Paper 37-27

Combining Pattern-Matching And File I/O Functions: A SAS® Macro To Generate A Unique Variable Name List

Michael P.D. Bramley, Venturi Technology Partners, Waukegan, IL

ABSTRACT

When a large number of variables in numerous datasets must be processed in different ways depending upon their names or labels, implementing a macro that returns a variable list based on patterns makes sense. It turns out that mimicking the basic SAS® variable list operators is not too hard and provides an opportunity to add those features that the author wishes SAS Institute had implemented.

In addition to pattern-matching in the SAS® System, this paper illustrates the SAS® programming techniques of how to use the %SYSFUNC macro function and how to interrogate the contents of a dataset. This paper does not assume any advanced SAS® MACRO knowledge on the part of the reader.

Finally, this paper combines the above techniques into a macro function that utilizes the SAS® pattern-matching capabilities to return a unique variable list based on user-specified criteria.

THE PROBLEM

The basis for this paper is the need to identify variable names in datasets that are to be processed in a similar fashion, based on user-specified pattern-matching criteria. The criteria included the ability to choose variables by:

1. Matching text against their prefix and/or suffix.

2. The presence or absence of text in the name or label.

3. Specifying the variable type (Num/Char/Both).

SAS® currently implements two built-in operators to meet this general need:

1. The colon operator, which matches all variable names that have the same prefix.

2. The minus operator, which matches all variable names that have the same prefix and a numeric suffix within the specified range.

Note that the above list does not include the double-minus operator. This operator matches variables that exist between two variables, and as such, performs no pattern-matching.

While the above built-in operators cover many situations, they do have some limitations, namely:

1. The operators only apply to the DATASTEP or procedure code. That is, these operators cannot be used directly in a macro, say, as part of a macro parameter.

2. The operators do not include a complement operator, that is, a way of returning everything not matching the pattern.

3. The operators only match prefixes--not suffixes. For example, SAS® cannot match all variables that end with the characters 'KM'.

4. When using multiple operators, for example, VAR1-VAR99 VAR9:, the expression may result in the same variable name being listed more than once. While this has no adverse effect on the PDV, it can produce redundancy in procedure output, warning messages in the log, and/or errors in SAS® code.

THE SOLUTION

To overcome these limitations, the pattern-matching capabilities in SAS® were successfully employed to create a macro that extends the basic SAS® variable name operators. The final solution had to meet the following criteria in order to be beneficial:

1. Easy-to-Learn: the macro must use syntax that SAS® programmers are already familiar with.

2. Easy-to-Use: the macro must not create RUN-boundaries, to make it usable anywhere in the code stream.

3. Easy-to-Extend: the macro must be well designed and written to allow for simple modifications/extensions.

If the macro (or any code) meets the above criteria, then it should intrinsically add value to the shop and provide a professional polish--the ability to complete difficult tasks in time and in an elegant way.

Before going into the details of the final macro, it is vital to understand the two main components that make this macro work: pattern-matching and interrogating the contents of a SAS® dataset.

AN OVERVIEW OF PATTERN-MATCHING

Pattern-matching in SAS® is a method of efficiently and quickly searching (and modifying) character strings for simple or multiple text patterns. Pattern-matching in SAS® is accomplished by a core set of functions:

1. rxid=RXPARSE(pattern): parses and stores the pattern in an internal format for future use. Pattern is a non-null string and may include character classes that have special meaning and make life much easier. This function returns a numeric value (rxid) that must be retained and used for all future RX-function calls related to this pattern.

2. RXFREE(rxid): frees the memory used to hold the parsed pattern setup by RXPARSE. This function should be paired with each instance of RXPARSE, similar to a DO/END.

3. pos=RXMATCH(rxid,string): searches the string for patterns. This function returns the position (pos) in string that matches a pattern defined by rxid (defined by a previous call to RXPARSE).

4. RXSUBSTR(rxid,string <,length <,score>>): same as the RXMATCH function, except that it optionally returns the length and/or score of the pattern matched.

5. RXCHANGE(rxid,rep,string <,new-string>): changes the contents of string using patterns in rxid up to rep times and

optionally stores results in new-string. The value of rep can range from 0 to 2^{31}-1.

A dollar sign and a letter or number identifies the character classes used by RXPARSE. Some pre-defined character classes are $p (prefix), $s (suffix), $-, and $a. The $p/$s classes search for text at the start/end of tokens within a string, while $- forces a search from right to left. The $a class searches a string for letters (ignoring case). For example, the pattern ‘$p IN’ matches the text ‘RIN INS TIN’ at position 5. Patterns may be or-ed together with an exclamation mark, and the complement obtained by preceding a pattern with a carat.

