Using Regular Expressions with SAS®

Brian Conley, Prevision Marketing, MA

ABSTRACT

It is well known that SAS is extremely powerful at processing numeric data, however, anyone who has been faced with processing large amounts of textual data understands that this can be quite a challenge. Using RXPARSE, RXMATCH, Call RXCHANGE, and Call RXSUBSTR will enhance a programmer's ability to manipulate strings and match patterns.

This paper will investigate SAS' regular expression (RX) functions and call routines. By presenting several examples, it can be shown how these functions and call routines can be used together to process strings using character classes, pattern abbreviations, special symbols and more.

INTRODUCTION

SAS' regular expression functions are only slightly documented, and sometimes awkward to use. As such, this paper isn't to advocate their use, but only to serve as a brief introduction to their existence. However, there are some circumstances where one of these functions may be able to succinctly and clearly work in ways that the usual SAS string functions would not.

The foundation function that is used in conjunction with the other RX functions and call routines is Rxparse. This function is quite versatile and will be the first this to become familiar with.

RXPARSE

The first function to become familiar with is *RXPARSE*. By using the RXPARSE function, one instantiates the parsing routine to look for a particular pattern expression. Some examples of valid pattern expressions would be a string, a digit, a character class, or a user-defined character class.

Some of the more useful default character classes are contained in the following table:

Use this:	To match this:
$a or $A	a-z A-Z
$c or $C	0-9 a-z A-Z
$d or $D	0-9
$I or $I	a-z A-Z (*only if first character in string*)
$l or $L	a-z
$u or $U	A-Z
$w or $W	whitespace

The RXPARSE function is used in conjunction with the other parsing functions and call routines. The way RXPARSE communicates with these other functions is by passing a value, customarily called RX. Normally, the RXPARSE function is used as shown:

```
Rx = rxparse ("$character class");
```

Once the programmer has chosen a pattern to match, the rxparse function can be executed, and then other parsing functions can be used. One thing to keep in mind when using rxparse, however, is that huge performance degradation will occur if this line is executed multiple times. In other words, one needs to be careful to add logic such as the following:

```
If _n_ = 1 then do;
  Rxparse = ("$character class");
End;
```

USER-DEFINED CHARACTER CLASSES

The syntax for defining user defined character classes is very simple. Simply enclose the pattern that is desired for matching in single quotes. SAS will then find a match to the first character within the quotes. So, if one wants to match on only capital letters, the following code could be used:

```
Rx = rxparse("$ 'A-Z');
```

Conversely, SAS can search for expressions that do not match a string. These are called character class complements, and would be used as follows. In this example, the user wants to match on consonants:

```
Rx = rxparse("^ 'AEIOU');
```

MATCHING ENTIRE WORDS

So far, we have seen how Rxparse can match characters or numbers, but matching entire is also possible. Refer to the following table to see the pattern abbreviations:

Use This:	To Match This:
$f or $F	Floating Point Number
$n or $N	SAS Name
$p or $P	Prefix (User Specified)
$q or $Q	User specified String
$s or $S	Suffix (User Specified)

For example, suppose your data contained address information, and you wanted to find last names that contained the word 'Smith' as the first part of their name, but you didn't want to find where the last name was exactly 'Smith', or where the name ended with Smith as in 'Blacksmith'. This could be coded as follows:

```
Rx = rxparse("$P 'Smith');
```

CHANGE EXPRESSIONS AND CALL RXCHANGE

The rxparse function also has the capability to change strings. To do this, one needs to add the *TO* keyword to the function. Consider an example where address data come in with a string, stored in a variable called addr_1, that is abbreviated, such as 'St.' and the programmer wishes to change it to 'Street'. This can be done as follows:

```
Rx = rxparse("St. to Street");
```

This tells SAS that you want to change St. to Street. In order for SAS to make this change, the call routine Rxchange needs to be called. Call Rxchange is one of the ancillary functions and call routines that go hand in hand with rxparse.

Continuing the example:

Call rxchange(rx, 1000, addr_1);

This tells SAS to look at the variable addr_1, and if there is the string 'St.' change it *in place* to 'Street'. The second argument in the call routine is just a number that specifies the maximum number of times the substring should be changed.

Note that using call rxchange as seen in the previous example will change the string in place. If the programmer wants to preserve the original data and write the changed data out to a new variable, an optional fourth argument can be specified:

Call rxchange(rx, 1000, addr_1, newAddr_1);

This will preserve the original addr_1 variable, and write out any changes to newAddr_1. If there are no changes, then addr_1 is copied unchanged to the new variable.

CALL RXSUBSTR

Call RxSubstr can return some simple statistics about the parsed expression, such as position, length, and score. In this example, the score argument has been left off, as this is optional. The score argument is the fifth argument in the call rxsubstr routine.

Consider an example where there is a variable addr_1 which contain the string '123 West St.':

```
rx2 = rxparse(" St. to Street");
call rxsubstr(rx2,addr_1,position,length);
```

Will result in position = 10 and length = 3.

The Rxsubstr function also gives you the ability to use the score argument. The user can assign scores to certain strings, which tells SAS to search the string and find all matches, but overrides the default behavior of returning the position and length of the longest substring.

For example, say you wanted to build a simple state machine to process some address data, and there were four states based on whether the address contained North, South, East, or West. While this could be done with the traditional if-then-else code, it could also be done using the scoring feature of rxsubstr.

First, like in the previous examples, set up the rxparse expression:

```
staterx = "`[$# 'North' #9] | [$# 'South' #4] |
           [$# 'East' #2] | [$# 'West' #1] ";

rx = rxparse(staterx);
```

In this case I have chosen to assign the rxparse argument to a variable, because experience shows that this makes things a little less confusing. Also, note that the first character after the first double quote is a slanted single quote (`), from the top left corner of the keyboard, and not the vertical single quote from the right side of the keyboard. Beware of confusing the two, as the function will not work otherwise.

The last bit to program is to add the rxsubstr call routine:

call rxsubstr(rx, addr_1, start, len, state);

Now, the variable STATE is available to the datastep for further processing. If the variable addr_1 contained North, then STATE would have the value of 9. If addr_1 contained South, STATE would have the value of 4, and so on.

RXMATCH

Rxmatch will return the position of the matched string. It is the same as the position variable in the RxSubstr call routine, but without Score

CONCLUSION

This paper has briefly outlined some of the more approachable aspects of SAS' regular expression functions and call routines, by presenting some simplified examples. There are many more abilities of the RX facility, but many are somewhat cryptic.

One hopes that people will use these functions and call routines in an experimental fashion, and perhaps the SAS Institute will further develop and enhance both the code and the documentation, because regular expressions can be a powerful tool.

REFERENCES

SAS Institute Inc.
SAS OnlineDoc®, Version 8
February 2000

CONTACT INFORMATION

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

Brian Conley
Brian_Conley_J@yahoo.com