Paper 004-2012

Use the Full Power of SAS® in Your Function-Style Macros

Mike Rhoads, Westat, Rockville, MD

ABSTRACT

Function-style macros are macros that can be used within a SAS statement. Although such macros are extremely flexible, they can only use macro language code.

So what do you do if you need a macro that can be embedded within a SAS statement, but the underlying task requires the execution of one or more DATA or PROC steps? You can now leverage the capabilities of user-written SAS functions to circumvent this limitation, meaning that your macros can now have the flexibility of a function-style interface while still being able to execute one or more SAS steps in the background. This paper describes the necessary techniques for developing such macros and provides examples of situations where they can be most beneficial.

FIRST, A WORD OF WARNING

The techniques described in this paper should work, but I have run into some technical issues in trying to implement them. Therefore, through at least the first maintenance release of SAS 9.3 (TS1M1), you will need to exercise some care when employing these methods, taking into consideration the issues that are mentioned in this section. According to SAS Technical Support, these problems will be rectified in a later release of SAS.

In SAS 9.2, the most direct limitation is that values passed from a user-written function to a macro through the RUN_MACRO function cannot be longer than 262 characters. This is problematic for situations where a list of variables is to be returned, since when many variables are involved this limit can easily be exceeded. I also ran into some other difficulties with these routines in 9.2, including the “once and done” problem described below. Therefore, I would not recommend making any serious use of these techniques under SAS 9.2.

When SAS 9.3 arrived, I was delighted to find that the 262-character limit was gone. My initial testing went well, until one particular situation where I tried to run the same macro a second time within the same SAS session. This produced a long pause, which finally ended with the appearance of a dialog box saying that SAS had encountered a problem and needed to close. It turned out that, under the initial release of SAS 9.3, calling one of the macros using the techniques described below from within DATA step or PROC step code would only work once within a SAS job; subsequent calls to that macro would produce an error. This is scheduled to be fixed in the next release.

While finalizing this paper, I did figure out a workaround that alleviated the problem. The key step is to precede each step that calls the macro with a call from within a %LET or %PUT statement in open code, outside a DATA step, PROC step, or macro. For instance:

```
%LET DUMMYVAR = %ExpandVarList(data=sashelp.class);
```

By adding this dummy call prior to every step that contained an embedded call to the same macro, I was able to use the macro multiple times within the same program without any problems.

INTRODUCTION

This paper was actually inspired by a SAS-L listserv posting in December of 2010. The poster was searching for a good way to take a macro parameter value containing a SAS variable list in any form (e.g., A1-A3 FIRST—LAST VARA-NUMERIC-VARZ) into a simple list of all of the referenced variables (e.g., A1 A2 A3 FIRST MIDDLE LAST VARA VARM VARN VARZ). Quite a few possible solutions were discussed; some of these returned the variable names in the same order in which they appeared in the original data set, while others maintained the order of the specified variable list. Of the latter, my favorite approach due to its simplicity was one suggested by legendary SAS-L poster data _null_ (John King), which utilized a PROC TRANSPOSE with OBS=0 as an option on the input data set.

```
proc transpose  data=sashelp.class(obs=0)  out=varlist;
   var name-numeric-weight name-character-weight; /* or some other specification */
   run;
```

This can then easily be followed by a PROC SQL invocation to produce a macro variable containing the expanded list of variables in the desired order.

```
proc sql noprint;
   select _name_ into :varlist separated by ' '
```

```
      from varlist;
   quit;
```

Both elegant and effective – why write code to manually parse the string and handle all of the possible types of variable lists when you can get a SAS PROC to do the work for you? Unfortunately, it seemed that the approach did have one major drawback. Let's say I am developing a macro that runs a report on each of a specified set of variables from a data set. The user is to invoke the macro as follows:

```
%RunReports (data=mydata, var=A1-A3 FIRST—LAST VARA-NUMERIC-VARZ)
```

Within the %RunReports macro, I would like to call a standard utility macro to expand the user's specified list. Ideally this would be invoked from within %RunReport in the following manner.

```
%let varlist = %ExpandVarList(data=&DATA, var=&VAR);
```

This style of invocation requires that %ExpandVarList be a **function-style macro**, meaning that it is used by embedding it within some other SAS statement or SAS macro statement – in the example above, a macro %LET statement. The name comes from the resemblance of such macros to SAS functions; they are (usually) passed one or more arguments, and they return text that becomes part of the statement within which they are called, just as SAS functions are called as the right-hand-side of an assignment statement, or as part of an expression.

