBSTITUTION FUNCTION AND PRXCHANGE

As we mentioned before, Perl has a substitution function, which looks like :

```
s/pattern to match/changed text to insert instead/
```

When we use the substitution function in a SAS function like PRXCHANGE, we must remember to enclose it in quotes, and to put the quoted string in parentheses, just as we did in the code examples above. The simplest form of the PRXCHANGE function looks like:

```
call prxchange(pattern, n, variablename);
```

Here the 'pattern' is the regular expression previously built using PRXPARSE, 'n' is the number of times to search for and replace the pattern, and 'variablename' is the SAS variable on which to work. 'n' can be any positive integer, or -1. The -1 tells SAS to match and replace every occurrence of the pattern in that string.

There are optional parameters you can use in the CALL PRXCHANGE function. However, since the parameters are not named, you have to add them in sequentially. If you want to use the last parameter, you have to include all of them. So the optional forms look like:

```
call prxchange(pattern, n, variablename, newvariablename);
```

```
call prxchange(pattern, n, variablename, newvariablename, length);
```

```
call prxchange(pattern, n, variablename, newvariablename, length, trunc);
```

```
call prxchange(pattern, n, variablename, newvariablename, length, trunc,
changecount);
```

'NEWVARIABLENAME' names a new SAS variable in which to insert the substituted version of the text. This can keep you from accidentally destroying your old (but correct) strings if your regex fails to work as you expected.

'LENGTH' names a variable that is assigned the length of the string after the replacement. 'TRUNC' names a variable that is assigned a 0 or 1, depending on whether the new string is so long that its tail is truncated when it is put back in the variable. A truncation causes the variable to be assigned a 1. This makes for easy checking if truncation would be an error condition. 'CHANGECOUNT' names a variable that is assigned the number of replacements made in the string.

Now let us make two changes in an earlier regex. First, we want to catch any case where the name starts with 'Mac' or 'Mc' and has a space, followed by the rest of the name – which is always 'word' characters in our database. So we could write the pattern as `/Ma?c \w+/` to find the problem. But we want to 'capture' the parts before and after the space, and keep only those for our substitution. By putting these parts of the pattern in parentheses, Perl captures these chunks to memory for later use. Perl remembers these as `\1`, `\2`, etc, where the number is based on the order in which the left parentheses are seen in the pattern. So we can substitute and remove the space with the function

```
s/(Ma?c) (\w+)\1\2/
```

Second, let us find any value of last name which starts at the beginning of the string with one or more spaces, then is followed by the desired name. We will capture the name, including spaces if any, and substitute so that we have no spaces at the beginning. We can do that as

```
s/^\s+(\w.*)\1/
```

This describes the pattern as beginning of line, followed by one or more 'space' characters, followed by a 'word' character and whatever follows. This allows us to have non-word characters, like a space, in the captured chunk. So the code looks like:

```
data check_change;
  set prx.checkfile;
  if _n_=1 then do;
    retain re1 re2;
    re1 = prxparse('s/(Ma?c) (\w+)\1\2/');
    re2 = prxparse('s/^\s+(\w.*)\1/');
    if missing(re1) or missing (re2) then do;
      putlog 'ERROR: a regex is malformed';
      stop;
    end;
  end;

  call prxchange(re1,-1,lastname);
  call prxchange(re2,-1,lastname);
run;

proc print data=check_change;
  var lastname firstname;
  title 'fixed last names with incorrect spaces';
run;
```

Note that we have used more than one regular expression in the step. You are not restricted to a single regular expression in any DATA step. However, you should be aware that these regular expressions take up space in memory, so over-use of multiple regular expressions in a DATA step has drawbacks. In this example, the '-1' is not really necessary. Both cases have been designed so that only one substitution is done per record. A '1' could have been used as the second parameter in both PRXCHANGE functions. However, using the '-1' will allow you to handle cases where multiple substitutions need to be performed in a single field.

