The Perl in the Crown: Regular Expressions in SAS

Bob Newman, Amadeus Software Limited

ABSTRACT

This paper introduces the PRX family of SAS functions and call routines, which bring the power of Perl regular expressions to SAS. Examples are given of every member of the PRX family (apart from PRXDEBUG), as well as some undocumented features. No prior knowledge of Perl is required.

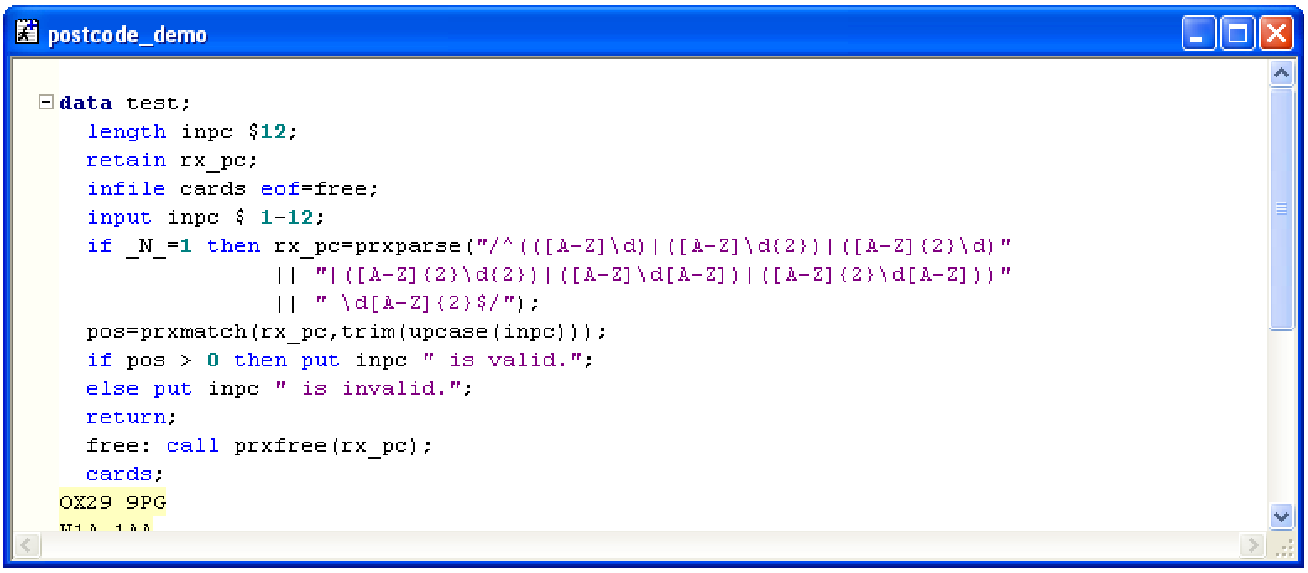
INTRODUCTION

The PRX family of functions were introduced in SAS 9, and offer powerful facilities for parsing, matching and transforming character strings using Perl regular expressions, which have become standard in the IT world. The value of the PRX functions is perhaps not fully appreciated throughout the SAS community.

The PRX functions effectively supersede a family of RX functions which provided broadly similar facilities, but used a totally different form of regular expression. The RX functions remain supported, but they are no longer documented and their use in new programs is deprecated. They are not covered in this paper.

A FIRST EXAMPLE

Here is a program demonstrating the use of the PRX functions to validate postcodes. It is not sophisticated enough for use in production applications, but it is useful as an introduction.



```
data test;
  length inpc $12;
  retain rx_pc;
  infile cards eof=free;
  input inpc $ 1-12;
  if _N_=1 then rx_pc=prxparse("/^(([A-Z]\d)|([A-Z]\d{2})|([A-Z]{2}\d)"
                 || "|([A-Z]{2}\d{2})|([A-Z]\d[A-Z])|([A-Z]{2}\d[A-Z]))"
                 || " \d[A-Z]{2}$/");
  pos=prxmatch(rx_pc,trim(upcase(inpc)));
  if pos > 0 then put inpc " is valid.";
  else put inpc " is invalid.";
  return;
  free: call prxfree(rx_pc);
  cards;
OX29 9PG
```

First the PRXPARSE function is used to parse a regular expression, which we retain in variable RX_PC. Then we use the PRXMATCH function to see which of the input strings match RX_PC. Notice that the expression is parsed once only, and at end-of-file PRXFREE is called to release the memory it used. It's all very straightforward, apart of course from understanding that rather intimidating regular expression – which is here split into three substrings, purely for the sake of readability. But don't panic…..

- The "/" characters just mark the beginning and end of the expression.
- "^" denotes the beginning of the input string, and "$" the end of it. These are called "anchors". Because "$" is used, the TRIM function is needed later to eliminate trailing white space.
- The first two substrings together contain, in brackets, a list of six alternatives for the first part of the postcode – the "outward code". The "|" character means "or". Each of the six alternatives has its own set of brackets. The first alternative is "([A-Z]\d)" which means "one letter followed by one digit". The last is "([A-Z]{2}\d[A-Z])" which means "two letters followed by a digit and a letter".
- The third substring specifies a space followed by a digit and two letters i.e. the second part of the postcode, the "inward code".

It is not unusual for regular expressions to look as complicated and hostile as this one may have done at first sight, but they are quite docile once you get to know them, and they will enable you to do a lot of things that would be far more trouble to do any other way. An undocumented feature given later in this paper shows how they can be laid out in ways that make them easier to understand.

The output from this program tells us:

```
OX29 9PG  is valid.
W1A 1AA  is valid.
pe69bd  is invalid.
RG126HE  is invalid.
bs7  is invalid.
ip1   3bh  is invalid.
XY7 3ZZ  is valid.
GIR 0AA  is invalid.
FIQQ 1ZZ  is invalid.
BFPO 324  is invalid.
SAN TA  is invalid.
```

Some of the postcodes have been rejected because they do not contain a space in the middle, and one of them because it has too many spaces. “XY7 3ZZ” has been accepted, since it is correctly formatted and the program is not clever enough to know that there is no such posttown as “XY”. The last four have been rejected, despite being valid, but they are all unusual types of postcode that the program does not check for (Girobank, Falkland Islands, British Forces, and Father Christmas). More detailed validation of the six standard types is also possible; for example, certain letters can never occur in certain positions. Enhancing the program to address all these issues is left as an exercise for the reader.