Since calls to function-style macros are located within SAS statements and return text that will become part of that statement when it is parsed by SAS, their big limitation has always been that they could not submit any code themselves. All of their processing had to be accomplished using the SAS macro language. Although the macro language does have functions that allow some limited processing of SAS data sets and external files, they are much less powerful than the integrated set of capabilities available with the full panoply of SAS PROCs and DATA step functionality.

Fortunately, this restriction no longer exists! The remaining sections of this paper discuss the new SAS functionality that eliminates this previous limitation, and provide examples of how these capabilities can help you develop enhanced SAS utility macros that can extend the power and flexibility of your SAS programs.

USER-WRITTEN FUNCTIONS: THE KEY THAT UNLOCKS THE DOOR

SAS has long had a rich set of built-in functions for manipulating one or more numeric or character arguments and returning a result – who among us has not used SUBSTR, SCAN, ROUND, or MAX? These functions and the many others found in recent SAS releases provide a reliable, tested mechanism for implementing simple and complex algorithms. Using them makes program development simpler and much more robust.

Even though the SAS system has several hundred of these built-in functions, the SAS development team could hardly anticipate or implement every possible function that someone might want. So, what can we SAS programmers do when we need a function that does not exist in our licensed SAS components? Up until recently, there has not really been a satisfactory mechanism for us to "roll our own" functions. LINK and RETURN allow for program control to be transferred and returned within a DATA step, but they do not support parameter passing or local variables. Other alternatives, such as using the macro language or implementing functions that are written in C or Java, have their own set of drawbacks.

Fortunately, SAS 9.2 finally addressed this problem. SAS programmers can now create functions (as well as CALL routines) in PROC FCMP that can be invoked from DATA steps in a very similar manner to using built-in SAS functions. These functions are written using DATA step syntax, support parameter passing, and permit using variables that are local to the function. Since they can be called recursively, they allow problems to be solved that were difficult to handle with traditional DATA step programming. For instance, Secosky (2007) provides a detailed example of a PROC FCMP subroutine that can be used to traverse a directory hierarchy.

Since these new user-written functions are primarily designed to be called within DATA steps (although they can also be used in PROC REPORT and PROC SQL), they do not seem to directly address our interest in expanding the capabilities of function-style macros. However, two little-noticed features of this enhancement provide the key to the technique that is described below:

1. Functions created in PROC FCMP can be called from macros using %SYSFUNC.
2. PROC FCMP functions can use a special function called RUN_MACRO to invoke a SAS macro. The capability has been discussed in recent papers by Christian and Rioux (2010) and Poling (2011).

Taken together, these two features allow us to apply a simple three-step technique for implementing function-style macros that can use the full power of SAS DATA steps and procedures. The steps are as follows:

1. **Outer macro.** This is the "visible" macro that users will reference in their programs. It does not perform any actual processing, except perhaps for error checking. Its purpose is to invoke a user-written function with %SYSFUNC, passing the user's arguments to the user-written function and providing the user with the returned value (generated text).
2. **User-written function.** This routine also does not perform any actual processing, but serves as a "bridge" between the outer and inner macros. Its main purpose is to accept parameters from the outer macro and pass them to an "inner" macro, which it invokes using RUN_MACRO. It also accepts the text generated by the inner macro and passes it back to the outer macro.
3. **Inner macro.** This is where the actual work gets done. Using the parameter values specified by the user in the outer macro and passed along by the user-written function, the inner macro may contain whatever sequence of DATA or PROC steps is necessary to generate the desired results. Once these steps have been executed and the desired text has been obtained, the inner macro sends the text back to the user-written function.

It seemed to me that this technique was useful enough to deserve a catchy name, but I was singularly unsuccessful in attempting to think of one. Fortunately my long-time colleague Michael Raithel came to the rescue; since we wind up with two macros with a user-written function in the middle, he suggested that we call it the “Macro Function Sandwich.” The examples below demonstrate how you can use the Macro Function Sandwich (MFS) to implement some useful macro-style functions.