The output now looks like:

| Obs | lastname | firstname |
|-----|----------|-----------|
| 1 | Marx | Chico |
| 2 | Marx | Harpo |
| 3 | Marx | Groucho |

| | | |
|----|--------------|-----------|
| 4 | Marx | Karl |
| 5 | Ma rx | Zeppo |
| 6 | Matrix | Jon |
| 7 | von Trapp | Maria |
| 8 | van den Hoff | Friedrich |
| 9 | MacDonald | Ole |
| 10 | MacDuff | Killegan |
| 11 | McMurphy | Randall |
| 12 | MacHeath | Mack |
| 13 | Potter | Lily |
| 14 | Potter | James |

We have fixed all but one of the problems we saw at the beginning of the paper, and have not messed up last names where a space should be maintained. Clearly, we could have added a third PRXPARSE line to handle our special case of 'Ma rx'. Or we could have assigned the changed strings to a new variable for further data validation. So there are lots of options when structuring changes.

CALL PRXSUBSTR

We have already noticed that the PRXMATCH function returns not a zero or 1 value, but the position at which the matching string starts. That is often enough. But, if you then want to use SUBSTR() to grab the matching text, you are lacking a length value. CALL PRXSUBSTR solves that problem. CALL PRXSUBSTR takes the form:

```
call prxsubstr (re, variable, start <, length>);
```

Note that this will now return the start value into a variable called 'START' (or whatever you name it), and can optionally assign the length of the matching string to the variable called 'LEN'. Below, we use CALL PRXSUBSTR to create the variables RE_START and RE_LEN, the starting position and length of the matching text, respectively. The regex /\w \w.+ / will match a word character, followed by a blank, followed by a word character and more characters (including the blanks at the end of the string). This will let us find all the last names with blanks in them, whether correct or not. CALL PRXSUBSTR will then return enough information that we can collect the text for later use.

Also note that we are using the SAS 9 function SUBSTRN() instead of SUBSTR() here. SUBSTRN() has the added advantage that it will gracefully handle the case where the length of the string to be gathered is 0. In that case, SUBSTRN() politely returns a missing value instead of complaining to the log.

One additional point to note is the offsets used in the SUBSTRN() function. Because we matched the desired 'tail' piece of the last name as well as the preceding space and letter, we are starting our match two characters before we want to begin the SUBSTRN() function, and our match is therefore two characters longer than desired. We remedy this by inserting the offsets below.

```
data check_sub;
  set prx.checkfile;
  if _n_=1 then do;
    retain re;
    re = prxparse('/\w \w.+ /');
    if missing(re) then do;
      putlog 'ERROR: a regex is malformed';
      stop;
    end;
  end;

  call prxsubstr(re, lastname, re_start, re_len);
  tail = substrn(lastname, re_start+2, re_len-2);
  if re_start > 0 then output;
run;

proc print data=check_sub;
  var lastname tail re_start re_len;
  title 'tail component of last names';
run;
```

This program generates the following output. Note that the TAIL variable has been designed so that the undesired part at the beginning of the regex has been dropped. There are more advanced Perl regular expression features which would actually permit us to eliminate the need for this kind of adjustment, but there simply isn't room for a complete exposition on all the regular expression features that are available.

| Obs | lastname | tail | re_start | re_len |
|-----|--------------|----------|----------|--------|
| 1 | Ma rx | rx | 2 | 14 |
| 2 | von Trapp | Trapp | 3 | 13 |
| 3 | van den Hoff | den Hoff | 3 | 13 |
| 4 | Mac Heath | Heath | 3 | 13 |

Also note that the RE_START value still shows where the regex actually began matching, and the RE_LEN value demonstrates that the regex matched all the trailing blanks as well as the desired letters.