CHARACTER CLASSES

INTRODUCING CHARACTER CLASSES

The above “postcodes” example made frequent use of “\d” to specify a single digit, and “[A-Z]” to indicate a single (upper case) letter. These are both examples of *character classes*. “[A-Z]” is a user-defined character class; “\d” is a pre-defined one. (“Character class” is the standard Perl terminology, no longer used in the SAS documentation.)

PRX stands for “Perl Regular eXpression”, and the Perl language comes from the wonderful world of Unix, where lower case is good and upper case is quite different. You should not therefore be surprised to learn that, while “\d” indicates a digit, “\D” indicates “any character other than a digit”.

Other classes that can be specified using similar syntax are:

- “\w” indicates a “word character” i.e. a-z, A-Z, 0-9 or underscore
- “\W” any character other than a “word character”
- “\s” indicates a “white space” character – space, tab, LF, FF etc.
- “\S” any character other than white space
- “\t” indicates a tab character

ESCAPING

You also need to know that “\” can be used *anywhere within a regular expression* as an “escape” character, to prevent the next character from having any special effect. Within square brackets (which we are about to start using), there are only 5 characters that have special effects; these are “]” , “^” , “$” , “-” and “\”.

USER-DEFINED CHARACTER CLASSES

User-defined character classes can be specified by putting a list of characters between square brackets. “-“ can be used to indicate a range, and “^” for negation (though only if it is the first character within the square brackets).

Here are some examples of user-defined character classes using all of these features. Where “\” acts as the “escape” character, it is highlighted here in red.

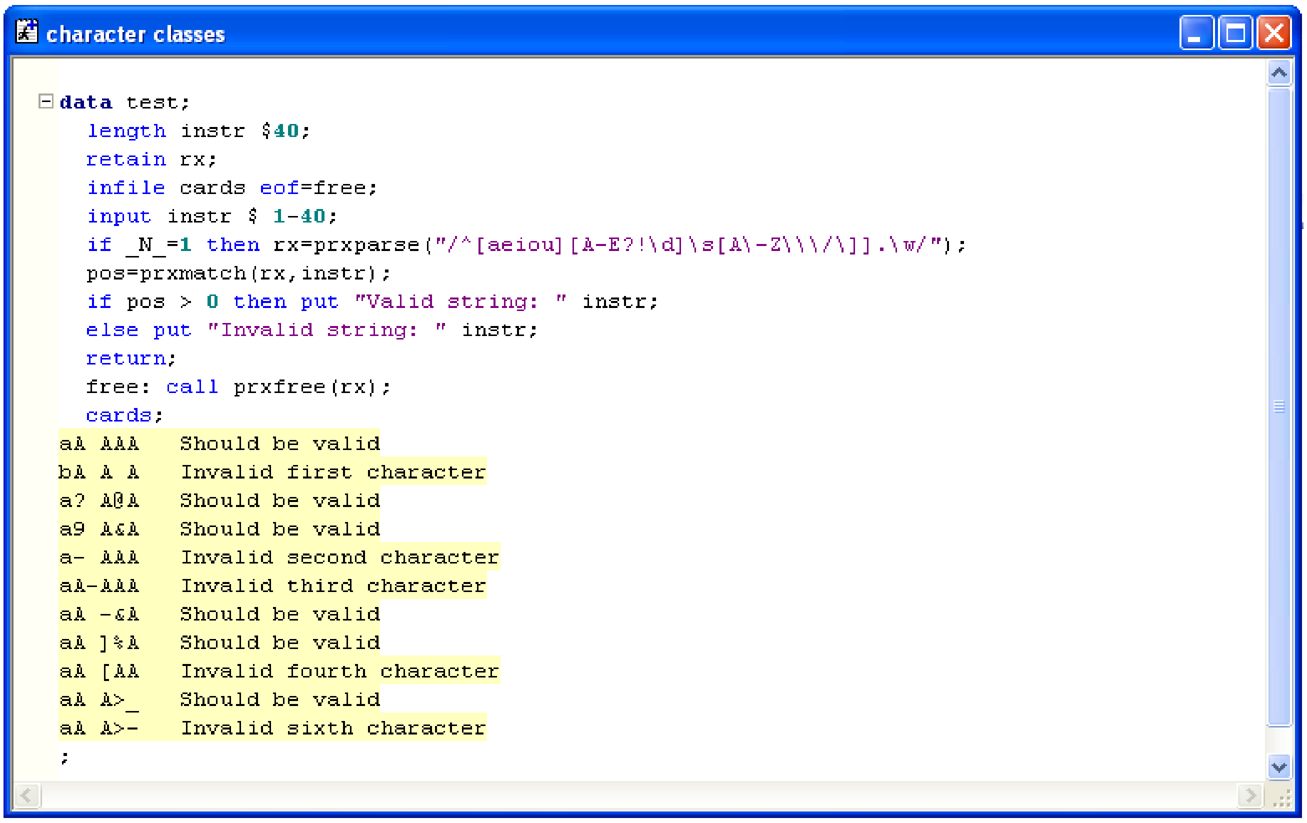
- [aeiou] matches any lower-case vowel
- [A-E?!\d] matches any of 5 upper case letters, question mark, exclamation mark, or any digit
- [^XYZ] matches any character other than upper case X, Y or Z.
- [\^XYZ] or [X^YZ] matches “^”, X, Y or Z.
- [A\-Z\\/\]] matches A, “-“, Z,”\”, “/” or “]”

Recall that we have already seen “^” used as an “anchor” denoting the beginning of the input string. Do not be confused by this character having a quite different meaning when found between square brackets.

WILD CHARACTERS

One more easy piece of syntax before we give an example of all this: “.” is the “wild character”, denoting any single character. Formally speaking, it is a character class that matches any character except “new line”.

CHARACTER CLASSES EXAMPLE, USING PRXMATCH



```
data test;
  length instr $40;
  retain rx;
  infile cards eof=free;
  input instr $ 1-40;
  if _N_=1 then rx=prxparse("/^[aeiou][A-E?!\d]\s[A\-Z\\\/\]].\w/");
  pos=prxmatch(rx,instr);
  if pos > 0 then put "Valid string: " instr;
  else put "Invalid string: " instr;
  return;
  free: call prxfree(rx);
  cards;
aA AAA   Should be valid
bA A A   Invalid first character
a? A@A   Should be valid
a9 A&A   Should be valid
a- AAA   Invalid second character
aA-AAA   Invalid third character
aA -&A   Should be valid
aA ]%A   Should be valid
aA [AA   Invalid fourth character
aA A>_   Should be valid
aA A>-   Invalid sixth character
;
```

This sample program uses three of the user-defined classes we considered earlier, together with "\s", "\w" and ".". Our regular expression uses "^" to anchor to the beginning of the input string, but this time we have not used "$" to anchor to the end of the input string. The effect of this is that we are checking the first 6 characters of the input string, but don't care about anything that comes later. In our test data, what comes later is text describing the verdict we expect on the first 6 characters. (In every case, we are right!)

