Paper SY11

The Perks of PRX…

David L. Cassell, Design Pathways, Corvallis, OR

ABSTRACT

Pattern matching lets you find chunks of text that have a particular pattern - even patterns that would be very hard to find using SAS® functions like index() and substr(). For example, try finding all strings which have two consecutive words that are the same, not counting capitalization. Difficult with traditional SAS® functions, but Perl's 'regular expressions' give you a simple language to search for patterns, extract patterns from strings, and even change text that matches your patterns. This is invaluable for all manner of text manipulation, including validation, text replacement, and string testing. With the advent of SAS® 9, the power of Perl's regular expressions is now available in the DATA step.

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).

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:

```
/* PRX1.SAS  -  match the string 'arx' */

data check1;
  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=check1;
  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 expression. The two types of Perl functions we will see are the above matching function 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 regualr 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 DATA step will 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, but 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. 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.

MORE PERL REGULAR EXPRESSION FORMS

Now that we have seen 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

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 occurences of the preceding pattern
+	matches 1 or more occurences of the preceding pattern
?	matches exactly 0 or 1 occurences of the preceding pattern
{k}	matches exactly k occurences of the preceding pattern
{n,m}	matches at least n and at most m occurences 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 files and directories), 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, since zero occurences 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. Use care when using the ‘*’ iterator!

```
/r+x/
```

This would match one or more occurences of ‘r’, followed immediately by an ‘x’. So it would match ‘rx’ and ‘rrrx’ and ‘rrrrrrrrrx’. 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.

ANOTHER CODE EXAMPLE – WILDCARDS AND ITERATORS

Now let us use these features to begin validating our database. Suppose we need to find records which have an inappropriate space: a space accidentally entered before or in the middle of the last names. We want to look for a space, followed by one or more letters. We could create a piece of SAS code similar to the above program.

```
data check2;
  set prx.checkfile;
  if _n_=1 then do;
    retain re;
    re = prxparse('/ \w+/');
```

```
    if missing(re) then do;
      putlog 'ERROR: regex is malformed';
      stop;
      end;
    end;

  if prxmatch(re,lastname);
  run;

proc print data=check2;
  var lastname firstname;
  title 'Last name matches blank plus "word" ';
  run;
```

Note that the regular expression uses concatenation (a blank followed by 'word' characters), wildcards (the \w metacharacter), and iterators (the '+' to get one or more 'word' characters). This produces the following output.

```
Obs    lastname        firstname

 1       Marx          Groucho
 2     Ma rx           Zeppo
 3     von Trapp       Maria
 4     van den Hoff    Friedrich
 5     Mac Heath       Mack
```

We can see that observations 3 and 4 are legitimate last names, and should not be altered.

MORE BASICS OF PERL REGULAR EXPRESSIONS

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.

```
/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.

EXAMPLE 3 – ALTERNATION AND GROUPING

If we now want to search for only those names starting with 'Mc' or 'Mac', followed by a space, then the rest of the name, we can modify our above code. We can use our newly-learned techniques and write the pattern as `/(Mc|Mac) \w+/` Then our program would look like:

```
data check3;
  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);
    run;

proc print data=check3;
  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. There is usually more than one way to work out a regular expression.

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) which take some getting used to.

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

The '^' and '$' indicate that the pattern must match at the beginnning 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'.

```
/^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, but not one of the characters which are represented by the \w metacharacter.

```
/^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'.

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. 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.

EXAMPLE 4 – 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 check4;
  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=check4;
  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 patter 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 are in the list).

EXAMPLE 5 – CHARACTER CLASSES

The previous example helped us find phone numbers in a particular pattern. But it did not handle the fact that legal phone numbers cannot start with a 0 or 1, and legal area codes cannot start with a 0 or 1. But we can address that problem using character classes. We can change the pattern to look like
`/\([2-9]\d{2}\) ?[2-9]\d{2}-\d{4}/` instead.

We will make one other change in our code. This time, instead of sending the results to another data set for later printing, we will use the PUTLOG statement to send the records we want directly to the log for later review. Since there is no output, we can write this using a DATA _NULL_ statement which will not create a new data set.

```
data _null_;
  set prx.checkfile;
  if _n_=1 then do;
    retain re;
      * this time, *legal* phone numbers in this format;
    re = prxparse('/\([2-9]\d{2}\) ?[2-9]\d{2}-\d{4}/');
    if missing(re) then do;
      putlog 'ERROR: regex is malformed';
      stop;
      end;
    end;

  if prxmatch(re,phonenum) then
    putlog phonenum= lastname= firstname= ;
  run;
```

This prints the results in the log:

```
phonenum=(909) 555-3389 lastname=Marx firstname=Groucho
phonenum=(404)555-9977 lastname=Marx firstname=Karl
phonenum=(664) 555-8574 lastname=Ma rx firstname=Zeppo
phonenum=(703) 555-6732 lastname=Matrix firstname=Jon
phonenum=(854)555-8493 lastname=MacDuff firstname=Killegan
```

```
phonenum=(646) 555-3324 lastname=Potter firstname=Lily
phonenum=(646) 555-3324 lastname=Potter firstname=James
```

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. Then 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.

Now let us make two changes. 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 check6;
  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=check6;
  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.

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 give us handy ways of using them to search for arcane patterns and 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.

There are also more PRX functions and CALL routines than we can demonstrate here. But their power comes from the functionality of the Perl regular expression engine under the hood. As you practice with them, 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