The real workhorse in the above list is RXCHANGE. This routine requires that you first specify an expression of "old-pattern TO new-pattern <, ...>" in a call to RXPARSE (note the keyword TO). RXCHANGE will then replace occurrances of old-patern with new-pattern. Thus, you can replace many IF/INDEX/SUBSTR blocks with three lines for each variable that you have to modify: RXPARSE, RXCHANGE, and RXFREE. This could make the DATASTEP simpler and easier to maintain.

A PATTERN-MATCHING EXAMPLE

Suppose you have a character variable, YSTRING, in the dataset TEST. You are interested in keeping only those observations where the contents of YSTRING contain 'abc' that is not followed by 'xyz'. The following DATASTEP performs this task efficiently:

```
Data POSABS(Where=(POSITION > 0)) ;
   Set TEST End = EOF ;
   Keep YSTRING POSITION ;
   Retain RX ;

   If _N_ = 1 Then
      RX = RXPARSE("'abc' ^'xyz'") ;

   POSITION = RXMATCH(RX,YSTRING) ;

   If EOF Then
      Call RXFREE(RX) ;
Run ;
```

In the first iteration of the DATASTEP, the program sets up the pattern ‘abc’ ^’xyz’ and assigns the result to RX. Each observation’s YSTRING is then searched for this pattern with the result assigned to the numeric variable POSITION. After the last observation has been processed, the memory holding the parsed pattern is released. The dataset option on the output dataset ensures that only those observations that match the pattern are saved to the new dataset.

PITFALLS, WARNINGS AND HINTS

SAS Institute implements true pattern-matching, not the UNIX variety (cf.egrep), with its affinity for escape characters that can confuse even the best programmer. However, this does not mean that all pattern-matching is a breeze: the user must be aware of the following traps when designing patterns. (This is not an exhaustive list).

Specifying a pattern of ab xy does not mean search for the characters a, b, space, x, y. As the pattern is unquoted, SAS® strips the pattern’s spaces and searches for any occurrence of abxy, without regard to case. To make spaces significant, enclose them in single quotes, as in, ab ' ' xy. Thus, the SAS® code would be:

```
RX = RXPARSE(" ab ' ' xy " ) ;
```

The double-quotes are needed for the whole pattern, and the quoted space is now significant. Spaces can (and should) be liberally inserted into patterns to make them more legible. Don’t forget to add meaningful comments to the code!

To make the case 'sticky', embed the pattern in single quotes. So, to look for capital A, capital B, lowercase x, followed by capital Y, specify a pattern of 'ABxY'. The SAS® code for this is:

```
RX = RXPARSE(" 'ABxY' " ) ;
```

Patterns can be combined to create complex expressions, as in, 'AxB' : 'XbF'. This is a natural extension of the colon operator and will cause SAS® to search for 'AxB' followed by 'XbF' with zero or more characters in between. In SAS® pattern-matching, both the prefix and the suffix are optional for the colon operator. Thus, a pattern of ':' matches every string, including null strings.

When you debug or validate your code, you may not always see the results you expect. If a pattern matches more than one substring, then it is important to remember that the RXMATCH/RXSUBSTR routines will return information on the longest substring matched.

If you are using SAS Version 6, you have to be wary of a bug. The RXPARSE function always returns a value of one. This means that it is impossible to keep track of more than one pattern at a time. Also, all patterns must be preceded by a backquote (under the tilde) in this version.

The RXCHANGE function will pad your string argument with spaces if its length is less than its defined length. If your string argument is also your new-string argument, the contents of your variable may be corrupted. A simple DATASTEP demonstrates these limitations:

```
Data _Null_ ;
  Length STRING NEWSTRING $20 ;
  STRING = "Too many blanks" ;

  RX = RXPARSE( " ' ' TO '*' " ) ;
  Call RXCHANGE(RX, 999, STRING, NEWSTRING) ;
  Put NEWSTRING = ;
  Call RXCHANGE(RX, 999, TRIM(STRING),
                NEWSTRING ) ;
  Put NEWSTRING = ;

  Call RXCHANGE(RX, 999, STRING, STRING ) ;
  Put STRING= ;

  Call RXFREE(RX) ;
Run ;
```

The following output is generated by the above DATASTEP. The first line reveals that RXCHANGE has padded the STRING variable to its defined length, with the result that those spaces are converted to astericks. By trimming the STRING variable, the expected results are obtained, as shown in the second line. The last line demonstrates that the third and fourth arguments to RXCHANGE should not be the same variable.

```
NEWSTRING=Too*many*blanks*****
NEWSTRING=Too*many*blanks
STRING=********************
```

There are four important things to remember when writing code that performs pattern-matching. First, create a small test dataset that you can quickly use to validate your code. Second, begin by creating a simple pattern and once you know it works, add to it in steps. Third, no detail is too small to be ignored--always double-check your pattern. If you start with a complex pattern, you could spend hours trying to determine why it doesn't work as expected, only to find that something wasn't quoted. Last, remember that the time required for matching is roughly proportional to the length of the pattern multiplied by the length of the string being searched. Thus, always try to make patterns as short as possible, especially when you have a lot of data to process.