REPEAT COUNTS

REPEAT COUNT SYNTAX

Repeat counts are one of the most important components of PRX regular expressions. The list of repeat specifiers looks like this (although it is not quite complete – there are some useful undocumented features in this area, which will be covered separately later).

*	means "any number of times". Warning: "any number" includes zero!
+	means "one or more times".
?	means "zero or one times".
{n}	means "exactly n times" (n being a positive integer).
{n,}	means "n or more times".
{m,n}	means "between m and n times".

The repeat count always follows the character (or character class, or subexpression) it refers to.

For example, an expression that would match all standard UK postcodes (while not validating them very strictly) is:

"/[A-Z]{1,2}\d{1,2}[A-Z]?\s+\d[A-Z]{2}/"

meaning "either one or two letters, then either one or two digits, then optionally another letter, then one or more white space characters followed by a digit and exactly two letters". The repeat counts are highlighted here in red.

THE GOLDEN RULE OF PATTERN MATCHING

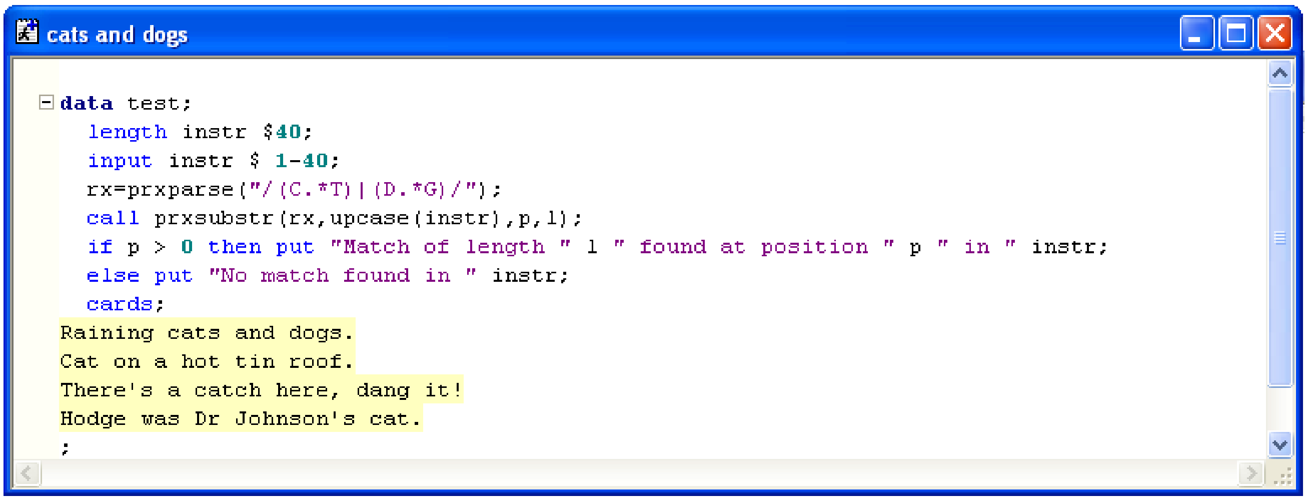
In the absence of any anchors ("^" and "$" which tie the expression to the beginning and/or end of the input string), PRXMATCH scans the entire input string looking for a match for its regular expression. The rule is:

- It will return the earliest match it can find.
- Of matches beginning equally early in the string, it will return the longest. (This is known as "greedy" matching.)

NB The undocumented features described later include a way of modifying this rule, so that "non-greedy" matching is used instead.

EXAMPLE USING PRXSUBSTR

Previous examples have used the PRXMATCH function, which returns the position of the match found (or 0 if none). This next one uses instead the PRXSUBSTR call routine, which does exactly the same as PRXMATCH except that it also returns the length of the matching substring.



```
data test;
  length instr $40;
  input instr $ 1-40;
  rx=prxparse("/(C.*T)|(D.*G)/");
  call prxsubstr(rx,upcase(instr),p,l);
  if p > 0 then put "Match of length " l " found at position " p " in " instr;
  else put "No match found in " instr;
  cards;
Raining cats and dogs.
Cat on a hot tin roof.
There's a catch here, dang it!
Hodge was Dr Johnson's cat.
;
```

The output is:

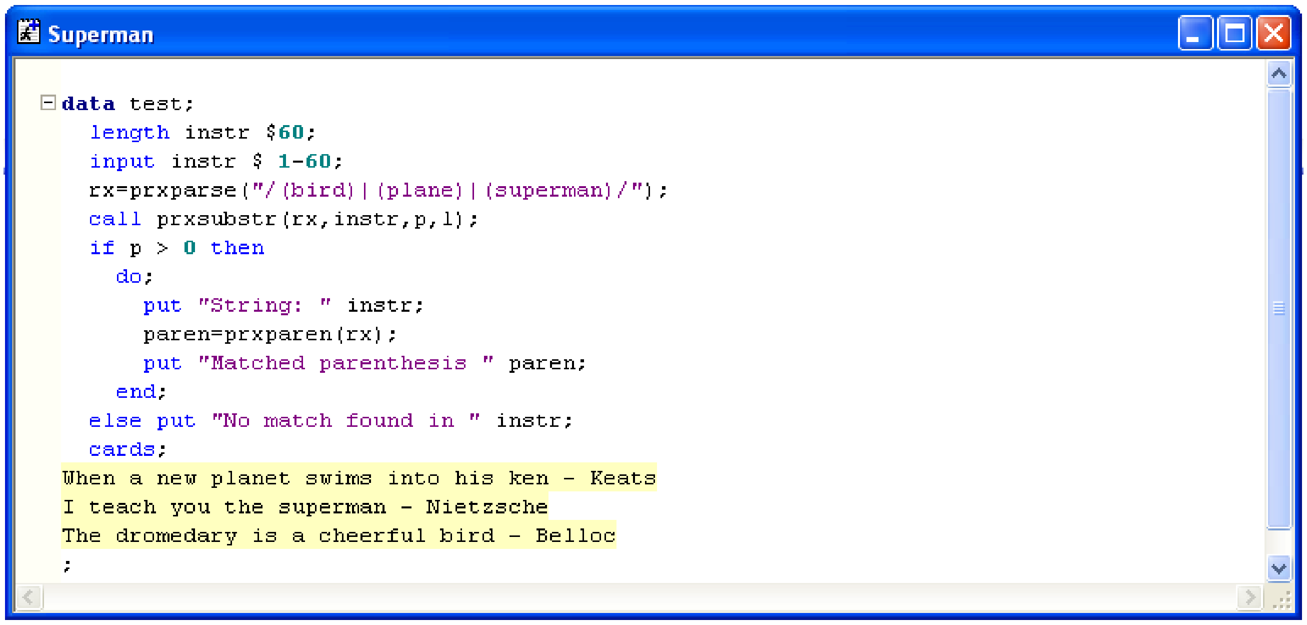
```
Match of length 3  found at position 9  in Raining cats and dogs.
Match of length 14  found at position 1  in Cat on a hot tin roof.
Match of length 19  found at position 11  in There's a catch here, dang it!
Match of length 2  found at position 3  in Hodge was Dr Johnson's cat.
```