EXAMPLE 1: EXPANDING A USER-SPECIFIED VARIABLE LIST

This first MFS example is based on the scenario described in the introduction. We want to implement a general utility macro, %ExpandVarList, that takes as arguments the name of a SAS data set and a list containing the names of some of the variables in the data set. The macro is to return an "expanded" version of the user-specified variable list, which lists the name of each variable individually, with a blank space between each one. We want this to be a function-style macro, but since the desired implementation strategy uses PROC TRANSPOSE and PROC SQL, we will have to utilize the technique described above, with a user-written function serving as the intermediary between the outer macro that the user calls and an inner macro that actually does the work.

OUTER MACRO

The code for the "outer macro" shown below is trivial. The macro accepts two keyword-style parameters: DATA= to indicate the data set, and VAR= to specify the variable list. _LAST_ (referring to the most recently created SAS data set) and _ALL_ (which specifies all of the variables in a data set) serve as default values if the user omits the corresponding argument in the macro invocation. If the DATA= parameter has the value of _LAST_, either by default or explicitly, the macro resolves this value by assigning the value of the automatic macro variable &SYSLAST to &DATA. (I found that simply passing _LAST_ through did not work reliably.) The only other action the macro has to perform is to invoke the user-written function. This is done using the %SYSFUNC function, just as if you were calling a built-in SAS function. (Note that character arguments to functions are NOT quoted when the function is called in macro via %SYSFUNC.) The call is nested within the %TRIM function to make sure that any trailing blanks are removed. Since the call to the user-written function stands alone rather than being embedded within a macro language statement, the text returned by the user-written function will also be returned as the result of the macro call.

Note that the invocation of the user-written function should NOT have a semicolon at the end, since if it does the semicolon will be included in the returned value. This is a relatively subtle error, since the macro will still work correctly if it is called at the end of a statement, which is frequently the case. It will not work, however, if it followed by other text that is intended to be part of the original statement, since the statement will be ended by the returned semicolon.

It is, of course, possible to make the macro more involved. In a production environment, it might be desirable to do a certain amount of error-checking, such as making sure that the specified data set actually exists. Additional parameters could also be specified and processed, such as the desired delimiter to separate the variable names when the list is returned. (For use in a PROC SQL SELECT statement, for instance, the user would probably prefer the result to have comma-delimited names.)

```
%MACRO ExpandVarList (
  data=_LAST_,
  var=_ALL_
);
  %if %upcase(%superq(data)) = _LAST_
    %then %let data = &SYSLAST;
  %trim(%sysfunc(ExpandVarList(&DATA,&VAR)))
%MEND ExpandVarList;
```

USER-WRITTEN FUNCTION

The code for the user-written function that will serve as a go-between for the outer and inner macros is shown below.

```
proc fcmp  outlib=sasuser.mysubs.utility;
  function ExpandVarList(data $,var $) $ 32000;
    length data $ 48  var $ 2048  varnames $ 32000;
    rc = run_macro('ExpandVarList_Inner',data,var,varnames);
    if rc = 0 then return(trim(varnames));
    else return('*** ERROR ***');
  endsub;
run;
```

User-written functions and subroutines are stored in "packages" within SAS data sets, where a package is defined as a set of routines (presumably related) that each have unique names. On the **PROC FCMP statement** in this example, the OUTLIB= argument indicates that the function will be created in the data set SASUSER.MYSUBS, within a package called UTILITY.

The function definition begins with a **FUNCTION statement** and ends with an **ENDSUB statement**. The parenthetical on the statement specifies that the function accepts two arguments, DATA and VAR. Each of these is a character string, specified with the $ sign. Note that DATA and VAR don't have to be the same names as were used for the outer macro's parameters, although they are in this case. Similarly, the user-written function does not have to have the same name as the macro that invokes it.

The $ 32000 at the end of the FUNCTION statement indicates that the function will return a character string that can have a length of up to 32000 characters.

The **LENGTH statement** assigns lengths for the character variables we will be using. We first specify lengths for the two arguments we listed in the FUNCTION statement: DATA and VAR. We will be using the variable VARNAMES to hold the expanded variable list that will be returned when we call the inner macro, so we give it a length of 32000. Note that all of these variables are local in scope to the user-written function; they will cease to exist once the function finishes executing.

In the following statement, the **RUN_MACRO function** is used to invoke the inner macro, which we will call ExpandVarList_Inner. The three character variables listed in the LENGTH statement are passed to the macro. DATA and VAR pass along the values the user specified when calling the outer macro. VARNAMES is the variable into which the macro will place the expanded list of variables.

