Some Practical Ways to Use the New SAS Pattern-Matching Functions

Mike Rhoads, Westat, Rockville, MD

ABSTRACT

The 6.11 and 6.09E releases of SAS® software contain, in experimental form, a powerful new set of pattern-matching functions, which can be used in DATA steps, macros, or SCL code. This paper provides a summary description of these new routines and includes several examples of how their use can significantly reduce the coding effort required to perform common tasks.

INTRODUCTION

SAS software has always contained a fairly extensive set of functions for finding particular characters within strings (INDEX, INDEXC, VERIFY) and for changing the contents of character strings in certain ways (TRANSLATE, COMPRESS, UPCASE, LEFT, RIGHT, SUBSTR on the left side of an assignment statement). New functions have been added periodically -- for instance, version 6.07 added COMPBL, TRANWRD, INDEXW, QUOTE, and DEQUOTE. Nevertheless, while these functions are extremely useful, they lack the kind of generality and expressive power that is found in some other software, such as the **grep** command in Unix.

The 6.11 and 6.09E releases of base SAS software add two functions and three CALL routines that provide pattern matching and changing capabilities using regular expression patterns. The pattern notation allows for built-in and user-defined character classes, wild card characters, elements that are optional or occur a varying number of times, and either/or subpatterns.

These functions are currently in experimental status and are expected to become production in a later release of the SAS system. They can be called from DATA steps, SCL modules, or macro routines (using the %SYSFUNC interface). Draft documentation can be downloaded from SAS Institute's Internet site (see *References* for the full Internet address).

The next section of this paper summarizes how this facility operates and briefly describes the five routines that it contains. This is followed by a summary of the major constructs that can be used in pattern specification. The remainder of the paper shows how these routines can be used both to search for patterns within character strings and to modify the original string based on the results of the search.

BASIC CAPABILITIES AND SYNTAX

Using the pattern-matching routines in a DATA step involves three distinct phases. (The basic principles are similar if the functions are being used in macro or SCL code instead.) These phases are as follows:

1. Use the RXPARSE function to parse (compile) the pattern. This is normally done once, at the beginning of the step. The facility is designed in this manner to improve performance when processing large amounts of data, since the time-consuming step of translating the pattern into a more efficient internal representation needs to be performed only once, rather than every time the pattern is used.

2. Use a pointer to the parsed version of the pattern in a function or call routine for string searching or modification. This step is normally performed multiple times (e.g. for every record in a data set). The available routines are **RXMATCH** (basic pattern matching), **RXSUBSTR** (advanced pattern matching), and **RXCHANGE** (string modification).

3. At the end of the step, use **RXFREE** to free up the memory occupied by the parsed version of the pattern.

The syntax of these routines is as follows:

Initial pattern compilation

rx = **RXPARSE** (*pattern*);

This function parses a pattern (which may be a character constant, variable, or expression) for later use. The returned value *rx*, while nominally a numeric variable, actually serves as a pointer to the parsed version of the pattern and is used as an argument to all of the other RX routines. The documentation warns that this value should not be printed, saved to an output data set, or otherwise operated on, since it may not actually represent a valid number. Note also that the first character of the pattern must be a backquote (`), otherwise it will be interpreted as a literal rather than as a pattern.

Basic pattern matching

position = **RXMATCH** (*rx,string*);

This is the most basic of the RX routines. It searches *string* for the previously-parsed pattern pointed to by *rx*, returning the position within *string* where the match begins (or 0 if no match is found).

Enhanced pattern matching

CALL RXSUBSTR (*rx, string, position, length, score*);

RXSUBSTR performs the same matching as does RXMATCH, and the first three arguments have the same meanings as for RXMATCH. The final two optional arguments allow additional information about the match to be returned. *Length* indicates the number of

characters that matched, while *score* returns a numeric score value that can be set using expressions within the pattern.

String modification

CALL RXCHANGE (*rx, times, string, result*);

RXCHANGE uses the parsed expression pointed to by *rx* to search *string* for a pattern and change the matched substring as specified. *Times* specifies the maximum number of changes that will be made. The modified character string is placed into *result*, or back into *string* if the optional final argument is not specified.

Cleanup

CALL RXFREE (*rx*);

RXFREE should be called when you are finished with a pattern expression to free up the memory that is occupied by the parsed version of the pattern.

SPECIFYING THE PATTERN EXPRESSION

The power of the RX routines really lies in the richness of the constructs that can be used in the pattern expression. This section summarizes many of the basic constructs for pattern matching. A few additional constructs that are used only when changing strings are discussed in a later section of the paper.

Quoted literals. Patterns may contain character strings in single or double quotes, which indicate that the string being searched must contain exactly that sequence of characters.