Recall that “|” means “or”, so this regular expression is looking for a string that *either* begins in C and ends in T, *or* begins in D and ends in G. In the first example the “cat” comes before the “dog”, so it is the match returned. In the second example, the longest of the matches beginning with the C of “cat” is that one that ends with the T of “tin”. Something similar happens with the third example. In the fourth example, our matching string is the “dg” of “Hodge” – recall that the “*” means it’s OK to have *zero* characters between the D and the G. This time there is a longer match – “cat” – appearing later in the string, but this is ignored.

ANALYSING MATCHES

USING PRXPAREN TO FIND OUT WHICH SUBEXPRESSION WAS MATCHED

That “cats and dogs” example showed how to check whether the input string matched either of two alternatives. We saw how the PRXSUBSTR call routine could tell us exactly what substring of the input gave the match - but PRXSUBSTR did not actually tell us which of the two alternatives we had found a match for. There is another function that can do this: PRXPAREN.



```
data test;
  length instr $60;
  input instr $ 1-60;
  rx=prxparse("/(bird)|(plane)|(superman)/");
  call prxsubstr(rx,instr,p,l);
  if p > 0 then
    do;
      put "String: " instr;
      paren=prxparen(rx);
      put "Matched parenthesis " paren;
    end;
  else put "No match found in " instr;
  cards;
When a new planet swims into his ken - Keats
I teach you the superman - Nietzsche
The dromedary is a cheerful bird - Belloc
;
```

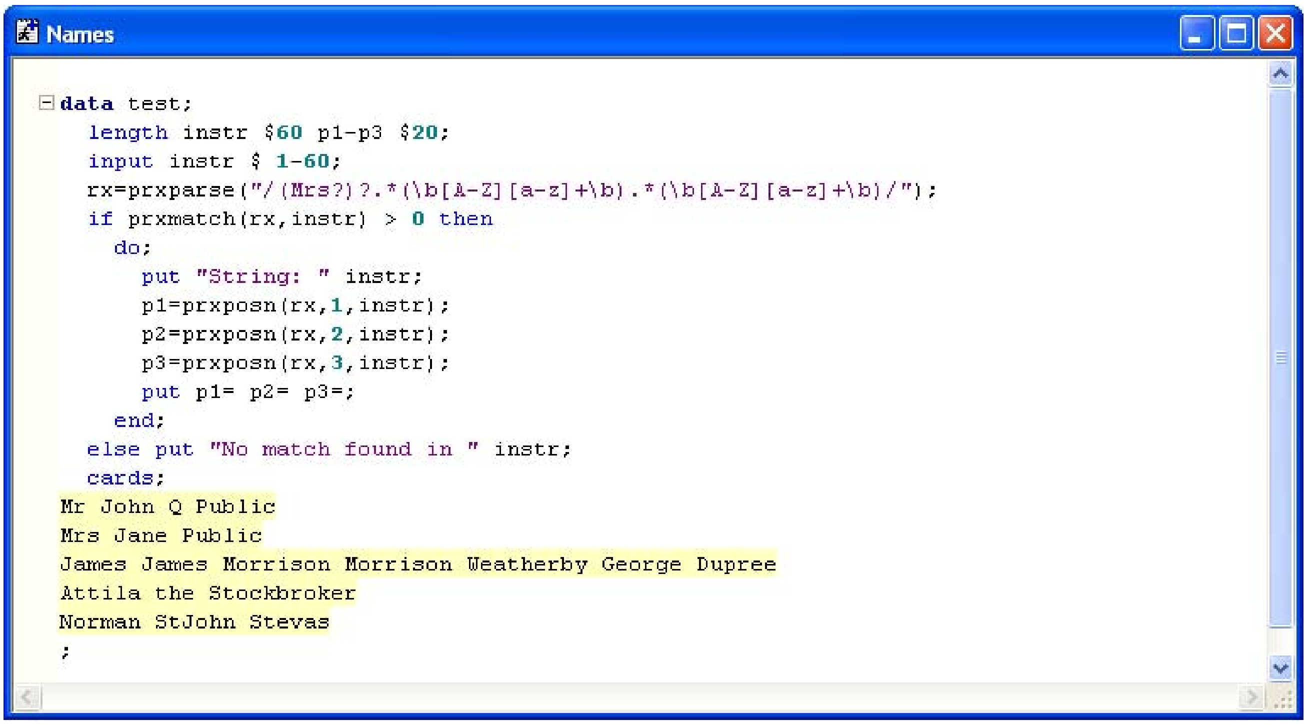
The output is:

```
String: When a new planet swims into his ken - Keats
Matched parenthesis 2
String: I teach you the superman - Nietzsche
Matched parenthesis 3
String: The dromedary is a cheerful bird - Belloc
Matched parenthesis 1
```

USING PRXPOSN TO FIND WHICH PART OF THE INPUT STRING MATCHED A SUBEXPRESSION

Parts of the expression that are enclosed in parentheses, like “(bird)” and “(plane)” above, are called *subexpressions*. Another function, PRXPOSN, can tell us what substring of the input matched a particular subexpression.