The **RETURN statement** returns the result of the user-written function back to the calling program (in our example, the outer macro). If RUN_MACRO was able to successfully invoke the %ExpandVarList_Inner macro, the returned value (RC) will be 0, and we will return the trimmed and expanded variable list as supplied by the macro. If RUN_MACRO did not succeed, we will return a predefined error string.

Note that a zero value for RC only indicates that RUN_MACRO was able to successfully invoke the macro; it might still be the case that the macro itself identified some sort of error. A more robust implementation would pass an additional variable from the user-written function to the inner macro, which the latter could use to flag any errors it may have found.

INNER MACRO

Finally, here's the code for the macro that actually does the work.

```
%macro ExpandVarList_Inner;
  %let data=%sysfunc(dequote(&data));
  %let var =%sysfunc(dequote(&var));
  %local temp_varnames;
  proc transpose  data=&DATA (obs=0)  out=ExpandVarList_temp;
    var &VAR;
  run;

  proc sql noprint;
    select _name_ into :temp_varnames separated by ' '
      from ExpandVarList_temp
    ;
    drop table ExpandVarList_temp;
  quit;
  %let varnames = %trim(&temp_varnames);
```

```
%mend ExpandVarList_Inner;
```

The first three statements of the macro reflect the way the RUN_MACRO function actually works. When the RUN_MACRO call includes variables that are to be passed to the macro, RUN_MACRO actually creates a corresponding macro variable for each variable in the RUN_MACRO call, and copies the value from the variable in the user-defined function into the macro variable. Once the macro completes, RUN_MACRO copies the values back from the macro variables into the user-written function's variables. This explains why the %MACRO statement does not include a parameter list, as you would otherwise expect.

Another characteristic of RUN_MACRO is that it puts quotation marks around the values of character variables as it copies them into the corresponding macro variables. This explains the need to DEQUOTE the values of DATA and VAR before we use them.

Most of the remaining code in the inner macro actually implements the algorithm to expand the variable list. At the end, it trims the text in the local macro variable TEMP_VARNAMES, and places it back into the variable VARNAMES, from which RUN_MACRO will copy the value back into the variable declared in the user-written function.

PUTTING IT ALL TOGETHER

Once you have defined the outer and inner macros and the user-written function shown above, you can use code such as the following to invoke the outer macro.

```
options  cmplib=sasuser.mysubs;

data d1;
  retain CHAR ' ' X1 X2 X3 X4 X5 Y Z JUNK1 JUNK2 AGE 0 NAME '        '
    SEX ' ' HEIGHT WEIGHT JUNK3 0;
run;

%PUT Final result is %ExpandVarList(data=d1, var=junk2 x3-x5 age-numeric-weight);
```

The only new concept in this block of code is the CMPLIB= system option, which tells SAS where to look to find any user-written functions that are referenced in your program. In this case, SAS will look in the SASUSER.MYSUBS data set. If more than one package within the data set contained a function called ExpandVarList, the call to it in the other macro would have to be qualified with the package name (e.g., %trim(%sysfunc(**Utility.ExpandVarList**(&DATA,&VAR)));).

The important thing, of course, is that the macro that you called within a %PUT statement has caused a PROC TRANSPOSE and a PROC SQL to be executed invisibly behind the scenes, thus enabling you to use the full power of SAS to implement the variable list expansion.

EXAMPLE 2: USING AN SQL SELECT STATEMENT AS INPUT TO A DATA STEP

In SAS, the DATA step and PROC SQL each have their own advantages. PROC SQL offers a flexible and relatively standardized syntax for manipulating data tables, and it can perform operations such as Cartesian products in a much more natural fashion that is possible with DATA step coding. On the other hand, the procedural statements in the DATA step allow it to accomplish certain tasks, such as those requiring the recognition of the first or last record in a group, that are difficult or impossible to carry out with SQL.

To take full advantage of the complementary capabilities of these two SAS programming techniques, it would be wonderful if we could use an SQL query directly within DATA step syntax, something like the following:

```
data maledata;
  set %GetSQL(select name, age from sashelp.class where sex = 'M' order by age);
  /* DATA step logic here */
run;
```

Here again, what initially looks to be impossible – after all, you can’t suddenly break into SQL in the middle of compiling a DATA step – turns out to be quite achievable through the use of a function-style macro used in an MFS. As a bonus, you can also use such a macro with DATA= in a PROC statement.