Unquoted literals. Patterns may also contain letters, digits, periods, and underscores as unquoted literals, which indicate that the string must contain that character. Unquoted letters match either upper case or lower case.

Wild card characters. The question mark (?) matches any single character, while the colon (:) matches any sequence of zero or more characters.

Character classes. Character classes specify that the character in the search string must match one of the characters in the class. There are a number of predefined classes, such as $d for any digit and $u for any uppercase letter. You can also specify your own classes by preceding a quoted string with a dollar sign. For instance, $'13579' matches any odd digit. The dollar sign can be replaced by a ~ or ^ to specify the complement of a set of characters: e.g., ^'13579' matches any character except an odd digit.

Miscellaneous abbreviations. There are also a number of abbreviations that are not character classes as such. For instance, $f matches any floating point number, $n matches any valid SAS name, and $q matches any quoted string.

Modifiers. There are a number of ways in which the effect of individual pattern elements can be modified. Enclosing the pattern in brackets indicates that it is optional, following it with an asterisk indicates that it may occur zero or more times, while following it with a plus sign means it may occur one or more times. A single vertical bar between patterns means that one or the other must match. Parentheses may be used to group pattern elements.

Location specifiers. A notation using the at-sign (@) can be used to specify that a match or portion thereof begin or end in a specific position of the string. @1 specifies the first column, and in general @n specifies the *n*th column from the beginning if n is positive and the *n*th character from the end of the string if n is negative. @0 denotes the end of the string.

USING PATTERNS FOR DATA VALIDATION

One obvious area for exploiting the versatility of the RX routines is in data validation rules that go beyond what can be conveniently done with traditional SAS functions and formats. As a simple example, take the following pattern, which can be used to validate an 11-character field that should contain a Social Security number:

```
`  $d$d$d '-' $d$d '-' $d$d$d$d
```

This is a relatively simple but useful pattern. It specifies that, in order to match, the string must contain 3 digits, followed by a hyphen, followed by 2 digits, another hyphen, and then 4 digits. Note that the various elements of the pattern are logically concatenated, that white space may be used to improve readability, and that the first character of the pattern must be a backquote.

Shown below is a short DATA step that illustrates how this pattern could be parsed and used. The first section of the program is devoted to setup and housekeeping: the pattern is parsed on the first iteration of the DATA step and a pointer to the parsed version is saved in the retained variable RX. RXFREE is then used to free up memory when data set processing has been completed.

RXMATCH is then used to verify the data, with the starting position of the match being placed in the variable MATCHPOS. The VALID message is displayed if RXMATCH's search of the string found an occurrence of the pattern, thus returning the starting position of the match. If the pattern was not found, MATCHPOS is set to 0 and the message INVALID is printed.

```
data _null_;
  /* HOUSEKEEPING */
  if _n_ = 1 then rx = rxparse
    ("`  $d$d$d '-' $d$d '-' $d$d$d$d  ");
  retain rx;
  if eof then call rxfree (rx);

  set mystuff end=eof;

  /* NOW USE THE PATTERN */
  matchpos = rxmatch (rx, ssndata);
  if matchpos > 0
```

```
    then put ssndata=  '  VALID';
  else   put ssndata=  '  INVALID';
run;
```

The following more complex example demonstrates additional elements that can be used in an expression. It shows a pattern that could be used to check U.S. monetary data, such as $14,598 or $2.95. (The numbered line below the pattern refers to the notes.) Although the results produced by the example are similar to the operation of the COMMAw.d informat, the techniques illustrated could be adapted to validate complex string patterns for which SAS does not provide its own informats.

```
` @1 '$' $d[$d][$d] (','$d$d$d)* [.$d$d] ' '* @0
   1  2      3           4         5     6    7
```

The @1 (1) "anchors" the match at the beginning of the string. The quoted dollar sign (2) must be matched exactly in the string, and this must be immediately followed by at least one and up to three digits (3). The pattern consisting of a comma followed by three digits (4) is enclosed in parentheses and followed by an asterisk, indicating that it may occur any number of times or not at all. Optionally, this may be followed by a decimal point and two more digits representing cents (5). Then, there may be some blanks (6) between the matched dollar amount and the end of the string (7).

ENHANCED PATTERN MATCHING

The RXSUBSTR routine adds more versatility by allowing the user to get back the length and "score" of the match, in addition to its starting position. The pattern expression syntax contains notation to modify the score of a match as it proceeds -- for instance, #5 means "add 5 to the current score". Let's take the following pattern and CALL statement as a quick illustration:

```
`  cal #1  [' ' ripken #2]  [' ' jr #4]
call rxsubstr (rx,string,start,len,score);
```

In this example, the string "Cal" would return 3 for LEN and 1 for SCORE, while "Cal Ripken" would return 10 for LEN and 3 for SCORE (1 + 2), and "Cal Ripken Jr" would produce a value of 13 for LEN and 7 for SCORE (1 + 2 + 4). Note that the unquoted letters in the pattern match either uppercase or lowercase characters.