The following example uses a useful piece of new syntax: “\b” indicates a “word boundary”. (“\B”, of course, means “not a word boundary”.)



```
data test;
  length instr $60 p1-p3 $20;
  input instr $ 1-60;
  rx=prxparse("/(Mrs?)?.*(\b[A-Z][a-z]+\b).*(\b[A-Z][a-z]+\b)/");
  if prxmatch(rx,instr) > 0 then
    do;
      put "String: " instr;
      p1=prxposn(rx,1,instr);
      p2=prxposn(rx,2,instr);
      p3=prxposn(rx,3,instr);
      put p1= p2= p3=;
    end;
  else put "No match found in " instr;
  cards;
Mr John Q Public
Mrs Jane Public
James James Morrison Morrison Weatherby George Dupree
Attila the Stockbroker
Norman StJohn Stevas
;
```

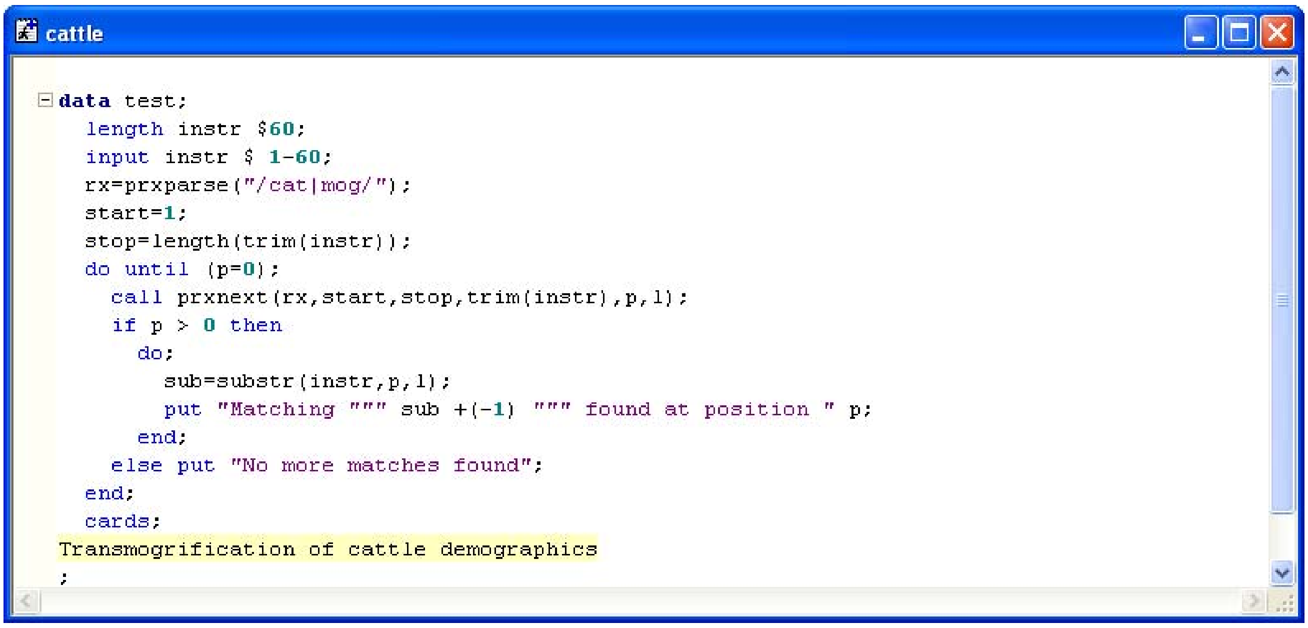
```
String: Mr John Q Public
p1=Mr p2=John p3=Public
String: Mrs Jane Public
p1=Mrs p2=Jane p3=Public
String: James James Morrison Morrison Weatherby George Dupree
p1=  p2=George p3=Dupree
String: Attila the Stockbroker
p1=  p2=Attila p3=Stockbroker
String: Norman StJohn Stevas
p1=  p2=Norman p3=Stevas
```

This regular expression contains three subexpressions. The first (for which the matching substring is stored in P1) matches “Mr” or “Mrs”. The first “?” says that the “s” is optional; the second says that the whole subexpression is optional. The others (P2 and P3) each match a name, consisting of an upper-case letter followed by one or more lower-case letters, and both beginning and ending at a word boundary. Since the PRX functions return (by default) the longest possible matches, the names returned in P2 and P3 are the last two in the string. Neither “Q” nor “the” nor “StJohn” gives a match, the first because it contains no lower-case letters, the second because it is not capitalised and the third because of the capital “J” in the middle.

MATCHES AND REPLAYS

USING PRXNEXT FOR REPEATED MATCHING

The PRXNEXT function can be called repeatedly to find successive matches for a regular expression within a source string. In this example we look for all occurrences of “cat” or “mog”.



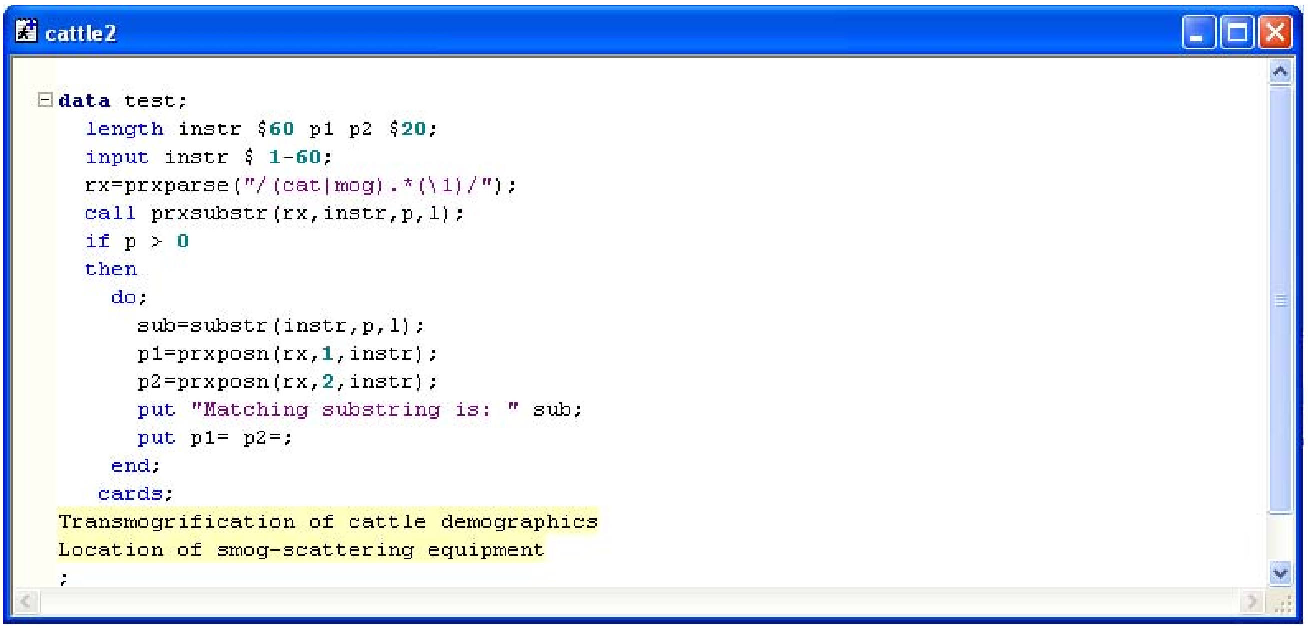
```
data test;
  length instr $60;
  input instr $ 1-60;
  rx=prxparse("/cat|mog/");
  start=1;
  stop=length(trim(instr));
  do until (p=0);
    call prxnext(rx,start,stop,trim(instr),p,l);
    if p > 0 then
      do;
        sub=substr(instr,p,l);
        put "Matching """ sub +(-1) """ found at position " p;
      end;
    else put "No more matches found";
  end;
  cards;
Transmogrification of cattle demographics
;
```