OUTER MACRO

```
%MACRO GetSQL / PARMBUFF;
  %let SYSPBUFF = %superq(SYSPBUFF);
  %let SYSPBUFF = %substr(&SYSPBUFF,2,%LENGTH(&SYSPBUFF) - 2);
```

```
    %let SYSPBUFF = %superq(SYSPBUFF);
    %let SYSPBUFF = %sysfunc(quote(&SYSPBUFF));
    %local UNIQUE_INDEX;
    %let UNIQUE_INDEX = &SYSINDEX;
    %sysfunc(GetSQL(&UNIQUE_INDEX,&SYSPBUFF))
  %MEND GetSQL;
```

As with Example 1, the job of the outer “wrapper” macro is to accept arguments from the user, send them along to the user-written function, and get the final returned text string back to the user. We need to use some different techniques to accomplish this, however, than in the previous example.

Here, the user will only be passing in one argument: an SQL SELECT statement. Since this may well contain commas, however, we use the PARMBUFF option to obtain the entire string that the user enclosed within parentheses when calling the macro, which is placed into the special macro variable &SYSPBUFF. As that value includes the leading and trailing parentheses, we first use %SUBSTR to remove them. We also must make sure that any commas are not misinterpreted as argument separators, so we use the system QUOTE function to place quotation marks around the query string before we pass it to the user-written function. The %SUPERQ calls are included to make sure that the function calls following them are executed properly.

It is possible for a user to call GETSQL more than once in a single DATA step, for example:

```
  merge %GetSQL(select-statement-1) %GetSQL(select-statement-2);
```

For this reason, we need to make sure that each invocation creates a different view, rather than overwriting the same view over and over. One obvious approach is to append a unique number to the name of the PROC SQL view when it is created in the inner macro. The first iteration of this code used &SYSINDEX within the inner macro to do this, since this automatic macro variable is incremented each time a macro is invoked “within a SAS job or session”. Unfortunately, user-written functions and any macros they call exist within a separate session from the main job, so the &SYSINDEX values were always 1. This result can be circumvented by referencing &SYSINDEX within the outer macro, and passing its value through the user-written function to the inner macro.

USER-WRITTEN FUNCTION

```
  proc fcmp  outlib=sasuser.mysubs.utility;
    function GetSQL(unique_index_2, query $) $ 32;
      length query query_arg $ 32000  viewname $ 32;
      query_arg = dequote(query);
      rc = run_macro('GetSQL_Inner', unique_index_2, query_arg, viewname);
      if rc = 0 then return(trim(viewname));
      else return('*** ERROR ***');
    endsub;
  run;
```

Since all the user-written function needs to do is pass arguments between the outer and inner macros, the code for this function looks very much like the code for the ExpandVarList function in Example 1. However, as we saw above, we had to quote the query string when passing it in from the outer macro; otherwise, any commas in its value would have been interpreted as separators in the list of arguments. For this reason, we use DEQUOTE to remove the quotation marks before using the value in the RUN_MACRO call (which adds its own set of quotation marks anyway).

INNER MACRO

```
  %MACRO GetSQL_Inner;
    %local query;
    %let query = %superq(query_arg);
    %let query = %sysfunc(dequote(&query));
    %let viewname = GetSQL_tmpview_&UNIQUE_INDEX_2;
    proc sql;
      create view &viewname as &query;
    quit;
  %MEND GetSQL_Inner;
```

As can be seen in the code above, the SAS technique we use to accomplish the “embedded SQL” within the DATA step is to call PROC SQL from the inner macro, creating a temporary view using the SQL SELECT statement specified by the user in the call to the outer macro. The macro variable VIEWNAME contains the name of the view being created, which is then passed back through the user-written function to the outer macro, where it becomes a value in a SET or MERGE statement, or perhaps in the DATA= option of your favorite SAS procedure.

In addition to VIEWNAME, the variables QUERY_ARG and UNIQUE_INDEX_2 are used to communicate between the user-written function and the inner macro. QUERY_ARG is the query passed by the user. As we noted above, RUN_MACRO surrounds character values with quotation marks when it passes them, so we need to remove these first. Before we do so, however, we need to macro-quote the value (using %SUPERQ) so that any commas within the query do not cause problems for the subsequent call to DEQUOTE. UNIQUE_INDEX_2 is a numeric value that will be used as the suffix for the name of the view.