When used in conjunction with the "or" operator (single vertical bar), the scoring facilities of RXSUBSTR can also be used for recoding, as shown by the following pattern:

```
` (@1 gold ' '* @0) #1  |
(@1 silver ' '* @0) #2  |  (@1 bronze ' '*
@0) #3
```

This pattern returns a score of 1 if the input string is equal to "gold" (any case), 2 for "silver", and 3 for "bronze". The vertical bars indicate that the match must consist of one of the three alternatives, each of which are contained within parentheses. Within each of these, the @1 forces the match to start at the beginning of the string (column 1), with no leading blanks or other characters. @0 refers to the end of the string being searched -- along with the expression ' '* (0 or more blanks), it allows trailing blanks but no other characters between the word of interest and the end of the string.

ALTERING STRINGS USING RXCHANGE

The RXCHANGE routine allows the value of the search string to be changed, based on how the match proceeds. To accomplish this, the pattern that is passed to RXPARSE must consist of a matching expression, followed by the keyword TO and then a "change expression".

We have already seen the syntax of the matching expression: the primary enhancement that is relevant here is that enclosing part of the expression in angle brackets (< >) "tags" that part for reference in the change expression.

Like the matching expression, the change expression may contain quoted and unquoted literals. In addition, values from the matched expression that have been tagged can be carried over into the change expression: =n copies the value of the *n*th tagged substring, =-n copies the value of the *n*th tagged substring from the end, and == copies over the entire matched substring.

One of the most useful functions of RXCHANGE is to "normalize" user-specified parameters so that they are then easier to break apart and work with. For instance, let's take an example where a user can input an arbitrarily complex SAS variable list, including single dashes, double dashes, etc. (For instance, A B C1-C3 D--G H I.) Using SCAN (with a blank as the only delimiter) to break such a compound list into more basic elements for further processing seems like a good way to proceed, but the problem becomes more difficult if the user likes to put blanks before or after his hyphens. Rather than complicating the coding or restricting the user's flexibility, RXCHANGE provides an easy way to preprocess the input string and magically eliminate all the white space surrounding hyphens, using the following pattern and statement:

```
`  ' '*  '-'  ' '*    TO    '-'

CALL RXCHANGE (rx, 99999, string, result);
```

This pattern works by specifying a hyphen, optionally preceded and/or followed by some blanks, as the pattern to be matched, and a single hyphen with no leading or trailing blanks as the replacement pattern. The call to RXCHANGE contains the pointer to the parsed pattern, the maximum number of changes to be made (here an arbitrarily high 99999), the original string, and the altered string (RESULT).

RXCHANGE can also be used to transpose the order of substrings within a string. The following expression, for instance, takes dates in MM/DD/YY format and changes their representation to YY/MM/DD:

```
` <$d[$d]>'/'<$d[$d]>'/'<$d$d>  TO  =3'/'=1'/'=2
```

This pattern works by setting up a search pattern for the complete date and using angle brackets (<>) to tag each of the three elements -- month, day, and year. The change expression after the TO then simply rearranges the elements, starting with the tagged element that was originally third (YY), and then following this with the first tagged element (MM) and finally the portion (DD) that was originally in the second position.

CONCLUSION

These new pattern-matching functions greatly expand the power and flexibility of base SAS software for manipulating character strings. Data validation, recoding, normalization, and string element rearrangement are just a few of the ways in which these new features can be used. As more programmers begin to learn about and use these routines, and as they gain official support and documentation from SAS Institute, it can be expected that additional innovative uses for these functions will be developed.

REFERENCES

SAS Institute Inc., *SAS Language: Reference, Version 6, First Edition*, Cary, NC: SAS Institute Inc., 1990. 1042 pp.

SAS Institute Inc., *Macro Facility Enhancements for Release 6.09E and Release 6.11*, unpublished 1996 draft. 103 pp. Available on the SAS Institute Web site at ftp://www.sas.com/techsup/download/base/macroenh.psl.

ACKNOWLEDGMENTS

The author wishes to thank Ian Whitlock and Roberta Garrison-Mogren of Westat and Jeff Polzin of SAS Institute for their assistance before and during the writing of this paper.

AUTHOR CONTACT INFORMATION

Michael D. Rhoads
Westat
1650 Research Blvd.
Rockville, MD 20850
rhoadsm1@westat.com