START and STOP variables are initialised with the positions of the beginning and end of the string, and PRXNEXT is called repeatedly till no more matches are found. The results are:

```
Matching "mog" found at position 6
Matching "cat" found at position 13
Matching "cat" found at position 23
Matching "mog" found at position 32
No more matches found
```

REFERRING TO THE MATCHES FOR EARLIER SUBEXPRESSIONS

Now here is a program that does something slightly different. We look for the first occurrence of either “cat” or “mog”, and then for the first recurrence of that, *whichever it was*.



```
data test;
  length instr $60 p1 p2 $20;
  input instr $ 1-60;
  rx=prxparse("/(cat|mog).*(\1)/");
  call prxsubstr(rx,instr,p,l);
  if p > 0
  then
    do;
      sub=substr(instr,p,l);
      p1=prxposn(rx,1,instr);
      p2=prxposn(rx,2,instr);
      put "Matching substring is: " sub;
      put p1= p2=;
    end;
   cards;
Transmogrification of cattle demographics
Location of smog-scattering equipment
;
```

The new syntax here is “\1” which means “the text that matched subexpression 1”. Here we display the entire matching substring, and also the matches for the two subexpressions. The output from this program is:

```
Matching substring is: mogrification of cattle demog
p1=mog p2=mog
Matching substring is: cation of smog-scat
p1=cat p2=cat
```

This syntax greatly extends the range of patterns that can be handled e.g. the regular expression "/([A-Za-z])\1/" matches the first doubled letter in a string - though NB in "Llanelli" it would match the *second* pair of "l"s , since the first pair differ in case.

ALTERING STRINGS USING PRXCHANGE

The PRXCHANGE call routine enables text strings to be modified, at times in quite clever ways. It uses a different form of regular expression from those we have seen before. Whereas PRXMATCH and the others use expressions of the form:

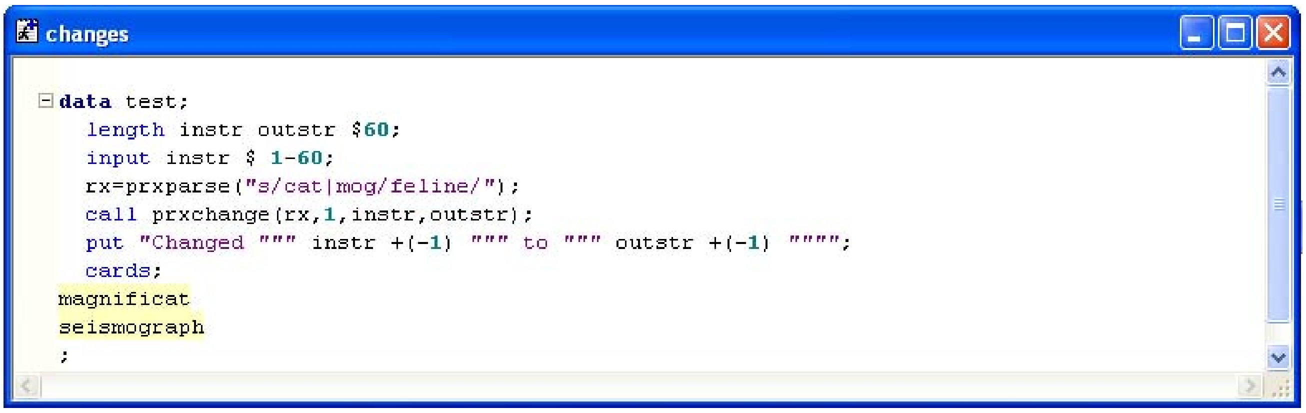
"/pattern/"

PRXCHANGE regular expressions are of the form:

"s/old/new/"

(Actually in both cases a character other than "/" can be used as delimiter if you prefer.)

Here is a simple example, in which cats and mogs are changed to felines:



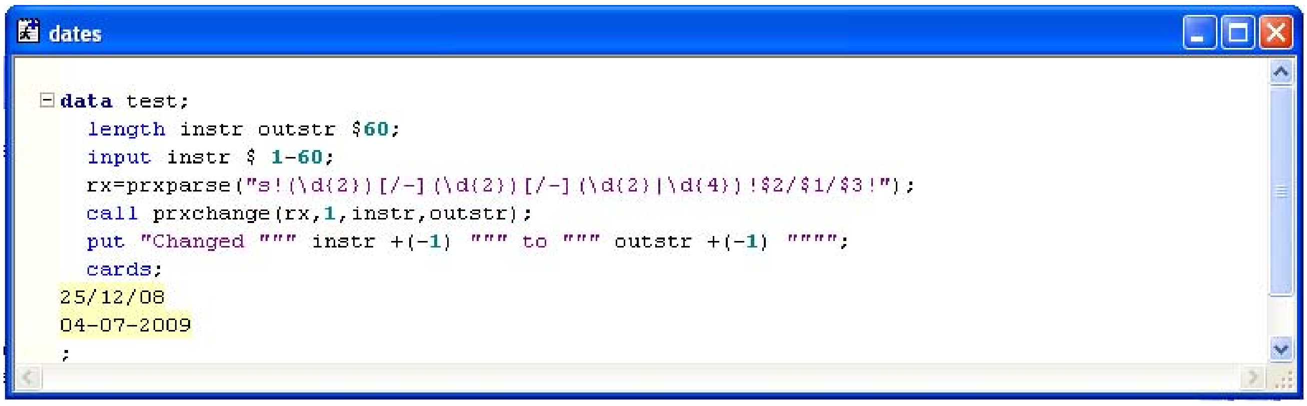
```
changes

data test;
   length instr outstr $60;
   input instr $ 1-60;
   rx=prxparse("s/cat|mog/feline/");
   call prxchange(rx,1,instr,outstr);
   put "Changed """ instr +(-1) """ to """ outstr +(-1) """";
   cards;
magnificat
seismograph
;
```