CALL PRXPOSN

While CALL PRXSUBSTR facilitates the use of the SUBSTR() and SUBSTRN() functions, CALL PRXPOSN helps us pull out capture buffers at the specified positions in the regular expression. If we have more than one set of capturing parentheses in a regex, then we have more than one capture buffer. We may want to pull these pieces out for special use, and we may need to know which capture buffer is which. CALL PRXPOSN helps us with that task. The form of the CALL routine is:

```
call prxposn( re, capture-buffer-number, buf-start <,buf-length> );
```

The variable RE is still the return from PRXPARSE(). The CAPTURE-BUFFER-NUMBER is a number (from one up to the number of capturing parentheses in the regex) telling which buffer to use. BUF-START is the name of the variable holding the starting position for the string referenced by the given capture buffer, just as the optional BUF-LENGTH is the variable holding the length of said string.

Let's work with our phone numbers again. This time we'll set up three capture buffers, one each for the area code, the exchange, and the number end. Then we can use CALL PRXPOSN to return variables holding the start position and (optionally) the length of the matching string so that we can catch the strings using SUBSTR() .

```
data check_posn;
  set prx.checkfile;
  if _n_=1 then do;
    retain re;
    re = prxparse('/\((\d{3})\) ?(\d{3})-(\d{4})/');
    if missing(re) then do;
      putlog 'ERROR: regex is malformed';
      stop;
    end;
  end;

  if prxmatch(re,phonenum) then do;
    call prxposn(re,1,areastart,arealen);
    areacode = substr(phonenum,areastart,arealen);
    call prxposn(re,2,exchstart,exchlen);
    exchange = substr(phonenum,exchstart,exchlen);
    call prxposn(re,3,endgstart);
    ending = substr(phonenum,endgstart,4);
    output;
  end;
run;

proc print data=check_posn;
  var phonenum lastname areacode exchange ending;
  title 'Phone number matching (xxx) ?xxx-xxxx ';
```

```
run;
```

Note that in the third CALL PRXPOSN line, we didn't bother to get the optional length. When we already know the length to be returned, we don't have to look for the length in a new variable. So the last SUBSTR() function uses a constant for the length. This could have been done in any of the three calls above, but we used the length variables for illustrative purposes. The resulting output looks like:

| Obs | phonenum | lastname | areacode | exchange | ending |
|-----|----------------|----------|----------|----------|--------|
| 1 | (909) 555-3389 | Marx | 909 | 555 | 3389 |
| 2 | (404) 555-9977 | Marx | 404 | 555 | 9977 |
| 3 | (664) 555-8574 | Ma rx | 664 | 555 | 8574 |
| 4 | (703) 555-6732 | Matrix | 703 | 555 | 6732 |
| 5 | (854) 555-8493 | MacDuff | 854 | 555 | 8493 |
| 6 | (646) 555-3324 | Potter | 646 | 555 | 3324 |
| 7 | (646) 555-3324 | Potter | 646 | 555 | 3324 |

CALL PRXNEXT

The CALL PRXNEXT routine becomes valuable when we have a regex which matches multiple times (or an unknown number of times) in our string of interest. Then we can use the call routine to get the *next* occurrence of the matching pattern. The syntax of CALL PRXNEXT is:

```
call prxnext( re, search-start, search-stop, varname, re_posn, re_length);
```

There are no optional variables this time. SEARCH-START and SEARCH-STOP let us define starting and stopping positions within which to search for matches. If you want to look through the entire length of the variable VARNAME, you set SEARCH-START to 1 and SEARCH-STOP to either -1 (the end of the variable) or to LENGTH(VARNAME) (the end of the variable, not counting trailing blanks).