INTERROGATING A DATASET

At the beginning of every SAS® dataset is a structure that SAS® uses to determine the contents of the dataset. The SAS® programmer can also access this content information by utilizing the following File I/O functions:

1. fid=OPEN(dataset): opens dataset for access. Returns a numeric file identifier (fid) to be retained and used for future calls to File I/O functions. This function returns zero in the event of an error.

2. RC=CLOSE(fid): closes the dataset referenced by fid. Returns a non-zero result if an error occurred. This function must be paired with each instance of OPEN, similar to a DO/END.

3. NVar=ATTRN(fid,attrib): returns the numeric value in NVar for the specified attribute from the dataset referenced by fid. Examples of attributes are the number of variables (NVARS) or the number of physical observations (NOBS).

4. CVar=ATTRC(fid,attrib): returns the character value in CVar for the specified attribute from the dataset referenced by fid. Examples of attributes are the name of the dataset engine (ENGINE) or the dataset's label (LABEL).

5. Var=VARNAME(fid,pos): returns the name of the variable in Var at position pos in the dataset referenced by fid. If pos is out of range, the function returns null and prints a warning in the log.

6. Var=VARNUM(fid,name): returns the position of the variable in Var with the specified name in the dataset referenced by fid. If the variable does not exist in the dataset, the function returns zero.

7. Var=VARTYPE(fid,pos): returns the variable type (C or N) of the variable at position pos in the dataset referenced by fid. This function returns null and prints a warning in the log if pos is out of range.

The order of operations for retrieving content information from a dataset using the File I/O functions is:

a) open the dataset,
b) obtain the required information,
c) close the dataset.

The last step is extremely vital: if you open a dataset, but do not close it, SAS® will not allow you to write, update or delete that dataset in the same session or batch job until the dataset is closed.

To obtain the content information from a dataset without creating RUN-boundaries, a macro must use the %SYSFUNC macro function. This macro function acts as a wrapper function for the DATASTEP function it calls, allowing the programmer to call most of the DATASTEP functions outside of the DATASTEP. This one function significantly extends the macro programmer's ability to write macros that carry out DATASTEP tasks without creating RUN-boundaries.

BUILDING A VARIABLE LIST

The following macro function demonstrates the use of the %SYSFUNC macro function and how to interrogate a dataset.

```
%Macro GetVars(Dset) ;
  %Local VarList ;

  /* open dataset */
  %Let FID = %SysFunc(Open(&Dset)) ;

  /* If accessable, process the
     contents of dataset */

  %If &FID %Then %Do ;
    %Do I=1 %To %SysFunc(ATTRN(&FID,NVARS)) ;
      %Let VarList = &VarList
                  %SysFunc(VarName(&FID,&I));
    %End ;

    /* close dataset when complete */
    %Let FID = %SysFunc(Close(&FID)) ;
  %End ;

  &VarList
%Mend ;
```

If the above macro can access the dataset that is passed as a parameter, it enters the loop with the macro variable I initialized to 1 and incremented by 1 at the end of the loop until the value of I is greater than the number of variables in the dataset (returned by the ATTRN function). In the loop, the macro calls the VARNAME function, passing it the file identifier of the open dataset and the variable position. This returns the variable name at the Ith position, which is then appended to the macro variable VarList. After exiting the loop, the file is closed and a list of all the variables in the dataset is 'returned'.

This macro reveals the power of the %SYSFUNC function in producing efficient routines. SAS® programmers can create macros to accomplish many data oriented tasks just by combining DATASTEP functions with MACRO code.

PUTTING IT ALL TOGETHER

Thus far, this article has shown how to perform pattern-matching in the SAS® System and how to interrogate a SAS® dataset for content information. The final goal of this paper is to combine these two techniques into one macro that will scan a dataset for multiple patterns and return a unique variable list.

As the output is fairly well defined, now is a good time to look at what inputs are needed. A minimal set of inputs to the macro should include a) a dataset name, b) variable type desired, c) variable names to exclude, and d) the pattern to match against.