The second parameter, here set to 1, is the number of times the change is to be applied. A value of -1 for this parameter would mean "as many times as possible". The output from the program is:

```
Changed "magnificat" to "magnifeline"
Changed "seismograph" to "seisfelineraph"
```

What makes matters more interesting is the ability to substitute into a target string the values that matched subexpressions. Here, for example, is a program that switches between British and American date formats:



```
dates

data test;
   length instr outstr $60;
   input instr $ 1-60;
   rx=prxparse("s!(\d{2})[/-](\d{2})[/-](\d{2}|\d{4})!$2/$1/$3!");
   call prxchange(rx,1,instr,outstr);
   put "Changed """ instr +(-1) """ to """ outstr +(-1) """";
   cards;
25/12/08
04-07-2009
;
```

The output is:

```
Changed "25/12/08" to "12/25/08"
Changed "04-07-2009" to "07/04/2009"
```

Notice first that "!" has been chosen as delimiter, since "/" will be appearing elsewhere in the expression (and using "\" to escape "/" can lead to insanity). The first two subexpressions – the parts in (parentheses) - are two-digit numbers; the third is the year, which can be either two or four digits. These subexpressions are separated by either slashes or hyphens. In the part of the regular expression that builds the new string, the matching substrings can be substituted in as $1, $2 and $3 respectively.

Recall that in a PRXMATCH regular expression, the syntax for referring to "the text that matched subexpression 1" was "\1". When building a new string in a PRXCHANGE regular expression, the syntax for the same thing is "$1". So, if we wanted to ensure that the separator within the date string – either "/" or "-" – was the same in both places, and reproduce it in the output, we could use the slightly fiercer-looking expression:

```
"s!(\d{2})([/-])(\d{2})\2(\d{2}|\d{4})!$3$2$1$2$4!"
```

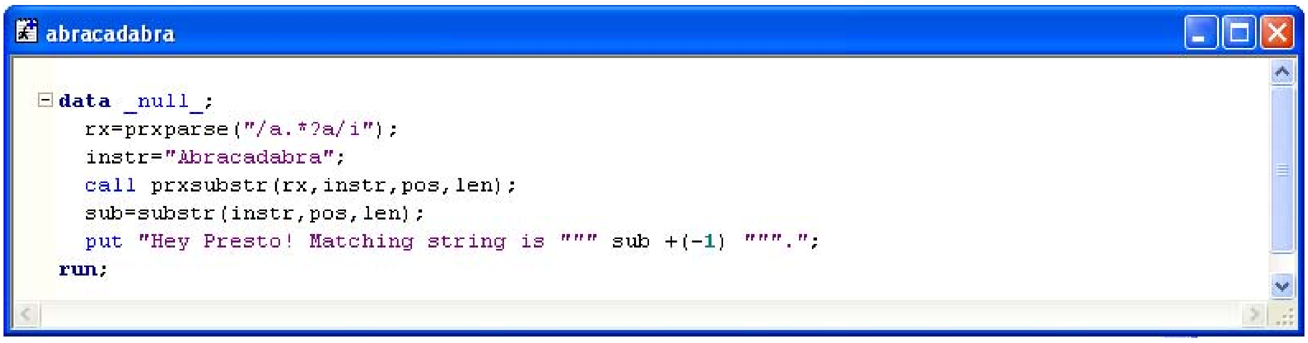
Here, the parts relating to separators have been highlighted in red. Both "\2" and "$2" appear, in the appropriate places.

UNDOCUMENTED FEATURES

The PRX functions in SAS use Perl regular expressions. The full Perl regular expression syntax is available on the web – see References below. The parts that SAS have not implemented are listed in the SAS Institute documentation, in a section referring to the "artistic license". Everything else ought, by implication, to be available, even though not all of it has been specifically documented by SAS Institute. All the features described below are currently working for me on Windows in SAS 9.1.3 and SAS 9.2, but they cannot be guaranteed to work on all platforms or for all versions of SAS.

NON-GREEDY REPETITION

Perhaps the most useful of these features is *non-greedy repetition*. As noted before, SAS will normally find the *longest* substring of the source that matches the regular expression i.e. the matching is *greedy*. The "?" character can however be used to make repeat specifiers non-greedy. For example:



```
data _null_;
  rx=prxparse("/a.*?a/i");
  instr="Abracadabra";
  call prxsubstr(rx,instr,pos,len);
  sub=substr(instr,pos,len);
  put "Hey Presto! Matching string is """ sub +(-1) """.";
run;
```

```
Hey Presto! Matching string is "Abra".
```

Rather than "*", the expression here uses a repeat specifier of "*?". This means "as few times as possible". With "*" alone, the matching string would have been the longest one that began and ended in "a": "Abracadabra". With "*?", it is the shortest. All repeat specifiers can be modified in this way e.g. "{2,}?" for "at least twice, but as few times as possible", and even "??" for "at most once – and preferably not at all".

CASE INSENSITIVITY

Did you spot the deliberate mistake in that last paragraph? In fact, the matching string found would normally have been "acadabra" rather than "Abracadabra", because normally matching is case sensitive. But the program above uses another undocumented feature: that "i" at the end, after the final delimiter, makes the whole expression *case insensitive*.

The two other undocumented features we are going to look at are both useful, ironically, for documenting your PRX-related code.