PUTTING IT ALL TOGETHER

The code below shows how the %GetSQL macro could be used. In most real-word applications, the SQL code would probably be more complicated.

```
data example;
  set %GetSQL
    (select name, age, weight from sashelp.class where sex = 'M' order by age)
      %GetSQL
    (select name, age, height from sashelp.class where sex = 'F' order by name)
  ;
run;

proc print  data=example;
run;
```

EXAMPLE 3: FINDING VARIABLES WHOSE NAMES MATCH A PATTERN

SAS has many forms of variable lists, but none of them support a true wild-card capability – for instance, perhaps you'd like to run a PROC UNIVARIATE on all variables in a DATA set whose names begin with A and end with PCT. This capability could be implemented with an MFS, which could be called as follows:

```
proc univariate  data=sashelp.demographics;
  var %VarPattern(data=sashelp.demographics, pattern=/^a.*pct *$/i);
run;
```

The argument to PATTERN= would be a Perl pattern-matching expression. (Note that the value of PATTERN would need to be enclosed in a macro quoting function such as %NRSTR if it contained any special characters that might confuse the macro processor.) The underlying functionality is provided by utilizing the PRXMATCH function and PROC SQL.

OUTER MACRO

```
%MACRO VarPattern (
  data=_LAST_, pattern=/.*/
  );
  %if %upcase(%superq(data)) = _LAST_
    %then %let data = &SYSLAST;
  %sysfunc(VarPattern(&DATA,&PATTERN))
%MEND VarPattern;
```

USER-WRITTEN FUNCTION

```
proc fcmp  outlib=sasuser.mysubs.utility;
  function VarPattern(data $, pattern $) $ 32000;
    length data $ 48  pattern $ 200  varnames $ 32000;
  rc = run_macro('VarPattern_Inner',data,pattern,varnames);
  if rc = 0 then return(trim(varnames));
  else return('no_such_var  /* ERROR */');
  endsub;
run;
```

INNER MACRO

```
%MACRO VarPattern_Inner;
  %let data = %sysfunc(dequote(&data));
```

```
    %let data = %upcase(&data);
    %local libname memname;
    %let memname = %scan(&data,-1,.);
    %let libname = %scan(&data,-2,.);
    %if %length(&libname) = 0
      %then %let libname = WORK;
    proc sql  noprint;
      select name
        into :varnames separated by ' '
      from dictionary.columns
      where libname = "&LIBNAME"
        and memname = "&MEMNAME"
        and prxmatch (&pattern,name)
      order by varnum;
    quit;
  %MEND VarPattern_Inner;
```

The outer macro and the user-written function above are almost identical to those used in earlier examples, and so are not discussed here. The first several lines of the inner macro break out the user's data set specification into a library name and member name, after first uppercasing the specification. (The SAS dictionary tables store these values as upper-case characters.) They also assign WORK as the libname if the user supplied a one-level data set name. The PROC SQL code should look familiar to anyone who has extracted variable names for a particular data set from the SAS dictionary tables. The main difference here from other examples of this technique is the use of the PRXMATCH function to return all of the variables matching the regular expression provided in the macro call.

PUTTING IT ALL TOGETHER

```
  proc univariate  data=sashelp.demographics;
    var %VarPattern(data=sashelp.demographics, pattern=/^a.*pct *$/i);
  run;
```

This example is very similar to the one at the beginning of this section, except here we want to process all variables whose names start with "a" and end with the string "pct". The table below deciphers the regular expression that is the value of PATTERN=; while certainly not a comprehensive introduction to Perl regular expressions, it illustrates many of the most useful constructs for matching variable names.

/ ... /i	The two slashes identify the beginning and end of the pattern. The "i" at the end specifies that matches should ignore case, so it won't matter whether the user's pattern contains uppercase or lowercase letters.
^a	The "^" matches the beginning of the "line". Since this is immediately followed by "a", this indicates that the variable name must begin with the letter "a".
.*	Together, the "." (match any character) and the "*" (match 0 or more times) indicate that there may be any number of other characters between the "a" and the next more specific element to be matched.
pct	The string "pct" must be found in the variable name.
*$	We don't just want "pct" to be somewhere in the variable name, but we want it to be at the end. Therefore, the only thing that can be in the dictionary table entry for the variable name following the "pct" can be trailing spaces. The "$" matches the end of the "line", and this is preceded by a blank and an asterisk. Together, these indicate that, to complete the match, the "pct" must be followed by 0 to many blanks, and then the end of the string.