It makes sense to allow the input pattern to be a space-delimited list of patterns. This makes the macro easier to use if the user wants to specify multiple, but separate, patterns to create one variable list. Additionally, if no pattern is specified, the macro should return all variable names in the dataset, while respecting variable type and exclusion parameters.

Finally, it should be a simple task to allow the excluded variable names parameter to be a list of patterns. This should help the macro meet the "Ease-of-Use" criteria specified earlier.

Based on the above criteria, the pseudocode for the macro, named BuildVar, is as follows:

```
1) initialize local variables
2) check input parameters
3) check access to dataset
4) if patternlist is blank then patternlist = :
5) if excludelist has patterns then execute
    BuildVar with patternlist = excludelist

6) while patternlist is not blank do
7)   extract pattern from patternlist
8)   extract type from typelist
9)   do for each variable in dataset
10)     if variable is not in excludelist then
11)       perform pattern-matching against variable
          name using pattern and type information
12)       if result of match not in variable list then
            append result to variable list
13)   end
14) end

15) close dataset
16) return variable list
```

Step 4) is required to ensure that the macro complies with the specifications: if no pattern is supplied, return all of the variable names (respecting type and exclusion information). When no pattern is given, the if statement forces the macro to enter the loop and match all of the variable names in the dataset.

BUILDVAR EXAMPLES

Suppose the dataset TEST has the structure shown in the following table:

Variable Name	Variable Type
TRT	Num
ENDWGT	Num
BEGWGT	Num
ENDGKM	Num
END1GKM	Num
END2GKM	Num
END1LKM	Num
END2LKM	Num
ENDPT1	Num

If a user needs to create a variable list from TEST that contains all of the numeric variables that have a prefix of 'END' and a suffix of 'GKM', the following code does the trick:

```
   %BuildVar(TEST,N,, END:GKM) ;
```

The above macro call will return ENDGKM END1GKM END2GKM. To create a variable list of all the numeric variables that have a suffix of 'WGT', simply use:

```
   %BuildVar(Test,N,, :WGT) ;
```

which will return ENDWGT BEGWGT.

By compounding the colon operator, a user can create a variable list of all the numeric variables that have a '1' in their name:

```
   %BuildVar(Test,N,, :1:) ;
```

The above macro call will return END1GKM END1LKM ENDPT1. A nice feature of the BuildVar macro is that it returns a variable list in which all variable names are unique. An example of this would be the following code:

```
   %BuildVar(Test,N,, :1: END:GKM ) ;
```

This searches for all variable names that contain a ‘1’ and then all variable names that have a prefix of ‘END’ and a suffix of ‘GKM’. This macro call returns END1GKM END1LKM ENDPT1 ENDGKM ENG2GKM. Note that the variable END1KM is listed once, and that the order of the patterns can affect the order of the variable names.

While the BuildVar macro supports any valid pattern, users should initially concentrate on a subset of patterns, and build on these. One operator that SAS® users will find natural to use is the colon, as it is an extension of the built-in operator. Other operators include the ^ (carat), which returns the complement of the pattern. This allows users to select variables that don’t match a pattern. Indeed, the colon and the complement operators should cover most situations.

CONCLUSION

Pattern-matching in SAS® is a method of efficiently and quickly searching character data for text. The pattern-matching functions in SAS® have been explained and examples provided to show how to use them. While this technique may be somewhat difficult to use at first, it is a very powerful tool that all programmers should learn and embrace.

Every SAS® dataset contains information on its structure and content. Some of the SAS® File I/O functions that access this information have been described, and techniques to use them have been demonstrated in SAS® MACRO with the %SYSFUNC macro function. While it is not possible to cover the other dataset access functions and all of their options (ATTRN can return information on 26 different attributes), the reader is encouraged to peruse the help files and experiment with them.

Finally, this paper has shown how to marry these two distinct techniques to yield a product that builds on the capabilities of SAS® in a natural way, using syntax that is familiar to any SAS® user.

REFERENCES

Extensive information on pattern-matching in SAS® can be downloaded from:

ftp://ftp.sas.com/pub/neural/patt612.txt
ftp://ftp.sas.com/pub/neural/rx612.txt

A regular expression reference with a focus on Perl used by SAS® technical support:

Mastering Regular Expressions, Jeffrey Freidl (O'Reilly, 1997) ISBN: 1-56592-257-3.

CONTACT INFORMATION

The complete SAS® code can be obtained from the author by emailing the address below.

Your comments and questions are valued and encouraged. Contact the author at:
Michael P.D. Bramley, Site Manager
Venturi Technology Partners

223 Martha Lane
Fairfield, OH 45014

Email: mbramley@att.net