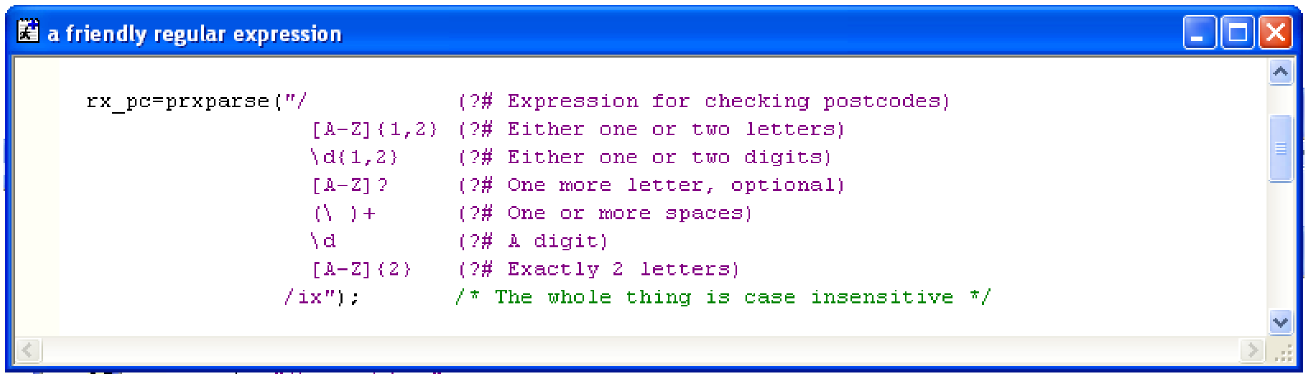
WHITE SPACE USING EXTENDED REGULAR EXPRESSIONS

An "x" right at the end of an expression, after the final delimiter – where the "i" was in the last example - will convert it to an *extended regular expression*. This is less impressive than it sounds; all it means is that you can insert white space within it without altering the meaning. This can greatly improve readability, though. If you actually want a space as a meaningful part of the expression, you will need to escape it with "\" – in fact it might be clearer if you made a subexpression of it: "(\ )".

COMMENTS USING EXTENDED PATTERNS

You can insert *comments* into your expressions, in the form of specially-formatted subexpressions thus: "(?# This is a comment.)". In the Perl terminology, this is one form of *extended pattern*; there are others.

These two features enable regular expressions to be laid out in ways that make them much easier to understand, for example:



```
rx_pc=prxparse("/           (?# Expression for checking postcodes)
                 [A-Z]{1,2} (?# Either one or two letters)
                 \d{1,2}    (?# Either one or two digits)
                 [A-Z]?     (?# One more letter, optional)
                 (\ )+      (?# One or more spaces)
                 \d         (?# A digit)
                 [A-Z]{2}   (?# Exactly 2 letters)
               /ix");       /* The whole thing is case insensitive */
```

SAS is liable to warn that your quoted string is getting unusually long and that you may have unbalanced quotes, but the expression will still work. Note though that if you are defining expressions for use with PRXCHANGE, you cannot use either of these readability features on the "new string" side of the expression.

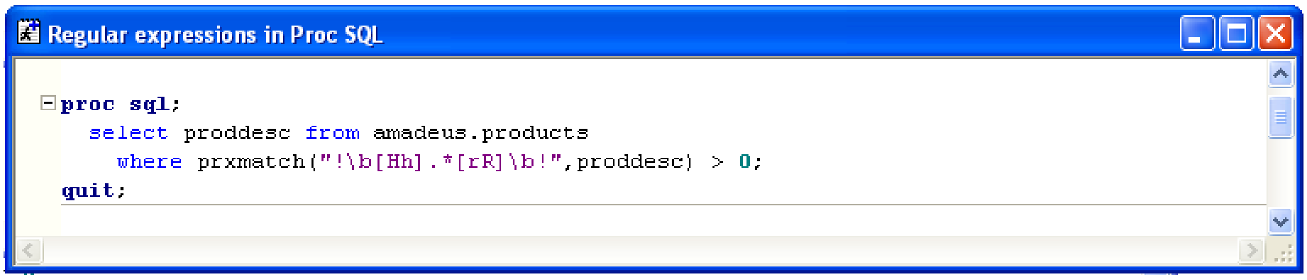
OTHER UNDOCUMENTED FEATURES

Careful consideration of the full Perl regular expression syntax and the parts excluded by SAS may reveal other features that ought to be available despite not being specifically documented by SAS Institute. Among these are *local* case insensitivity, lookahead, and negative lookahead, all of which do indeed appear to work. However, some other features that might be expected to work - such as lookbehind and negative lookbehind – appear not to.

OTHER WAYS OF USING REGULAR EXPRESSIONS

IN PROCEDURES SUCH AS PROC SQL

All the examples given above have shown the PRX functions being used within a data step. The PRX functions also enable regular expressions to be used in procedures, for example in Proc SQL:



```
proc sql;
  select proddesc from amadeus.products
    where prxmatch("!\b[Hh].*[rR]\b!",proddesc) > 0;
quit;
```

This query will find products with descriptions like "Patio heater", "Hover mower" and "Hand cultivator". Here the regular expression is being specified to PRXMATCH as a character string. This contrasts with the data step examples above, in which PRXPARSE is used to parse the regular expression, and its parsed form is used as the PRXMATCH parameter. Within a data step, that is the better approach, as it means the regular expression is parsed once only, rather than once per observation.

WHEN DEFINING YOUR OWN FUNCTIONS

The PRX functions can be used within user-defined functions created using Proc FCMP. This could provide a good way of preventing regular expressions from being seen by those who might be alarmed by them.

IN THE ENHANCED EDITOR

The "find" and "replace" functions of the SAS Enhanced Editor also support regular expressions, although not to the same extent as the PRX functions. See References below.

CONCLUSION

Perl regular expressions greatly enhance the power of the SAS language. It is worth investing a little time in getting to know them.

REFERENCES

The full Perl regular expression syntax is documented at http://perldoc.perl.org/perlre.html. Parts of it not implemented in SAS are listed in SAS Institute documentation, as noted above.

Ken Borowiak and Russ Lavery, 2008 "An Animated Guide: SAS Editor Regular Expressions"

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Author Name:	Bob Newman
Company:	Amadeus Software Limited
Address:	Mulberry House, 9 Church Green, Witney, Oxon OX28 4AZ
Work Phone:	+44 (0) 1993 848010
Email:	bob.newman@amadeus.co.uk
Web:	www.amadeus.co.uk