We now want to search through a database which has our earlier last names and first names, along with a long string which may contain one or more ID codes. Legal ID codes will be defined as having length 4 or 5 only. Here's what our data set looks like:

| | | |
|--------------|-----------|-----------------------|
| Marx | Chico | 4546 |
| Marx | Harpo | 5474 or 13373 or 1774 |
| Marx | Groucho | no idcodes |
| Marx | Karl | 4385 and 49387 |
| Marx | Zeppo | 6485 |
| Matrix | Jon | 77553 24653 |
| von Trapp | Maria | 2263 |
| van den Hoff | Friedrich | 548236842 |
| MacDonald | Ole | 90510 |
| MacDuff | Killegan | @NA |
| McMurphy | Randall | 227 |
| MacHeath | Mack | . |
| Potter | Lily | 46455 |
| Potter | James | 46454 |

Note that some of the strings have unexpected notes, and some have more than one ID code. Also note that the eighth row has a number too long to be legal, while the eleventh record has a number that is too short. We can handle both those problems with a regex which only allows 4 or 5 digits. `/\b\d{4,5}\b/` uses the `\b` metacharacter to mark the beginning and end of the number sequence (remember that digits count as 'word' characters, so we can use `\b` to spot the beginning and end of digit strings too). In between the `\b` characters, the only regex which will match will be a sequence of exactly four or five digits. Our program becomes:

```
data check_next;
  array ids{3} id1-id3;
  set prx.checkfile2;
```

```

if _n_=1 then do;
  retain re;
  re = prxparse('/\b\d{4,5}\b/');
  if missing(re) then do;
    putlog 'ERROR: a regex is malformed';
    stop;
  end;
end;

start = 1;
stop = length(idcodes);
call prxnext(re,start,stop,idcodes,startposn,len);
do i = 1 to 3 while(startposn > 0);
  ids{i} = input(substr(idcodes,startposn,len),5.);
  call prxnext(re,start,stop,idcodes,startposn,len);
end;
run;

proc print data=check_next;
var lastname idcodes id1 id2 id3;
title 'legal ID codes from the string IDCODES';
run;

```

Note that the array will only hold three IDs. A longer array would need to be defined in some cases. When CALL PRXNEXT fails to find another matching occurrence, it will return a zero for the start position of the match. At that point, the do-loop will stop and the remaining entries in the array will be left with a missing value. So the output looks like:

| Obs | lastname | idcodes | id1 | id2 | id3 |
|-----|--------------|-----------------------|-------|-------|------|
| 1 | Marx | 4546 | 4546 | . | . |
| 2 | Marx | 5474 or 13373 or 1774 | 5474 | 13373 | 1774 |
| 3 | Marx | no idcodes | . | . | . |
| 4 | Marx | 4385 and 49387 | 4385 | 49387 | . |
| 5 | Marx | 6485 | 6485 | . | . |
| 6 | Matrix | 77553 24653 | 77553 | 24653 | . |
| 7 | von Trapp | 2263 | 2263 | . | . |
| 8 | van den Hoff | 548236842 | . | . | . |
| 9 | MacDonald | 90510 | 90510 | . | . |
| 10 | MacDuff | @NA | . | . | . |
| 11 | McMurphy | 227 | . | . | . |
| 12 | MacHeath | . | . | . | . |
| 13 | Potter | 46455 | 46455 | . | . |
| 14 | Potter | 46454 | 46454 | . | . |

The regex successfully parses out one legal ID code each time, until there are none left. Note that the uses of the parsing function allows us to avoid worrying about what sort of separators or words are used between the ID codes. Another point of interest is that the program used the INPUT() function to turn the substrings (which are still character strings) into numeric variables, thus making them easier to work with in some circumstances. You can see from the alignment of the variables above that the first two are character and the last three are numeric.

THE PRXPAREN FUNCTION

The PRXPAREN function is less common than the previous functions and CALL routines. It is useful when you have alternation with capturing parentheses, and you want to see which capture buffer was used.

```
prxparen(re)
```

is the form we use. It only requires the regex RE, although it does need PRXMATCH to be previously invoked, so that there is a matched regular expression to work with.