OTHER POSSIBLE USES

Once you start thinking in terms of the Macro Function Sandwich strategy, you will probably come up with even more situations in SAS that would benefit from its use.

Transparent access to zipped files. You could develop an MFS to allow zipped SAS data sets to be referenced wherever an input SAS data set can be specified. Behind the scenes, the inner macro would extract the data set from the WinZip archive into the SAS work library. The syntax below assumes that the zip file contains a single SAS data set whose name is the same as the name of the zip file, but you could certainly add an argument that would specify the SAS data set explicitly. You could also add arguments, or develop a parallel function, to handle non-SAS files as well.

```
data RevisedFile;
  set %Unzip(C:\Project7\Data Archive\Class.zip);
  /* DATA step logic here */
run;
```

Parallel arrays. On several occasions I have been in a situation where I have needed to write a DATA step where one array consists of some subset of variables from an input data set, and I then need to create a second array with the same number of elements as that first array. What I'd really like to write is the following pair of statements:

```
array one {*} somevar--othervar;
array flag {dim(one)};
```

Unfortunately, although this seems perfectly logical to me, since SAS knows the dimension of array ONE as soon as it is declared, it is not valid DATA step syntax. With an MFS very similar to the %ExpandVarList function described above, however, we could our achieve our objective with only a small amount of additional coding:

```
array one {*} somevar--othervar;
array flag {%VarCount(data=mydata, var=somevar--othervar)};
```

CONCLUSION

The addition of user-written functions to the SAS programmer's toolkit has expanded our options for creating powerful and flexible SAS programs that are clearly-written and easy to maintain. In particular, the RUN_MACRO function now allows us to branch out from a SAS DATA step and execute other SAS data and PROC steps that are contained in a SAS macro.

The Macro Function Sandwich" technique elaborated in this paper takes advantage of these innovations to eliminate the previous limitations of function-style SAS macros. As the examples illustrate, there are many situations in which this technique can effectively simulate features that are missing from the SAS language. If you have ever complained about some missing capability in SAS, consider whether you may be able to effectively implement it yourself, using a Macro Function Sandwich.

DISCLAIMER: The contents of this paper are the work of the author and do not necessarily represent the opinions, recommendations, or practices of Westat.

REFERENCES

Christian, Stacey M. and Jacques Rioux. "Adding Statistical Functionality to the DATA Step with PROC FCMP." *Proceedings of the SAS® Global Forum 2010 Conference.* April 2010. <http://support.sas.com/resources/papers/proceedings10/326-2010.pdf> (March 10, 2012)

Eberhardt, Peter. "A Cup of Coffee and Proc FCMP: I Cannot Function Without Them." *Proceedings of the PharmaSUG 2011 Conference.* May 2011. <http://www.lexjansen.com/pharmasug/2011/tu/pharmasug-2011-tu07.pdf> (March 10, 2012)

King, John (Data _null_). "Re: Variable list into a single macro variable XXXX." Posting to the SAS-L listserv. December 15, 2010. <http://www.listserv.uga.edu/cgi-bin/wa?A2=ind1012C&L=sas-l&D=0&P=6436> (October 29, 2011)

Poling, Jeremy W. "Can't Decide Whether to Use a DATA Step or PROC SQL? You Can Have It Both Ways with the SQL Function!" *Proceedings of the 2011 SCSUG Educational Forum.* November 2011. <http://www.scsug.org/SCSUGProceedings/2011/poling1/SQL%20Function.pdf> (March 10, 2012)

Secosky, Jason. "User-Written DATA Step Functions." *Proceedings of the SAS® Global Forum 2007 Conference.* April 2007. <http://www2.sas.com/proceedings/forum2007/008-2007.pdf> (March 10, 2012)

ACKNOWLEDGMENTS

Thanks to Westat colleague Michael Raithel, both for his perceptive review of this paper and for coming up with the extremely apt "Macro Function Sandwich" appelation for this programming technique. Thanks also to Amber Elam, Stacey Christian, and Jason Secosky from SAS for patiently working with me in my bleeding-edge experiments with the boundaries of user-written functions.

CONTACT INFORMATION

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

Mike Rhoads
Westat
1600 Research Blvd.
Rockville, MD 20850
RhoadsM1@Westat.com