Suppose that we are still working with the above ID code database, and we want to capture three, four, or five-digit codes. If we use the regex

```
/\b(\d{3}\b)|(\d{4}\b)|(\d{5}\b)/
```

We will match the beginning of the digit sequence, three or four or five digits, and the end of the sequence. Note that each of the three alternates is enclosed in parentheses. Which will match on any given record? Since a regular expression matches leftmost first, we won't be surprised that this will match the lefthand ID code (of three to five digits). So we can use PRXPAREN() to see which of the alternates matched.

```
data check_paren;
  set prx.checkfile2;
  if _n_=1 then do;
    retain re;
    re = prxparse('/\b(\d{3}\b)|(\d{4}\b)|(\d{5}\b)/');
    if missing(re) then do;
      putlog 'ERROR: a regex is malformed';
      stop;
    end;
  end;

  if prxmatch(re,idcodes) then do;
    match_num = prxparen(re);
    output;
  end;
run;

proc print data=check_paren;
  var lastname idcodes match_num;
  title 'matching syntax from IDCODES';
run;
```

This only outputs the matching records, as opposed to the previous output, so we get the following output. We can see that, when there are multiple ID codes, the regex matches “leftmost first”, meaning that it matches starting at the left, and once successful, does not try repeatedly to make longer matches further right in the string. When there is only one ID, that is of course matched. When there are more than one, the lefthand digit sequence is matched, whether it is the longest possible match or not.

Note that MATCH_NUM is only equal to 3 (representing the longest digit sequence) when the first matching digit sequence is of length five, in records 5, 7, 8, 10, and 11. In record 2, there is a possible five-digit match. But there is an earlier sequence that matches our regular expression, so that one will be found instead.

| Obs | lastname | idcodes | match_num |
|-----|--------------|-----------------------|-----------|
| 1 | Marx | 4546 | 2 |
| 2 | Marx | 5474 or 13373 or 1774 | 2 |
| 3 | Marx | 4385 and 49387 | 2 |
| 4 | Marx | 6485 | 2 |
| 5 | Matrix | 77553 24653 | 3 |
| 6 | von Trapp | 2263 | 2 |
| 7 | van den Hoff | 548236842 | 3 |
| 8 | MacDonald | 90510 | 3 |
| 9 | McMurphy | 227 | 1 |
| 10 | Potter | 46455 | 3 |
| 11 | Potter | 46454 | 3 |

CONCLUSION

The Perl regular expressions give us a wide variety of choices in matching patterns, even though there are really only five general principles to follow. The PRX functions and CALL routines give us handy ways of using these

regular expressions to search for complex patterns and to make changes in our text strings. The two together give us new tools that extend the utility of the DATA step.

The regular expressions available in SAS 9 are built on the functionality of Perl 5.6.1. But there are more Perl regular expression features than we have been able to show in one short paper. And there are Perl regular expression features which are not available in SAS. Details on the features not in SAS are given in the SAS Online documentation. For full details of the features available in Perl, the Perl documentation is extensive and free.

In addition, there are two more PRX CALL routines we have not mentioned. CALL PRXDEBUG(1) can be executed in order to get debugging information printed to the log. This can help debug a malfunctioning regular expression as well as help explain why (or why not) a puzzling match occurs. CALL PRXFREE(regex-name) frees up the memory allocated to *regex-name* once it is no longer needed. In a normal DATA step, this is seldom required, since the memory will be automatically freed at the conclusion of the DATA step processing.

There are more features than we can demonstrate in anything short of a textbook. But the power comes from the functionality of the Perl regular expression engine under the hood. As you practice with regular expressions, you will gain an appreciation for the breadth of applicability these functions have.

ACKNOWLEDGMENTS

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

The Perl language was designed by Larry Wall, and is available for public use under both the GNU GPL and the Perl Copyleft.

Other brand and product names are registered trademarks or trademarks of their respective companies.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. The author may be contacted by mail at:

David L. Cassell
Design Pathways
3115 NW Norwood Pl.
Corvallis, OR 97330

or by e-mail at:
DavidLCassell@msn.com